ЭКСПЕРТНЫЕ СИСТЕМЫ

принципы разработки и программирование

ЧЕТВЕРТОЕ ИЗДАНИЕ

tiutla-Expert Systems.indd 1 21.11.2006 18:47:56

EXPERT SYSTEMS

principles and programming

FOURTH EDITION

Joseph C. Giarratano
University of Houston-Clear Lake

Gary D. Riley
PeopleSoft, Inc.

THOMSON
COURSE TECHNOLOGY

tiutla-Expert Systems.indd 2 21.11.2006 18:47:56

ЭКСПЕРТНЫЕ СИСТЕМЫ

принципы разработки и программирование

ЧЕТВЕРТОЕ ИЗДАНИЕ

Джозеф Джарратано
Университет Хьюстон-Клиэр-Лэйк

Гари Райли
PeopleSoft, Inc.



Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2007

tiutla-Expert Systems.indd 3 21.11.2006 18:47:56

ББК 32.973.26–018.2.75
Д40
УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *К.А. Птицына*

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419 , Москва, а/я 783; 03150, Киев, а/я 152

Джарратано, Джозеф, **Райли**, Гари.

Д40 Экспертные системы: принципы разработки и программирование, 4-е издание. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2007. — 1152 с. : ил. — Парал. тит. англ.

ISBN 978–5–8459–1156–8 (рус.)

Данное четвертое издание представляет собой результат существенного пересмотра известного во всем мире учебника по экспертным системам и разработке программного обеспечения с помощью инструментария языка экспертных систем CLIPS. Книга включает сведения, относящиеся к двум основным направлениям: в первой половине книги излагается теория экспертных систем и показано, какое место занимают экспертные системы во всем объеме компьютерных наук, а во второй приведены сведения по программированию с помощью языка CLIPS. Еще одним новым средством, описанным в данном издании, является объектно-ориентированный язык COOL. В начале книги содержится отдельное введение в тематику искусственного интеллекта, объем которого достаточен для изучения экспертных систем.

Теоретический материал изложен на уровне, доступном для восприятия студентов старших курсов и аспирантов, интересующихся экспертными системами, которые специализируются в области компьютерных наук, информационных управленческих систем, в программотехнике и других областях. Книга может оказаться полезной для широкого круга читателей, желающих применить экспертные системы в своей работе.

ББК 32.973.26–018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Course Technology.

Authorized translation from the English language edition published by Course Technology, a division of Thomson Learning, Inc. Copyright © 2005

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein.

ISBN 978–5–8459–1156–8 (рус.)
ISBN 0–534–38447–1 (англ.)

© Издательский дом “Вильямс”, 2007
© Course Technology, 2005

Оглавление

Предисловие 18
Глава 1. Введение в экспертные системы 27
Глава 2. Представление знаний 127
Глава 3. Методы логического вывода 193
Глава 4. Рассуждения в условиях неопределенности 295
Глава 5. Нестрогие рассуждения 389
Глава 6. Проектирование экспертных систем 505
Глава 7. Введение в CLIPS 551
Глава 8. Развитые средства сопоставления с шаблонами 625
Глава 9. Модульное проектирование, управление выполнением и эффективность правил 693
Глава 10. Процедурное программирование 779
Глава 11. Классы, экземпляры и обработчики сообщений 851
Глава 12. Примеры проектов экспертных систем 949
Приложение А. Некоторые широко применяемые эквивалентности 1015
Приложение Б. Некоторые элементарные кванторы и их значение 1017
Приложение В. Некоторые свойства множеств 1019
Приложение Г. Информация о поддержке CLIPS 1021
Приложение Д. Общие сведения о командах и функциях CLIPS 1023
Приложение Е. Определение языка в нормальной форме Бэкуса–Наура 1059
Приложение Ж. Программные ресурсы 1069
Приложение З. Литература 1119
Предметный указатель 1125

Содержание

Предисловие 18

Глава 1. Введение в экспертные системы 27
1.1 Введение 27
1.2 Определение понятия экспертной системы 27
1.3 Преимущества экспертных систем 38
1.4 Общие понятия экспертных систем 39
1.5 Характеристики экспертной системы 43
1.6 Разработка технологии экспертных систем 47
Решение задач человеком и продукционные правила 48
Широкое распространение систем, основанных на знаниях 54
1.7 Приложения и предметные области экспертных систем 57
Приложения экспертных систем 57
Наиболее подходящие области применения экспертных систем 61
1.8 Языки, командные интерпретаторы и инструментальные средства 65
1.9 Элементы экспертной системы 69
1.10 Продукционные системы 77
Продукционные системы Поста 78
Марковские алгоритмы 81
Rete-алгоритм 82
1.11 Процедурные подходы 85
Императивное программирование 86
Функциональное программирование 89
1.12 Непроцедурные подходы 94
Декларативное программирование 94
Объектно-ориентированное программирование 94
Логическое программирование 96

	Экспертные системы	100
	Недекларативное программирование	102
	Программирование на основе индукции	102
1.13	Искусственные нейронные системы	103
	Задача коммивояжера	105
	Элементы искусственной нейронной системы	106
	Характеристики искусственной нейронной системы	108
	Новейшие достижения в технологии искусственных нейронных систем	110
	Приложения технологии искусственных нейронных систем	113
1.14	Коннекционистские экспертные системы и индуктивное обучение	114
1.15	Современное состояние разработок в области искусственного интеллекта	115
1.16	Резюме	122
	Задачи	124
Глава 2. Представление знаний		127
2.1	Введение	127
2.2	Смысл знаний	131
2.3	Продукции	139
2.4	Семантические сети	144
2.5	Тройки “объект–атрибут–значение”	149
2.6	Язык PROLOG и семантические сети	150
	Основные сведения о языке PROLOG	151
	Обеспечение поиска в языке PROLOG	153
2.7	Трудности, связанные с использованием семантических сетей	157
2.8	Схемы	159
2.9	Фреймы	162
2.10	Трудности, связанные с использованием фреймов	167
2.11	Логика и теория множеств	169
2.12	Пропозициональная логика	172
2.13	Логика предикатов первого порядка	181
2.14	Квантор всеобщности	181
2.15	Квантор существования	184
2.16	Кванторы и множества	185
2.17	Ограничения логики предикатов	187
2.18	Резюме	187
	Задачи	188
Глава 3. Методы логического вывода		193
3.1	Введение	193

3.2	Деревья, решетки и графы	194
3.3	Пространства состояний и пространства задач	199
	Примеры пространств состояний	199
	Пространства слабо структурированных задач	205
3.4	Деревья AND–OR и цели	206
3.5	Дедуктивная логика и силлогизмы	211
3.6	Правила вывода	220
3.7	Ограничения пропозициональной логики	231
3.8	Логика предикатов первого порядка	233
3.9	Логические системы	235
3.10	Резолюция	241
3.11	Системы резолюции и дедукция	245
3.12	Поверхностные и причинные рассуждения	248
3.13	Резолюция и логика предикатов первого порядка	254
	Преобразование в форму с логическими выражениями	255
	Унификация и правила	259
3.14	Прямой и обратный логический вывод	262
3.15	Другие методы логического вывода	271
	Аналогия	271
	Метод формирования и проверки	274
	Абдукция	275
	Немонотонный вывод	278
3.16	Метазнания	284
3.17	Скрытые марковские модели	285
3.18	Резюме	288
	Задачи	289
Глава 4. Рассуждения в условиях неопределенности		295
4.1	Введение	295
4.2	Неопределенность	296
4.3	Типы ошибок	301
4.4	Ошибки и индукция	304
4.5	Классическая вероятность	308
	Определение классической вероятности	308
	Выборочные пространства	312
	Теория вероятностей	314
4.6	Экспериментальные и субъективные вероятности	315
4.7	Сложные вероятности	317
4.8	Условные вероятности	321
	Мультипликативный закон	321
	Теорема Байеса	326

4.9 Гипотетические рассуждения и обратная индукция 328
4.10 Временные рассуждения и марковские цепи 335
4.11 Анализ вероятностных систем на основе понятий шансов и убеждений 341
4.12 Достаточность и необходимость 344
4.13 Применение неопределенности при формировании цепей логического вывода 347
Несовместимость коэффициентов, заданных экспертом 348
Неопределенное свидетельство 349
Исправление недостатков, связанных с неопределенностью 353
4.14 Комбинация свидетельств 354
Варианты классификации неопределенных свидетельств 354
Комбинирование свидетельств с использованием нечеткой логики 357
Логическая комбинация свидетельств 359
Эффективные значения правдоподобия 359
Сложности, связанные с использованием условной независимости 361
4.15 Сети логического вывода 362
Система PROSPECTOR 363
Сети логического вывода 365
Отношения логического вывода 368
Архитектура сети логического вывода 373
4.16 Распространение вероятностей 377
4.17 Резюме 381
Задачи 383

Глава 5. Нестрогие рассуждения 389
5.1 Введение 389
5.2 Неопределенность и правила 390
Источники неопределенности в правилах 390
Отсутствие надежных теоретических оснований 392
Взаимодействия правил 393
Разрешение конфликтов 394
Обобщение и неопределенность 397
5.3 Коэффициенты достоверности 399
Трудности, связанные с применением байесовского метода 400
Степени доверия и недоверия 401
Меры, применяемые для измерения степени доверия и недоверия 404

Вычисления, проводимые с использованием коэффициентов достоверности 407
Сложности, связанные с использованием коэффициентов достоверности 412
5.4 Теория Демпстера–Шефера 414
Рамки различения 414
Массовые функции и незнание 417
Комбинирование свидетельств 422
Нормализация степени доверия 428
Движущиеся массы и множества 430
Сложности, связанные с применением теории Демпстера–Шефера 431
5.5 Приближенные рассуждения 432
Нечеткие множества и естественный язык 434
Операции с нечеткими множествами 447
Нечеткие отношения 453
Лингвистические переменные 459
Принцип расширения 465
Нечеткая логика 466
Нечеткие правила 469
Композиция max-min 473
Метод максимума и метод моментов 477
Возможность и вероятность 482
Правила преобразования 485
Неопределенность в нечетких экспертных системах 489
5.6 Состояние неопределенности 492
5.7 Некоторые коммерческие приложения нечеткой логики 495
5.8 Резюме 497
Задачи 497

Глава 6. Проектирование экспертных систем 505
6.1 Введение 505
6.2 Выбор соответствующей задачи 506
Выбор наиболее приемлемого подхода 508
Выигрыш 509
Инструментальные средства 509
Стоимость 512
6.3 Общее описание процесса разработки экспертной системы 514
Руководство проектом 514
Проблема доставки 517
Сопровождение и развитие 519

6.4 Ошибки, возникающие на различных этапах разработки 521
6.5 Разработка программного обеспечения и экспертные системы 529
6.6 Жизненный цикл экспертной системы 533
Расходы на сопровождение 534
Модель каскадного развития жизненного цикла 535
Модель развития жизненного цикла на основе “кодирования и исправления” 537
Инкрементная модель жизненного цикла 537
Модель спирального развития жизненного цикла 538
6.7 Подробная модель жизненного цикла 538
Планирование 541
Определение знаний 542
Проектирование знаний 544
Разработка кода и отладка 546
Верификация знаний 546
Оценка системы 547
6.8 Резюме 548
Задачи 549

Глава 7. Введение в CLIPS 551
7.1 Введение 551
7.2 Язык CLIPS 552
7.3 Система обозначений 554
7.4 Поля 556
7.5 Вход и выход из системы CLIPS 559
7.6 Факты 561
Конструкция `deftemplate` 562
Многозначные слоты 563
Упорядоченные факты 563
7.7 Добавление и удаление фактов 564
7.8 Модификация и дублирование фактов 568
7.9 Команда `watch` 570
7.10 Конструкция `deffacts` 571
7.11 Компоненты правила 573
7.12 Рабочий список правил и выполнение программы 576
Отображение рабочего списка правил 577
Правила и релаксация правил 578
Отслеживание активизаций, правил и статистических данных 579
7.13 Команды, применяемые для манипулирования конструкциями 581
Отображение списка элементов указанной конструкции 581

Отображение текстового представления указанного элемента конструкции 582
Удаление указанного элемента конструкции 583
Удаление всех конструкций из среды CLIPS (очистка среды) 584
7.14 Команда `printout` 585
7.15 Использование нескольких правил 586
Отражение в правиле свойств реального мира 587
Правила с несколькими шаблонами 587
7.16 Команда `set-break` 589
7.17 Загрузка и сохранение конструкций 590
Загрузка конструкций из файла 590
Отслеживание операций компиляции 591
Сохранение конструкций в файле 592
7.18 Комментирование конструкций 592
7.19 Переменные 593
7.20 Многократное использование переменных 595
7.21 Адреса фактов 596
7.22 Однозначные подстановочные символы 599
7.23 Мир блоков 601
7.24 Многозначные подстановочные символы и переменные 608
Многозначные подстановочные символы 608
Многозначные переменные 609
Применение нескольких различных способов согласования с шаблонами 611
Реализация стека 612
Еще один вариант программы для мира блоков 613
7.25 Резюме 615
Задачи 615

Глава 8. Развитые средства сопоставления с шаблонами 625
8.1 Введение 625
8.2 Ограничения полей 626
Ограничение поля `not` 626
Ограничение поля `or` 627
Ограничение поля `and` 628
Совместное применение ограничений полей с другими конструкциями 628
8.3 Функции и выражения 630
Элементарные математические функции 630
Переменное количество параметров 633
Определение приоритета и уровня вложенности выражений 634

8.4 Суммирование значений с использованием правил 635
8.5 Функция bind 638
8.6 Функции ввода-вывода 639
Функция read 639
Функция open 640
Функция close 642
Чтение из файла и запись в файл 643
Функция format 644
Функция readline 646
8.7 Игра в палочки 648
8.8 Предикативные функции 651
8.9 Условный элемент test 652
8.10 Предикативное ограничение поля 654
8.11 Ограничение поля для возвращаемого значения 656
8.12 Программа Sticks 658
8.13 Условный элемент or 659
8.14 Условный элемент and 662
8.15 Условный элемент not 664
8.16 Условный элемент exists 667
8.17 Условный элемент forall 670
8.18 Условный элемент logical 673
8.19 Резюме 678
Задачи 679

Глава 9. Модульное проектирование, управление выполнением и эффективность правил 693
9.1 Введение 693
9.2 Атрибуты deftemplate 694
Атрибут type 694
Статическая и динамическая проверка ограничений 696
Атрибуты допустимого значения 698
Атрибут range 699
Атрибут cardinality 699
Атрибут default 700
Атрибут default-dynamic 702
Конфликтующие атрибуты слота 704
9.3 Значимость 704
9.4 Фазы и управляющие факты 708
9.5 Неправильное употребление подхода на основе значимости 714
9.6 Конструкция defmodule 717
Способы задания имен модулей в командах 719

9.7 Импорт и экспорт фактов 722
9.8 Модули и управление выполнением программы 726
Команда `focus` 728
Манипулирование стеком фокусов и его изучение 730
Команда `return` 732
Средство `auto-focus` 734
Отказ от фаз и управляющих фактов 735
9.9 Rete-алгоритм сопоставления с шаблонами 737
9.10 Сеть шаблонов 741
9.11 Сеть соединений 744
9.12 Важность правильного выбора порядка расположения шаблонов 748
Учет характеристик правила `match-1` 749
Учет характеристик правила `match-2` 750
Команда `matches` 752
Наблюдение за изменяющимся состоянием 754
9.13 Упорядочение шаблонов в целях повышения эффективности 755
Первоочередное размещение наиболее конкретных шаблонов 756
Размещение в последнюю очередь шаблонов, согласующихся с непостоянными фактами 756
Первоочередное размещение шаблонов, сопоставляющихся с наименьшим количеством фактов 757
9.14 Многозначные переменные и эффективность 757
9.15 Условный элемент `test` и эффективность программы 758
9.16 Встроенные ограничения сопоставления с шаблонами 760
9.17 Сравнение общих правил с конкретными правилами 761
9.18 Сравнение простых правил со сложными правилами 763
9.19 Резюме 766
Задачи 768

Глава 10. Процедурное программирование 779
10.1 Введение 779
10.2 Процедурные функции 779
Функция `if` 780
Функция `while` 781
Функция `switch` 782
Функция `loop-for-count` 784
Функция `progn$` 786
Функция `break` 787
Функция `halt` 787
10.3 Конструкция `deffunction` 788
Функция `return` 790

Еще один вариант программы Sticks 792
Рекурсия 794
Предварительные объявления 794
Отслеживание работы конструкций `deffunction` 796
Параметр с подстановочным символом 798
Команды для работы с конструкцией `deffunction` 799
Функции, определяемые пользователем 801
10.4 Конструкция `defglobal` 802
Команды, применяемые наряду с конструкциями `defglobal` 804
Принципы сброса (переустановки) значения переменной `defglobal` 805
Отслеживание значений переменных `defglobal` 806
Переменные `defglobal` и сопоставление с шаблонами 807
Использование конструкций `defglobal` 808
10.5 Конструкции `defgeneric` и `defmethod` 811
Еще один вариант конструкции `deffunction` с именем `check-input` 813
Приоритеты методов 817
Ограничения запроса 824
Отслеживание работы универсальных функций и методов 828
Команды `defmethod` 829
Команды `defgeneric` 830
Перегруженные функции и команды 831
10.6 Процедурные конструкции и конструкции `defmodule` 832
10.7 Полезные команды и функции 835
Загрузка и сохранение фактов 835
Команда `system` 836
Команда `batch` 837
Команды `dribble-on` и `dribble-off` 839
Выработка случайных чисел 839
Преобразование строки в поле 840
Поиск символа 841
Сортировка списка полей 841
10.8 Резюме 843
Задачи 844

Глава 11. Классы, экземпляры и обработчики сообщений 851
11.1 Введение 851
11.2 Конструкция `defclass` 852
11.3 Создание экземпляров 853
11.4 Обработчики сообщений, определяемые системой 854

11.5 Конструкция definstances 857
11.6 Классы и наследование 858
Разрешение конфликтов между определениями слотов 862
Абстрактные и конкретные классы 865
Команды, относящиеся к конструкции defclass 866
11.7 Сопоставление с шаблоном объекта 869
Сопоставление с шаблоном объекта и наследование 871
Ключевые слова is-a и name 872
Активизация шаблонов объектов 875
Атрибут сопоставления с шаблонами 877
Шаблоны объектов и условный элемент logical 878
Шаблон initial-object 881
11.8 Обработчики сообщений, определяемые пользователем 882
Сокращенные ссылки на слоты 884
Отслеживание процесса передачи сообщений и функционирования обработчиков сообщений 887
Команды defmessage-handler 889
11.9 Доступ к слоту и создание обработчика 890
11.10 Обработчики сообщений before, after и around 894
Обработчики before и after 896
Обработчики around 899
Перекрытие параметров обработчика сообщений 903
Порядок вызова обработчиков на выполнение 906
11.11 Создание экземпляра, инициализация и удаление обработчиков сообщений 915
Атрибут storage 917
11.12 Модификация и дублирование экземпляров 919
11.13 Классы и универсальные функции 924
11.14 Функции запроса множества экземпляров 925
Определение того, успешно ли выполнен запрос 926
Определение экземпляров, удовлетворяющих запросу 928
Выполнение действий над экземплярами, удовлетворяющими запросу 929
11.15 Множественное наследование 931
Конфликты, связанные с множественным наследованием 933
Сохранение и восстановление значений слотов 935
11.16 Конструкции defclass и defmodule 939
11.17 Загрузка и сохранение экземпляров 942
11.18 Резюме 944
Задачи 945

Глава 12. Примеры проектов экспертных систем 949
12.1 Введение 949
12.2 Коэффициенты достоверности 949
12.3 Деревья решений 955
Деревья решений с несколькими ребрами 957
Деревья решений, позволяющие проводить обучение 959
Программа для работы с деревом решений, основанная на правилах 961
Пошаговая трассировка программы обучения дерева решений 967
12.4 Обратный логический вывод 972
Алгоритм работы программы обратного логического вывода 972
Представление правил обратного логического вывода в языке CLIPS 975
Машина обратного логического вывода на языке CLIPS 978
Поэтапная трассировка работы программы обратного логического вывода 980
12.5 Задача текущего контроля 986
Формулировка задачи 987
Уточнение деталей, с которого должна начинаться разработка 990
Определения структур представления знаний 992
Управление работой программы 995
Чтение бесформатных показаний датчиков 995
Распознавание тенденций 1004
Формирование предупреждающих сообщений 1008
12.6 Резюме 1011
Задачи 1012

Приложение А. Некоторые широко применяемые эквивалентности 1015

Приложение Б. Некоторые элементарные кванторы и их значение 1017

Приложение В. Некоторые свойства множеств 1019

Приложение Г. Информация о поддержке CLIPS 1021

Приложение Д. Общие сведения о командах и функциях CLIPS 1023

Приложение Е. Определение языка в нормальной форме Бэкуса–Наура 1059

Приложение Ж. Программные ресурсы 1069

Приложение З. Литература 1119

Предметный указатель 1125

Предисловие

Рекомендации по эффективному использованию книги

Данное четвертое издание представляет собой результат существенного пересмотра известного во всем мире учебника по экспертным системам и разработке программного обеспечения с помощью инструментария языка экспертных систем CLIPS. Коммерческое внедрение экспертных систем началось в 1980-х годах; с тех пор непрерывно происходит их беспрецедентное развитие и распространение. В настоящее время экспертные системы широко применяются в бизнесе, науке, технике, сельском хозяйстве, на производстве, в медицине, в видеоиграх, а также практически в любом направлении человеческой деятельности. В действительности трудно представить себе такую область, в которой бы в наши дни не использовались экспертные системы.

Настоящая книга позволяет усвоить значительный объем знаний в области теории и программирования экспертных систем. Теоретический материал изложен на уровне, доступном для восприятия студентов старших курсов и аспирантов, интересующихся экспертными системами, которые специализируются в области компьютерных наук, информационных управленческих систем, в программотехнике и других областях. Новые термины обозначаются полужирным шрифтом и сопровождаются пояснениями. Для поиска пояснений к используемым терминам можно воспользоваться предметным указателем. В книге приведены многочисленные примеры и ссылки, позволяющие уточнить смысл излагаемого материала и получить указания по выбору дополнительной литературы. Кроме того, в новом, 4-м издании в приложении Ж приведены ссылки на многие новые бесплатные и испытательные версии программных инструментальных средств, которые могут служить основой для выполнения дополнительных упражнений, а также указано, где можно найти вспомогательные учебные материалы.

Описание каждой новой темы, как правило, проводится в историческом контексте, что позволяет не просто пользоваться готовыми результатами, но и понять, какова была цель разработки тех или иных методов. Тем самым авторы

стремились поставить в основу обучения подход, позволяющий понять, чем была вызвана необходимость в разработке новых способов решения задач, а не такой подход, при котором обучение напоминает преподавание на курсах, где просто показывают, как использовать то или иное приложение.

Книга *Экспертные системы: принципы разработки и программирование* включает сведения, относящиеся к двум основным направлениям, — теоретический материал излагается в главах 1–6, а сведения по программированию с использованием инструментария языка экспертных систем CLIPS приведены в главах 7–12. Таким образом, в первой половине книги излагается теория экспертных систем и показано, какое место занимают экспертные системы во всем объеме компьютерных наук.

Безусловно, предварительное знакомство с областью искусственного интеллекта может оказаться полезным, но настоящая книга содержит отдельное введение в тематику искусственного интеллекта, приведенное в главе 1, объем которого достаточен для изучения экспертных систем. Разумеется, в одной главе невозможно изложить все, чему посвящены целые книги по искусственному интеллекту. Но данная глава поможет получить общее представление и об искусственном интеллекте, и о том, для чего предназначены экспертные системы. В целом в первых шести главах книги рассматриваются логика, теория вероятностей, структуры данных, основные понятия искусственного интеллекта и другие темы, которые лежат в основе теории экспертных систем.

Авторы стремились изложить теорию, на которой базируются экспертные системы, чтобы дать возможность читателю принять обоснованное решение по выбору подходящего направления использования технологии экспертных систем. Мы хотели подчеркнуть важную мысль, что экспертные системы следует рассматривать как одно из инструментальных средств в арсенале разработчика, и поэтому постараться понять, в чем состоят их преимущества и недостатки. В излагаемых теоретических сведениях дано также описание того, как экспертные системы связаны с другими методами программирования, такими как процедурное программирование. Еще одной причиной повышенного внимания к изложению теоретических сведений было стремление подготовить читателя к тому, чтобы он мог самостоятельно изучать современные научно-исследовательские статьи по экспертным системам. Но следует отметить, что экспертные системы создаются в соответствии с потребностями многих разных теоретических и практических направлений, поэтому на первых порах при чтении таких статей для достижения полного понимания придется преодолевать трудности.

Вторая половина данной книги представляет собой вводное описание инструментария языка экспертных систем CLIPS. В этих главах дано практическое введение в программирование экспертных систем, позволяющее объяснить и закрепить теоретические знания, изложенные в первых главах книги. Но так же, как не лишней окажется предварительная подготовка при изучении теоретических сведений,

изложенных в данной книге, сведения о программировании будут более доступными для читателей, имеющих некоторый опыт работы на языке высокого уровня. Процесс разработки экспертных систем, предназначенных для решения практических задач, демонстрируется с использованием языка CLIPS — современного мощного инструментального средства создания экспертных систем.

Еще одним новым средством, описанным в данном издании, является язык COOL (CLIPS Object-Oriented Language — объектно-ориентированный язык CLIPS). Язык COOL позволяет разрабатывать экспертные системы исключительно с использованием объектов или осуществлять гибридный подход на основе применения и правил, и объектов. Преимуществом объектно-ориентированного подхода является то, что объекты позволяют более легко группировать множества структур представления знаний в крупные коллекции, чем при использовании отдельных правил. Язык COOL предоставляет все общепринятые возможности работы с объектами (в том числе множественное наследование), что позволяет легко дополнять объекты более специализированными структурами представления знаний, а не “изобретать колесо” и каждый раз разрабатывать код с нуля, как при использовании систем, основанных исключительно на правилах. Кроме того, в настоящем издании обсуждаются средства процедурного программирования языка CLIPS, включая глобальные переменные, функции, а также универсальные функции.

Первая версия языка CLIPS была разработана NASA в Космическом центре Джонсона. В ходе этой разработки один из авторов данной книги, Гэри Райли (Gary Riley), выполнял функции ведущего программиста по разработке компонентов системы CLIPS, основанных на правилах, а другой автор, Джозеф К. Джарратано (Joseph C. Giarratano), работал в качестве консультанта и подготавливал официальные версии руководств пользователя языка CLIPS для NASA. В настоящее время язык CLIPS используется для реализации важных проектов в сфере государственного управления, в бизнесе, промышленности и буквально повсеместно. Поиск сведений об экспертных системах, написанных на языке CLIPS, проведенный с помощью любой машины поиска в Интернете, приводит к обнаружению сотен и даже тысяч ссылок. Кроме того, этот поиск позволяет обнаружить, что язык CLIPS входит в программу обучения многих университетов во всем мире.

Исходный код программного обеспечения языка CLIPS является переносимым, поэтому система CLIPS может эксплуатироваться практически на любом компьютере и под управлением любой операционной системы, которая поддерживает компилятор ANSI C или C++. К данной книге прилагается компакт-диск, содержащий исполняемые файлы CLIPS для Windows и MacOS, руководства *CLIPS Reference Manual* и *CLIPS Users Guide*, а также хорошо документированный полный исходный код C для CLIPS.

В процессе проведения некоторых курсов по экспертным системам осуществляется разработка проектов с заданными сроками. Участие в проекте — превосходный способ приобретения навыков создания экспертных систем. Студенты обычно успевают закончить разработку небольших экспертных систем с количеством правил от 50 до 150, участвуя по своему выбору в одном из проектов, рассчитанных на семестр. На основе настоящей книги созданы тысячи проектов и организованы сотни курсов по многим направлениям, включая медицинскую диагностику, диагностику неисправностей в автомобилях, планирование работы такси, планирование работы персонала, управление компьютерными сетями, прогнозирование погоды, предсказание ситуаций на фондовой бирже, консультирование покупателей потребительских товаров и многие другие направления. Проведя поиск в Интернете с использованием машины поиска, читатель сможет получить сведения о многочисленных курсах и воспользоваться такими информационными ресурсами, как слайды в формате PowerPoint, программы курсов лекций и учебные задания, разработанные университетами во всем мире.

Ниже приведен рекомендуемый план проведения курсов продолжительностью один семестр.

1. Пройти материал главы 1 для ознакомления с кратким введением в тематику экспертных систем. В частности, дать задание студентам решить задачи 1.1, 1.2 и 1.3.
2. Пройти материал глав 7–10 для ознакомления с основными сведениями по программированию на языке CLIPS. Для студентов полезно было бы повторно составить программу решения задачи 1.2 из главы 1, чтобы сравнить подход, основанный на языке экспертных систем, с тем подходом, который предусматривает использование языка, первоначально рассматриваемого в главе 1. Такое сравнение является очень полезным, поскольку позволяет проще понять различия между таким языком, основанном на правилах, как CLIPS (или языком искусственного интеллекта наподобие LISP или PROLOG), и языком, который первоначально использовался при решении задачи 1.2. Еще один вариант состоит в том, что после изучения материала главы 10 преподаватель может вернуться к теоретическим главам книги. Если студенты имеют хорошую подготовку в логике и языке PROLOG, то значительную часть глав 2 и 3 можно пропустить. Если требуется сделать особый упор на логику и фундаментальную теорию экспертных систем, то студентам пойдет на пользу изучение глав 2 и 3, независимо от того, прошли они вступительный курс по искусственному интеллекту на основе языка LISP или нет. Если студенты имеют качественную подготовку в теории вероятностей и математической статистике, то материал главы вплоть до раздела 4.11 можно пропустить.

3. В главах 4 и 5 обсуждаются темы, касающиеся учета неопределенности. Это очень важно, поскольку людям постоянно приходится решать задачи, связанные с неопределенностью, и без этого экспертные системы стали бы не более значимыми по сравнению с простыми деревьями решений. К основным темам при изучению проблематики неопределенности относятся вероятностный и байесовский логический вывод, коэффициенты достоверности, теория Демпстера–Шефера и теория нечетких множеств. Студенты приобретают достаточно широкие знания, относящиеся к этим методам, чтобы иметь возможность читать современные статьи в этой области и приступить к выполнению исследований, если есть такая необходимость.
4. В главе 6 рассматриваются темы приобретения знаний и программотехники экспертных систем; это означает, что для студентов открывается перспектива работы над крупными экспертными системами. С другой стороны, приступать к изучению этой главы, прежде чем студентам будет поручен проект с заданными сроками, нет необходимости. В действительности было бы лучше пройти данную главу в последнюю очередь, чтобы студенты могли сами оценить значение всех факторов, от которых зависит создание качественной экспертной системы.

Дополнительные ресурсы

Дополнительные материалы к данной книге представлены на сопровождающем ее Web-узле. Для того чтобы ознакомиться с тем, что входит в состав этих материалов и каковы условия их получения, перейдите по адресу `http://www.course.com` и проведите поиск, указав ISBN англоязычного издания настоящей книги (`0534384471`) в поле `Find:`. Кроме того, в связи с изложением многих тем в самой книге приведены многочисленные ссылки на программное обеспечение и другие ресурсы, доступ к которым можно получить через Web. Эти ресурсы были выбраны так, чтобы студенты могли лучше усвоить на практике такие темы, как логика и теория вероятностей, используя программное обеспечение для проведения экспериментов с нетривиальными задачами, а не решая задачи только на бумаге. Было также включено большое количество ресурсов, относящихся к искусственному интеллекту, логике, теории вероятностей, байесовскому логическому выводу, нечеткой логике и другим темам, чтобы студенты смогли получить более широкие знания в области искусственного интеллекта и ознакомиться с деятельностью сообщества пользователей экспертных систем во всем мире.

Участники разработки языка CLIPS

Мы хотели бы поблагодарить всех тех, кто внес свой вклад в успешное завершение проекта по созданию языка CLIPS. Как и в случае с любым крупным проектом, CLIPS — результат усилий многих людей. Основными участниками разработки были Роберт Сэйвли (Robert Savely), руководитель исследовательских работ по современным программным технологиям компании JSC, который сформулировал замысел проекта и обеспечил общее руководство и поддержку; Крис Калберт (Chris Culbert), руководитель отделения программной технологии, который руководил проектом и написал первоначальную версию справочного руководства *CLIPS Reference Manual*; Гари Райли (Gary Riley), который проектировал и разработал часть CLIPS, основанную на правилах, является одним из авторов руководств *CLIPS Reference Manual* и *CLIPS Architecture Manual*, разработал интерфейс Macintosh для CLIPS, а в настоящее время сопровождает систему CLIPS и ведет официальный Web-узел CLIPS (`http://www.ghg.net/clips/CLIPS.html`); Брайен Доннелл (Brian Donnell), который разработал язык COOL (CLIPS Object Oriented Language — объектно-ориентированный язык CLIPS) и был одним из авторов руководств *CLIPS Reference Manual* и *CLIPS Architecture Manual*; Биби Лай (Bebe Ly), который разработал интерфейс X Window для CLIPS; Крис Ортис (Chris Ortiz), который разработал интерфейс Windows 3.1 для CLIPS; доктор Джозеф Джарратано (Dr. Joseph Giarratano) из Университета Хьюстон Клиэр Лэйк, который подготовил официальную версию руководства *CLIPS User Guide* для NASA, прилагаемую к каждому выпуску CLIPS в агентстве NASA; и особенно Фрэнк Лопес (Frank Lopez), который написал первоначальную версию прототипа CLIPS.

Благодарности

Во время написания настоящей книги авторы имели возможность руководствоваться весьма полезными комментариями многих специалистов, включая следующих: Тэд Лейбфрид (Ted Leibfried), Джин Лесли (Jeanne Leslie), Мак Амфри (Mac Umphrey), Тэрри Фиджин (Terry Feagin), Деннис Мэрфи (Dennis Murphy), Дженна Джарратано (Jenna Giarratano) и Мелисса Джарратано (Melissa Giarratano). Мы хотели бы также поблагодарить рецензентов четвертого издания за сделанный ими важный вклад: Чиен-Чань Чена (Chien-Chung Chan), Университет Экрон; Константина Вассилиадиса (Constantine Vassiliadis), Университет штата Огайо; Дженни Скотт (Jenny Scott), Университет Конкордия, Канада; и Энтони Зигмонта (Anthony Zygmont), Университет Вилланова.

Мы хотели бы также выразить свою благодарность многим людям, которые вносили различные усовершенствования в систему CLIPS в течение всех двадцати лет, начиная с первого выпуска этой системы в 1985 году. Благодаря тому что полный исходный код системы CLIPS был в свое время предоставлен без ограничений во всеобщее пользование, у сообщества программистов, работающих в рамках движения за создание программного обеспечения с открытым исходным кодом, появилась возможность внести свой вклад в доработку этой системы. В результате мощь и популярность CLIPS возросли до такой степени, о которой мы не смели даже мечтать в 1985 году, впервые приступая к разработке CLIPS. В то время экспертные системы все еще представляли собой новую и непроверенную технологию, и никто не знал, выдержат ли они проверку временем. А за последние двадцать лет система CLIPS, некогда бывшая скромным начинанием в NASA, выросла до такой степени, что ее использует всемирное сообщество специалистов, состоящее из тысяч людей, которые доказывают преимущества CLIPS практически в любой области. Мы хотим особо поблагодарить всех разработчиков, которые улучшили функциональные характеристики и расширили возможности системы CLIPS, благодаря чему некогда небольшой и рискованный проект, предназначенный лишь для элементарной проверки технологии искусственного интеллекта в NASA, превратился в мировой феномен.

В числе тех людей, которые внесли особый вклад в распространение экспертных систем, можно смело назвать Эрнеста Фридмана-Хилла (Ernest Friedman-Hill), который, действуя независимо от других разработчиков, написал версию системы CLIPS на языке Java, получившую название JESS и обладающую новыми характеристиками. Кроме того, Эрнест написал книгу по языку JESS (*Jess in Action: Rule-Based Systems in Java*) с многочисленными интересными проектами.

- JESS. Дополнение к языку CLIPS (`http://herzberg.ca.sandia.gov/jess/`).
- KAPICLIPS 1.0 (`http://www.cs.umbc.edu/kqml/software/kapiclips.shtml`).

Другие системы, производные от CLIPS

Перечень других систем, которые происходят от CLIPS, приведен ниже.

- PerlCLIPS (`http://www.discomsys.com/~mps/dnld/clips-stuff/`).
- Protege. Редактор онтологий и баз знаний для CLIPS (`http://protege.stanford.edu/index.html`).
- Интерфейс Python–CLIPS (`http://www.yodanet.com/portal/Products/download/clips-python.tar.gz/view`).
- TixClips. Интегрированная среда разработки экспертной системы CLIPS (`http://starship.python.net/crew/mike/TixClips/`), в которой используется интерфейс Tix (`http://tix.sourceford.net/`).
- TclClips. Расширение Tcl (`www.eolas.net/tcl/clips`), созданное с помощью оболочки SWIG (`http://www.swig.org/`).
- WebCLIPS. Реализация системы CLIPS как приложения CGI (`http://www.monmouth.com/~km2580/wchome.htm`).
- wxCLIPS. Среда разработки приложений систем баз знаний с графическими интерфейсами пользователя (`http://www.anthemion.co.uk/wxclips/wxclips2.htm`).
- ZClips 0.1. Пакет, обеспечивающий взаимодействие среды Zope с системой CLIPS (`http://www.zope.org/Members/raystream/zZCLIPS0.1`).
- Система CLIPS/R2 компании Production Systems Technologies (`http://www.pst.com/clips_r2.htm`).
- National Research Council of Canada. Другие версии CLIPS (такие как FuzzyClips), предоставляемые Национальным научно-исследовательским советом Канады (`http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyClips/fuzzyCLIPSIndex.html`).
- Система FuzzyClips компании Togai InfraLogic, Inc. (`http://www.ortech-engr.com/fuzzy/fzyclips.html`).
- AdaCLIPS (`http://www.telepath.com/~dennison/Ted/AdaClips/AdaClips.html`).
- CLIPS и Perl с расширениями (`http://cape.sourceforge.net/`).
- Многие другие версии инструментальных средств на основе CLIPS предоставлены по адресу (`http://www.ghg.net/clips/OtherWeb.html`).

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать любые ваши замечания, касающиеся книги.

Мы ждем ваших комментариев. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail:	`info@williamspublishing.com`
WWW:	`http://www.williamspublishing.com`

Адреса для писем:

из России:	115419, Москва, а/я 783
из Украины:	03150, Киев, а/я 152

Глава 1

Введение в экспертные системы

1.1 Введение

Настоящая глава представляет собой общее введение в тематику экспертных систем. В ней рассматриваются фундаментальные принципы функционирования экспертных систем, обсуждаются преимущества и недостатки экспертных систем, а также описаны наиболее подходящие области их применения. Кроме того, обсуждается связь экспертных систем с другими методами программирования.

1.2 Определение понятия экспертной системы

В XX веке было сформулировано понятие **искусственного интеллекта** (ИИ) и предложен ряд определений этого понятия. Одним из первых определений, получивших широкое признание и до сих пор остающимся популярным, является следующее: “Способ заставить компьютеры думать как люди”. Распространенность этого определения подтверждается тем, что подобные взгляды пропагандируются во многих научно-фантастических фильмах. Но фактически истоком этого определения явился знаменитый тест Тьюринга, предложенный британским математиком и одним из первых исследователей в области компьютерных наук Аланом Тьюрингом (Alan Turing). В данном тесте экспериментатор пытается определить, является ли человеком или компьютерной программой то “лицо”, с которым он обменивается сообщениями с помощью дистанционной клавиатуры. Объект, успешно прошедший такой тест, считается обладающим **сильным искусственным интеллектом**. Термин *сильный искусственный интеллект* пропагандируется специалистами, которые считают, что искусственный интеллект должен базироваться на строгом логическом основании, в отличие от подхода, называемого ими **слабым искусственным интеллектом**, базирующимся на искус-

ственных нейронных сетях, генетических алгоритмах и эволюционных методах. В наши дни стало очевидно, что ни один из методов искусственного интеллекта не позволяет успешно справиться со всеми проблемами; лучше всего проявляет себя комбинация методов.

Первая программа, прошедшая тест Тьюринга, была написана в ходе проведения психологических экспериментов Стивеном Вейценбаумом (Steven Weizenbaum) в 1967 году. С тех пор уровень знаний в этой области значительно возрос, а способы взаимодействия с экспериментаторами стали гораздо более совершенными, поэтому проводятся отдельные соревнования с призовым фондом в 100 тысяч долларов США, называемые соревнованиями за **приз Лебнера**, в ходе которых определяется лучшая программа (`http://www.loebner.net/Prizef/loebner-prize.html`). Безусловно, в наши дни для общения с компьютером чаще всего применяются средства распознавания и синтеза речи, а не старомодный телетайп или клавиатура. Поэтому, если вас когда-либо поставит в тупик такая ситуация, что вы будете считать себя разговаривающим по телефону с человеком, а он не будет понимать, что́ вы ему говорите, то задайте вопрос, прошел ли он тест Тьюринга.

Экспертные системы были разработаны как научно-исследовательские инструментальные средства в 1960-х годах и рассматривались в качестве искусственного интеллекта специального типа, предназначенного для успешного решения сложных задач в узкой предметной области, такой как медицинская диагностика заболеваний. Классическая задача создания программы искусственного интеллекта общего назначения, которая была бы способна решить любую проблему без конкретных знаний в предметной области (например, медицинской диагностики заболеваний), оказалась слишком сложной. Коммерческое внедрение экспертных систем произошло в начале 1980-х годов, и с того времени экспертные системы получили очень широкое распространение. В настоящее время экспертные системы используются в бизнесе, науке, технике, на производстве, а также во многих других сферах, где существует вполне определенная предметная область. В действительности дело обстоит так, что если предприятие проходит аудит в налоговом управлении или частное лицо подает заявку на получение кредитной карточки, то окончательное решение чаще всего принимает экспертная система.

Выражение “вполне определенный”, встретившееся в предыдущем абзаце, представляет собой ключевое слово и обсуждается более подробно ниже в данной главе. Основной смысл этого выражения состоит в том, что эксперт способен определить этапы рассуждений, с помощью которых может быть решена любая задача из данной предметной области, а это означает, что аналогичные действия могут быть выполнены и с использованием экспертной системы. Ведь если какое-то лицо не способно объяснить ход своих рассуждений, то ему лучше попытать удачу не в научной лаборатории, а в Лас-Вегасе.

В качестве контрпримера можно указать, что многие специалисты пытались создавать экспертные системы для предсказания тенденций на фондовой бирже; фактически подобные системы постоянно применяются на Уолл-стрит. Однако, если отслеживаются абсолютно все взлеты и падения на Уолл-стрит, то не обнаруживаются какие-либо очевидные тенденции и экспертные системы действуют не лучше по сравнению с их создателями. Значительным преимуществом экспертных систем является то, что они способны осуществлять торговлю в реальном времени в таких условиях, что критическими становятся задержки при совершении биржевых сделок на миллисекунды, поскольку система конкурента может обнаружить такую же тенденцию, как и ваша, и быстрее выдать ордера на покупку или продажу акций стоимостью в сотни миллионов долларов, притом что быстродействие обеих систем будет намного выше по сравнению с реакцией человека. Как можно оценить результаты применения экспертных систем в этой области? Печально знаменитый крах фондового рынка, наступивший в 1987 году, стал причиной введения многих новых ограничений в биржевой торговле, позволяющих исключить такие случаи, когда компьютеры продавали акции на сотни миллионов, чтобы получить прибыль в несколько сотен долларов и создать тем самым предпосылки возникновения краха.

Экспертные системы представляют собой очень успешное приложение технологии искусственного интеллекта. Предложено также много гибридных подходов, позволяющих применять методы экспертных систем в сочетании с другими методами, такими как генетические алгоритмы и искусственные нейронные сети. Для обозначения систем, в которых используется искусственный интеллект, сложился общий термин — **интеллектуальная система**, или *автоматизированная система* [47].

Вообще говоря, первым шагом в решении любой задачи является определение проблемной области, или **предметной области**, в рамках которой необходимо найти решение. Такой подход относится не только к искусственному интеллекту, но и к обычному программированию. Но в широких кругах пользователей искусственный интеллект рассматривается как окруженный какой-то мистической завесой, поэтому многие продолжают еще верить устаревшей формулировке: “Если эта задача еще не решена, то она относится к области искусственного интеллекта”. Кроме того, большинство людей считают истинным еще одно определение: “Благодаря применению искусственного интеллекта компьютеры приобретают способность действовать так же, как они действуют в фильмах”. Подобные взгляды нашли особенно широкое распространение в 1970-х годах, когда искусственный интеллект находился полностью на стадии исследований. В настоящее время с помощью искусственного интеллекта решено много практических задач и создано много коммерческих приложений. Эта тематика обсуждается в оперативных журналах, таких как *PCAI.com*, на конференциях ти-

па *AAAI* (`http://aaii.org/conferences/conferences.htm`) и в книгах [69]. Дополнительные сведения на эту тему приведены в приложении Ж.

Прежде чем перейти к более подробному обсуждению искусственного интеллекта, необходимо сделать небольшое отступление и рассмотреть общую картину того, как искусственный интеллект вписывается в саму проблематику жизни и деятельности человечества. Такой подход ведет к формулировке первого вопроса: "Что такое жизнь?" Как описано Адами в [1], предложено много определений понятия *жизнь*, в том числе физиологические, метаболические, биохимические, генетические и термодинамические. Выбор исследователем за основу того или иного определения зависит от того, каких взглядов он придерживается. А какое определение является правильным с точки зрения рассматриваемой тематики? Это полностью зависит от того, какой аспект жизни рассматривается в данной предметной области. Но, по-видимому, самым простым является определение, сформулированное Шекспиром: "Жизнь — это история, рассказанная идиотом, наполненная шумом и яростью и не значащая ничего" ("Макбет", акт V, сцена 5).

С точки зрения компьютерных наук жизнь может быть представлена как программное обеспечение. И действительно, к книге Адами прилагается компакт-диск с программным обеспечением, которое позволяет пользователю создавать искусственные формы жизни и экспериментировать с ними. Имеется также метафизическое определение жизни, столь талантливо описанное в фильмах, подобных "Матрице" (*The Matrix*), в которых люди "живут" внутри гигантской компьютерной программы. Другие аспекты создания искусственной жизни в компьютере рассматриваются в книге Хелмрейха [46], который обсуждает более подробно философские и даже духовные аспекты компьютеризированной искусственной жизни.

А что касается биологической точки зрения, то в поисках способов создания искусственных форм жизни можно больше не ограничиваться компьютерными системами. Начиная с 1990-х годов появилась возможность клонировать млекопитающих, как овечку Долли, и компании теперь продают сельскохозяйственным предприятиям клонированных коров, а безутешным владельцам погибших домашних животных — клонированные копии их любимцев, таких как кошки. Но метод клонирования как создания точной копии живого существа (т.е. искусственной жизни) является только первым шагом в "усовершенствовании" жизни. Например, одна группа ученых создала кролика с дополнительным геном, взятым от светлячков, чтобы он светился в темноте. Такие формы искусственной жизни не имеют ни одного естественного предка, от которого они бы происходили, поэтому в полном смысле слова представляют собой искусственную жизнь. Вместе с тем такие создания являются интеллектуальными, поэтому могут также рассматриваться как обладающие искусственным интеллектом, хотя не таким, который принято считать создаваемым с использованием компьютеров.

На основе понятия искусственной жизни была создана новая область творческих эволюционных систем, которые позволяют системам с искусственной жизнью корректировать свои собственные программы в ответ на эволюционное давление, как показано в книге Бентли [4]. В литературе описано много разных методов (таких как генетические алгоритмы), имеющих практическое приложение в музыке, искусстве, проектировании интегральных микросхем, архитектуре и управлении самолетом-истребителем. К книге Бентли также прилагается компакт-диск, позволяющий пользователю проводить эксперименты с творческими эволюционными системами. Следует отметить, что указанная книга посвящена изучению компьютерного представления этих систем, а не новых биологических форм жизни, которые в будущем должны специально выращиваться в целях реализации замыслов проектировщика, например, светящихся кроликов (уже созданы), чтобы дети чувствовали себя спокойно в темноте, или обезьян, являющихся лучшими пилотами по сравнению с любым человеком.

В книге де Силва [20] приведено еще одно определение интеллекта, которое выглядит таким образом: “Интеллект — это способность учиться, способность приобретать, адаптировать, модифицировать и пополнять знания в целях решения задач”. В данном случае обнаруживается стремление к созданию интеллектуальных машин, которые взаимодействуют с реальным миром с помощью робототехники, производственного оборудования, приборов и других аппаратных средств. Для этого требуется реализовать в машинах сложные механизмы, помогающие человеку успешно действовать, несмотря на такие особенности реального мира, как двусмысленность, расплывчатость, общность, неточность, неопределенность, нечеткость, недостоверность и необходимость учитывать ограниченную степень правдоподобия. Многие из этих тем рассматриваются ниже в данной книге, в главах 4 и 5, посвященных проблемам неопределенности. Обратите внимание на то, что предыдущее предложение само является двусмысленным. Относится ли выражение “в данной книге” к книге де Силва или к книге, оригинал которой называется *Expert Systems Principles and Programming*? Живые существа способны успешно отвечать на вопросы, подобные этому, но роботы и компьютеры сталкиваются с затруднениями, обнаруживая аналогичные неопределенные ситуации, если в них используется только классическая логика.

Еще более интересные задачи возникают при разработке искусственных интеллектуальных систем, которые являются также обладающими сознанием. Безусловно, в настоящее время о работе мозга известно уже очень многое [18], но мы еще не знаем, где находится экран, на котором отображается сознание, или благодаря чему мы становимся теми людьми, какие мы есть. Однако новые инструментальные средства, такие как создание изображений с помощью функционального магнитного резонанса (functional Magnetic Resonance Imaging — fMRI), позволяют наблюдать в динамике активизацию различных структур мозга, контролируя связанные с этим изменения в атомах металлов. Безусловно, если бы мы

использовали модель искусственно клонированного животного, то уже добились бы успеха, поскольку нет сомнений в том, что овцы и кошки обладают сознанием. Но до сих пор неизвестно, как наделить сознанием машину. Еще более важно то, что после просмотра фильмов "Терминатор" (*Terminator*) или "Матрица" (*Matrix*) можно понять, что, по-видимому, и не нужно, чтобы интеллектуальная машина обладала сознанием. В конце концов, ведь никто не хочет, чтобы его отключили от разъема питания, ни люди, ни компьютеры.

Идеальные решения таких классических задач искусственного интеллекта, как перевод текста на естественном языке, понимание речи и создание системы машинного зрения, еще не найдены, но достаточно полезные решения могут быть предложены благодаря ограничению предметной области. Например, в наши дни несложно создать простую систему с поддержкой естественного языка, если ввод ограничивается предложениями в форме "подлежащее, сказуемое и дополнение". В настоящее время системы такого типа позволяют очень успешно предоставлять дружественные интерфейсы ко многим программным продуктам, таким как системы баз данных и электронные таблицы. Кроме того, теперь имеются системы распознавания речи, не зависящие от диктора, которые обладают высокой точностью и не требуют обучения голосу конкретного пользователя, как при эксплуатации первых таких систем. Подобные интеллектуальные системы в сочетании с экспертными системами в конечном итоге заменят многие телефонные центры обработки заказов, которые принимают заявки от заказчиков; этот период наступит после того, как указанные системы пройдут тест Тьюринга [69].

Разработан целый ряд коммерческих версий систем распознавания речи, которые работают под управлением стандартного программного обеспечения персонального компьютера и могут быть приобретены по весьма умеренным ценам. Кроме того, системы распознавания речи широко используются для бесконтактного управления сотовыми телефонами в автомобилях и обладают превосходными характеристиками распознавания, если предметная область ограничивается вводом только цифр, а не всех возможных слов. Синтаксические анализаторы, применяемые в наше время в составе программного обеспечения популярных компьютерных текстовых приключенческих игр, обнаруживают потрясающие способности к пониманию естественного языка, что необходимо для ведения таких игр с несколькими игроками в локальной сети, в которых использование ввода данных с клавиатуры привело бы к замедлению игры.

Экспертные системы применяются в сочетании с базами данных для обеспечения распознавания образов по такому же принципу, как это делает человек, и с автоматизированными системами принятия решений для обеспечения выявления знаний с помощью **анализа скрытых закономерностей в данных** и создания таким образом **интеллектуальной базы данных** [9]. Одной из важных областей использования систем распознавания образов являются системы безопасности аэропортов, в которых применяются средства распознавания лиц потенциальных

подозреваемых в качестве внешнего интерфейса к экспертной системе. В свою очередь, экспертная система, получив сведения об обнаружении потенциального подозреваемого, определяет, оправдан ли переход к таким дальнейшим действиям, как передача сообщения об этом ответственным за безопасность в аэропорту.

Еще одной интересной областью искусственного интеллекта является создание искусственных **систем совершения открытий**. Такие системы представляют собой компьютерные программы, которые действительно способны обнаруживать знания в определенных предметных областях. Например, программа AM (Automated Mathematician — автоматизированный математик) открыла несколько новых математических теорем и повторно совершила открытие, ранее сделанное людьми, которое касается значимости для математики простых чисел. Система совершения открытий BACON 3 получила новые научные знания, такие как одна из версий третьего закона планетарного движения Кеплера; общие сведения о многих системах совершения открытий приведены в [92].

В XX веке искусственный интеллект был первоначально определен как отрасль компьютерных наук, но теперь он рассматривается как отдельная дисциплина, которая совмещается с многими научными областями, такими как компьютерная наука, психология, биология, неврология и т.д., поэтому все большее число университетов предоставляют возможность защитить научную степень в области искусственного интеллекта.

На рис. 1.1 показаны некоторые области, представляющие интерес для искусственного интеллекта. Область экспертных систем представляет собой очень успешное приближенное решение классической задачи ИИ — программирование интеллекта. Профессор Эдвард Фейгенбаум из Станфордского университета, один из первых исследователей технологии экспертных систем, определил понятие **экспертной системы** как “. . . интеллектуальной компьютерной программы, в которой используются знания и процедуры логического вывода для решения задач, достаточно трудных для того, чтобы требовать для своего решения значительного объема экспертных знаний человека”. Таким образом, экспертная система — это компьютерная система, которая эмулирует способности эксперта к принятию решений. Термин **эмулирует** означает, что экспертная система обязана действовать во всех отношениях как эксперт-человек. Понятие эмуляции является гораздо более строгим, чем моделирование, поскольку моделирующая система обязана действовать подобно реальному объекту лишь в определенных отношениях.

Безусловно, еще не удалось создать такую систему, которая могла бы применяться в качестве универсального решателя задач, но экспертные системы действуют в своих ограниченных областях приложения весьма успешно. В качестве доказательства их успеха достаточно лишь отметить, как много приложений экспертных систем существует в наши дни в бизнесе, медицине, науке и технике и как много книг, журналов, конференций и программных продуктов посвящено

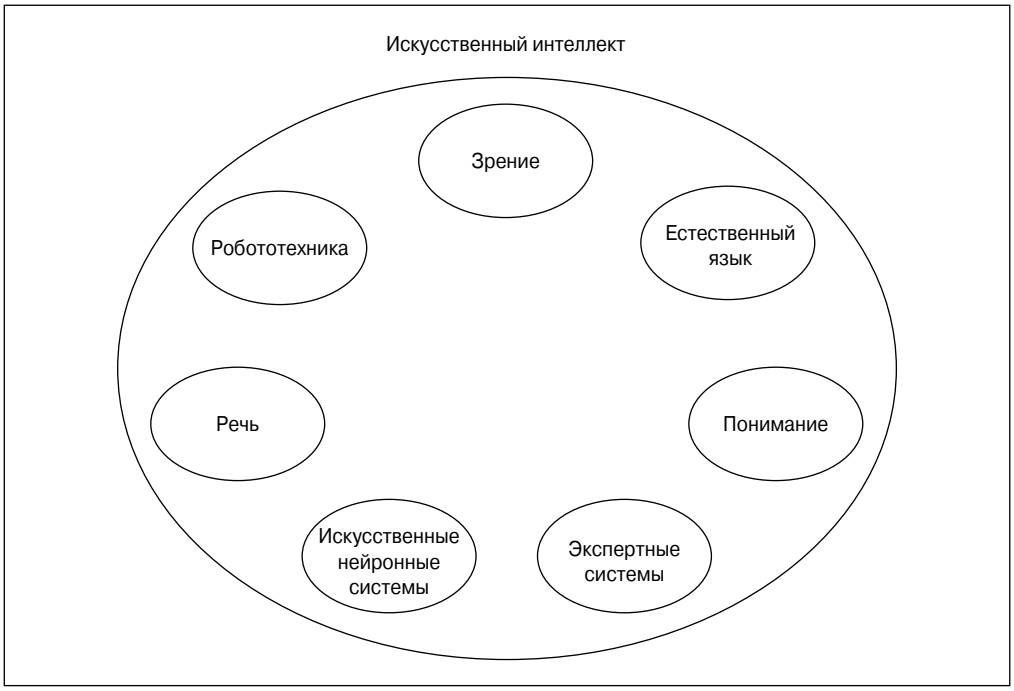


Рис. 1.1. Некоторые области искусственного интеллекта

экспертным системам. Некоторые сведения по этой теме приведены в приложении Ж.

В экспертных системах для решения задач на уровне эксперта-человека широко используются специализированные знания. Термином **эксперт** обозначается личность, обладающая экспертными знаниями в определенной области. Это означает, что эксперт имеет знания или специальные навыки, которые неизвестны или недоступны для большинства людей. Эксперт способен решать задачи, которые большинство людей не способны решить вообще, или решает их гораздо более эффективно (но не обязательно требует меньшую оплату по сравнению с обычными людьми). После того как были впервые разработаны экспертные системы, они содержали исключительно только экспертные знания. Однако в наши дни термин *экспертная система* часто применяется по отношению к любой системе, в которой используется технология экспертных систем. Технология экспертных систем может включать специальные языки экспертных систем, а также программные и аппаратные средства, предназначенные для обеспечения разработки и эксплуатации экспертных систем.

В качестве знаний в экспертных системах могут применяться либо экспертные знания, либо обычные общедоступные знания, которые могут быть получены

из книг, журналов и от хорошо осведомленных людей. В этом смысле обычные знания рассматриваются как понятие более низкого уровня по сравнению с более редкими экспертными знаниями. Термины **экспертная система**, **система, основанная на знаниях**, и **экспертная система, основанная на знаниях**, часто используются как синонимы. Но большинство людей используют только термин *экспертная система*, просто потому, что он короче, даже несмотря на то, что в экспертной системе, о которой идет речь, могут быть представлены не экспертные, а всего лишь обычные знания.

Принципы работы экспертной системы, основанной на знаниях, иллюстрируются на рис. 1.2. Пользователь передает в экспертную систему факты или другую информацию и получает в качестве результата экспертный совет или экспертные знания. По своей структуре экспертная система подразделяется на два основных компонента — базу знаний и **машину логического вывода**. База знаний содержит знания, на основании которых машина логического вывода формирует заключения. Эти заключения представляют собой ответы экспертной системы на запросы пользователя, желающего получить экспертные знания.

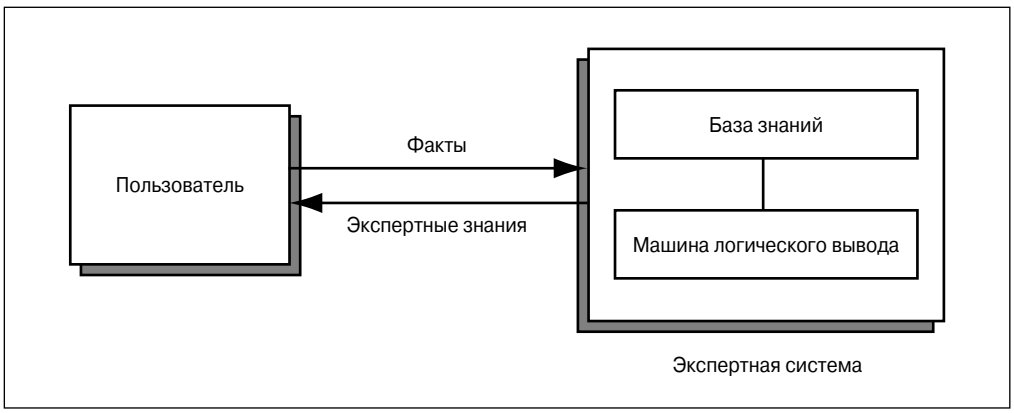


Рис. 1.2. Основные принципы функционирования экспертной системы

Кроме того, разработаны полезные системы, основанные на знаниях, которые предназначены для использования в качестве интеллектуального помощника для эксперта-человека. Эти интеллектуальные помощники проектируются на основе технологии экспертных систем, поскольку такая технология обеспечивает значительные преимущества при разработке. Чем больше знаний будет введено в базу знаний интеллектуального помощника, тем в большей степени его действия будут напоминать действия эксперта. Разработка интеллектуального помощника может стать полезным промежуточным шагом перед созданием полноценной экспертной системы. К тому же интеллектуальный помощник позволяет освободить для эксперта больше полезного времени, поскольку его применение способствует уско-

ренному решению задач. Еще одним приложением искусственного интеллекта являются интеллектуальные обучающие программы. В отличие от старых систем компьютеризированного обучения, интеллектуальные обучающие программы способны предоставлять учащемуся инструкции, зависящие от контекста [37].

Знания эксперта относятся только к одной **предметной области**, и в этом состоит отличие методов, основанных на использовании экспертных систем, от общих методов решения задач. *Предметная область* — это специальная проблемная область, такая как медицина, финансы, наука и техника, в которой может очень хорошо решать задачи лишь определенный эксперт. Экспертные системы, как и эксперты-люди, в целом предназначены для использования в качестве экспертов в одной предметной области. Например, обычно нельзя рассчитывать на то, что эксперт в области шахмат будет обладать экспертными знаниями, относящимися к медицине. Экспертные знания в одной предметной области не переносятся автоматически на другую область.

Знания эксперта, касающиеся решения конкретных задач, называются **областью знаний** эксперта. Например, медицинская экспертная система, предназначенная для диагностирования инфекционных заболеваний, должна обладать большим объемом знаний об определенных симптомах, вызванных инфекционными заболеваниями. В этом случае областью знаний является медицина, а сами знания состоят из сведений о заболеваниях, симптомах и методах лечения. Связь между предметной областью и областью знаний показана на рис. 1.3. Обратите внимание на то, что на данном рисунке область знаний полностью включена в предметную область. Часть, выходящая за пределы области знаний, символизирует область, в которой отсутствуют знания о какой-либо из задач, относящихся к данной предметной области.

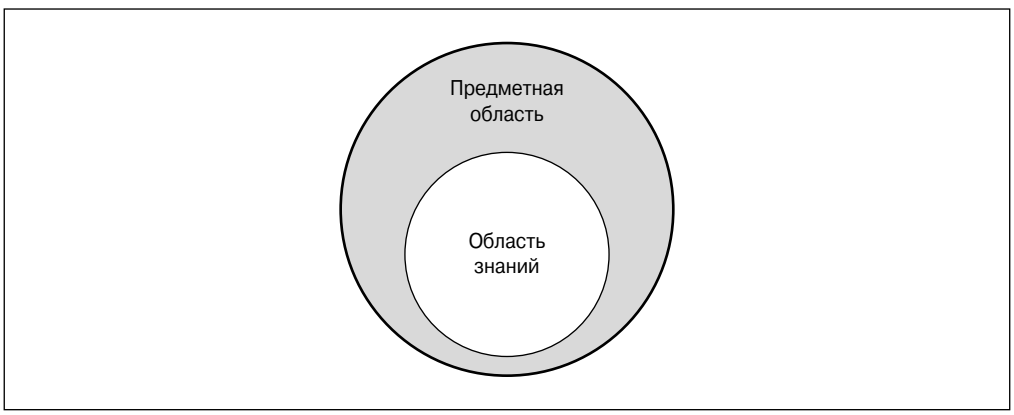


Рис. 1.3. Связь между предметной областью и областью знаний

В какой-либо конкретной экспертной системе, например, в системе диагностирования инфекционных заболеваний, обычно отсутствуют знания, относящиеся к другим областям, например, в данном случае к такой области медицины, как хирургия или педиатрия. Экспертная система может обладать знаниями об инфекционных заболеваниях, равными или превышающими знания отдельно взятого эксперта-человека, но эта экспертная система может не иметь никаких сведений из других областей знаний, если в нее не будут введены знания, относящиеся к одной из этих проблемных областей.

В области знаний, сведениями из которой располагает экспертная система, эта экспертная система проводит рассуждения или делает **логические выводы** по такому же принципу, как рассуждал бы эксперт-человек или приходил логическим путем к решению задачи. Это означает, что на основании определенных фактов путем рассуждений формируется логичное, оправданное заключение, которое следует из этих фактов. Например, если ваш близкий родственник не разговаривал с вами месяц, вы могли бы прийти к выводу, что ему просто нечего вам сказать. Но это лишь один из нескольких возможных логических выводов.

Как и по отношению к любой другой технологии, существует много способов оценки полезности технологии экспертных систем. В табл. 1.1 показано, как оценивают некоторую технологию со своей точки зрения разные категории людей, имеющих к ней отношение. В этой таблице в качестве технолога может рассматриваться инженер или разработчик программ, а в качестве технологии может рассматриваться аппаратное или программное обеспечение. В процессе решения любой задачи необходимо найти ответы на некоторые вопросы, так как в противном случае будет отсутствовать возможность успешного применения данной технологии. Кроме того, экспертные системы, как и любые другие инструментальные средства, могут иметь подходящие или неподходящие области применения. Проблема выбора подходящих областей применения для экспертных систем рассматривается более подробно в главе 6.

Таблица 1.1. Различные взгляды на некоторую технологию

Заинтересованное лицо	Вопрос
Руководитель	Для чего я могу ее использовать?
Технолог	Как я могу ее реализовать лучше всего?
Исследователь	Как я могу ее развить?
Потребитель	Позволит ли она сэкономить для меня время или деньги?
Владелец делового предприятия	Могу ли я сократить потребность в рабочей силе?
Биржевой маклер	Как она повлияет на ежеквартальную прибыль?

1.3 Преимущества экспертных систем

Как описано ниже, экспертные системы обладают многими привлекательными особенностями.

- **Повышенная доступность.** Для обеспечения доступа к экспертным знаниям могут применяться любые подходящие компьютерные аппаратные средства. В определенном смысле вполне оправдано утверждение, что экспертная система — это средство массового производства экспертных знаний.
- **Уменьшенные издержки.** Стоимость предоставления экспертных знаний в расчете на отдельного пользователя существенно снижается.
- **Уменьшенная опасность.** Экспертные системы могут использоваться в таких вариантах среды, которые могут оказаться опасными для человека.
- **Постоянство.** Экспертные знания никуда не исчезают. В отличие от экспертов-людей, которые могут уйти на пенсию, уволиться с работы или умереть, знания экспертной системы сохраняются в течение неопределенно долгого времени.
- **Возможность получения экспертных знаний из многих источников.** С помощью экспертных систем могут быть собраны знания многих экспертов и привлечены к работе над задачей, выполняемой одновременно и непрерывно, в любое время дня и ночи. Уровень экспертных знаний, скомбинированных путем объединения знаний нескольких экспертов, может превышать уровень знаний отдельно взятого эксперта-человека.
- **Повышенная надежность.** Применение экспертных систем позволяет повысить степень доверия к тому, что принято правильное решение, путем предоставления еще одного обоснованного мнения эксперту-человеку или посреднику при разрешении несогласованных мнений между несколькими экспертами-людьми. (Разумеется, такой метод разрешения несогласованных мнений не может использоваться, если экспертная система запрограммирована одним из экспертов, участвующих в столкновении мнений.) Решение экспертной системы должно всегда совпадать с решением эксперта; несовпадение может быть вызвано только ошибкой, допущенной экспертом, что может произойти, только если эксперт-человек устал или находится в состоянии стресса.
- **Объяснение.** Экспертная система способна подробно объяснить свои рассуждения, которые привели к определенному заключению. А человек может оказаться слишком усталым, не склонным к объяснениям или неспособным делать это постоянно. Возможность получить объяснение способствует повышению доверия к тому, что было принято правильное решение.
- **Быстрый отклик.** Для некоторых приложений может потребоваться быстрый отклик или отклик в реальном времени. В зависимости от используе-

мого аппаратного и программного обеспечения экспертная система может реагировать быстрее и быть более готовой к работе, чем эксперт-человек. В некоторых экстремальных ситуациях может потребоваться более быстрая реакция, чем у человека; в таком случае приемлемым вариантом становится применение экспертной системы, действующей в реальном времени.

- **Неизменно правильный, лишенный эмоций и полный ответ при любых обстоятельствах.** Такое свойство может оказаться очень важным в реальном времени и в экстремальных ситуациях, когда эксперт-человек может оказаться неспособным действовать с максимальной эффективностью из-за воздействия стресса или усталости.
- **Возможность применения в качестве интеллектуальной обучающей программы.** Экспертная система может действовать в качестве интеллектуальной обучающей программы, передавая учащемуся на выполнение примеры программ и объясняя, на чем основаны рассуждения системы.
- **Возможность применения в качестве интеллектуальной базы данных.** Экспертные системы могут использоваться для доступа к базам данных с помощью интеллектуального способа доступа. В качестве примера можно привести анализ скрытых закономерностей в данных.

Кроме того, процесс разработки экспертной системы приносит и косвенное преимущество, так как знания экспертов-людей должны быть предварительно преобразованы в явную форму для ввода в компьютер. Поскольку знания затем становятся явно известными, а не присутствуют неявно в мозгу эксперта, появляется возможность проверять знания на правильность, непротиворечивость и полноту. После такой проверки, возможно, придется откорректировать эти знания (за что эксперт вряд ли будет признателен!).

1.4 Общие понятия экспертных систем

Знания могут быть представлены в экспертной системы многими способами. Одним из широко применяемых методов представления знаний являются **правила** в форме `IF THEN`, как показано ниже.

```
IF горит красный свет THEN стоять
```

Если обнаруживается факт, что на светофоре горит красный свет, то этот факт согласуется с шаблоном “на светофоре горит красный свет”. Условие правила удовлетворяется и выполняется обусловленное в нем действие “стоять”. Хотя этот пример с виду кажется очень простым, создано много важных экспертных систем, основанных на представлении знаний экспертов в виде правил. В действительности подход к разработке экспертных систем, основанный на знаниях, полностью вытеснил применявшийся ранее (в 1950–1960-х гг.) подход к созданию

искусственного интеллекта, в котором предпринимались попытки использовать сложные методы формирования рассуждений без опоры на знания. В инструментальных средствах экспертных систем некоторых типов, таких как CLIPS, допускается применение не только правил, но и **объектов**. Знания могут быть представлены и в правилах, и в объектах. С условиями правил могут согласовываться не только факты, но и объекты. Еще один вариант состоит в том, что объекты могут применяться независимо от правил.

Первым успешным коммерческим применением экспертных систем стало создание системы XCON/R1 в компании Digital Equipment Corporation. Оказалось, что эта система обладает намного более широкими знаниями о том, как следует создавать конфигурации компьютерных систем, по сравнению с любым отдельно взятым экспертом-человеком. С тех пор экспертные системы снова и снова демонстрировали свою важность и полезность. Было создано много небольших систем для решения специализированных задач с несколькими сотнями правил. Такие небольшие системы могут не оперировать на уровне эксперта, но в принципе позволяют воспользоваться преимуществами технологии экспертных систем для решения задач, требующих большого объема знаний. Знания, необходимые для подобных небольших систем, могут быть взяты из книг, журналов или любой другой общедоступной документации.

В отличие от этого классическая экспертная система воплощает в себе неписанные знания, которые должны быть получены от эксперта с помощью обширных интервью, проводимых **инженером по знаниям** в течение длительного периода времени. Такой процесс создания экспертной системы называется **инженерией знаний** и осуществляется инженером по знаниям. *Инженерией знаний* называют получение знаний от эксперта-человека или из других источников и последующее представление знаний в экспертной системе.

Основные этапы разработки экспертной системы показаны на рис. 1.4. Вначале инженер по знаниям устанавливает диалог с экспертом-человеком, чтобы выявить знания эксперта. Этот этап аналогичен этапу работы, выполняемому системным проектировщиком при обычном программировании в ходе обсуждения требований к системе с клиентом, для которого создается программа. Затем инженер по знаниям представляет знания в явном виде для внесения в базу знаний. После этого эксперт проводит оценку экспертной системы и передает критические замечания инженеру по знаниям. Такой процесс повторяется снова и снова, до тех пор, пока эксперт не оценит результаты работы системы как удовлетворительные.

Выражение **система, основанная на знаниях**, представляет собой лучший термин для обозначения технологии, основанной на знаниях, поскольку он может применяться для обозначения и экспертных систем, и систем, основанных на знаниях. Но термин *экспертная система* (подобно термину *искусственный интеллект*) в наши дни принято использовать для обозначения и экспертных систем,

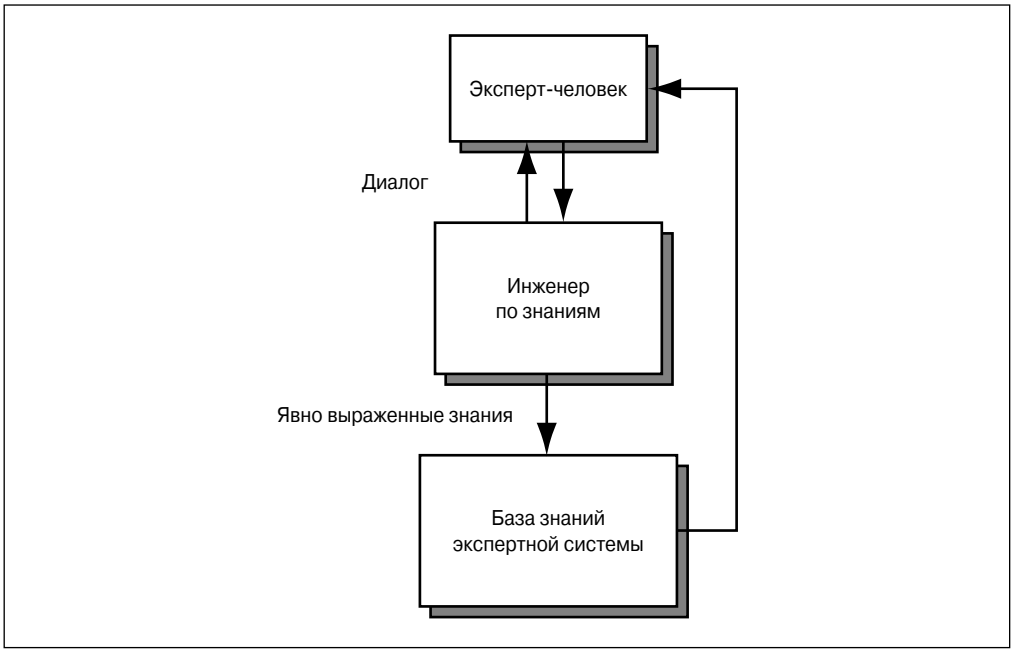


Рис. 1.4. Процесс разработки экспертной системы

и систем, основанных на знаниях, даже если знания, представленные в системе, не находятся на уровне, достойном эксперта-человека.

Вообще говоря, процесс создания экспертных систем намного отличается от процесса разработки обычных программ, ведь в экспертных системах рассматриваются задачи, не имеющие удовлетворительного алгоритмического решения, поэтому для достижения приемлемого решения используются логические выводы. *Алгоритм* — это идеальное решение задачи, поскольку он гарантирует получение ответа за конечное время [6]. Но алгоритм может стать неудовлетворительным после увеличения масштабов той же задачи, и с этим связана необходимость применения искусственного интеллекта. Следует отметить, что приемлемое решение — это почти самое лучшее, на что мы можем рассчитывать, если отсутствует алгоритм, который позволил бы нам достичь оптимального решения. Поскольку в основе функционирования экспертной системы лежит логический вывод, такая система должна обладать способностью объяснить свои рассуждения, чтобы можно было их проверить. Поэтому неотъемлемой частью любой сложной экспертной системы является **средство объяснения**. В действительности могут быть разработаны сложные средства объяснения, позволяющие пользователю исследовать многочисленные строки с вопросами наподобие "*Что будет, если...*", называемые **гипотетическими рассуждениями**.

Некоторые технологии экспертных систем позволяют даже создавать системы, изучающие правила на примерах с помощью **вывода правил методом индукции**, в котором система создает правила на основе данных. Формализация знаний экспертов в виде правил — нелегкая проблема, особенно если знания экспертов еще до сих пор систематически не исследовались. В знаниях эксперта могут быть несовместимости, двусмысленности, повторы или другие недостатки, которые не становятся очевидными до тех пор, пока не будут предприняты попытки формально представить эти знания в экспертной системе.

К тому же эксперты должны знать пределы своих знаний и объективно оценивать качество своего совета в тех условиях, когда задача по своей сложности достигает их **границ незнания**. Кроме того, человек, выполняющий роль эксперта, знает, когда нужно "нарушить правила". А если в проект экспертной системы явно не заложены способности справляться с неопределенностью, то система будет выдавать рекомендации с прежней уверенностью, даже если предоставляемые ей данные являются слишком неточными или неполными. Рекомендации экспертной системы, как и рекомендации эксперта-человека, должны корректно показывать снижение своего качества по мере приближения к границам незнания, а не внезапно становиться ложными.

В наши дни одним из практических ограничений многих экспертных систем является нехватка **причинных знаний**. Это означает, что в экспертной системе фактически отсутствуют сведения об основополагающих причинах и результатах в контролируемой системе. Дело в том, что экспертные системы гораздо проще запрограммировать с использованием **поверхностных знаний**, основанных на эмпирических и эвристических знаниях, чем с помощью **глубоких знаний**, основанных на понимании базовой структуры, функций и поведения объектов. Например, гораздо проще запрограммировать экспертную систему, которая назначала бы прием аспирина для пациента с головной болью, чем запрограммировать все базовые биохимические, физиологические, анатомические и неврологические знания о человеческом теле. Программирование причинно-следственной модели человеческого тела оказалось бы чрезвычайно сложной задачей, и даже в случае ее успешного выполнения время отклика системы, по-видимому, было бы исключительно продолжительным, поскольку приходилось бы обрабатывать всю информацию, имеющуюся в системе.

Одним из типов поверхностных знаний являются **эвристические знания**; термин *эвристика* происходит от греческого слова и означает *обнаружение*. В отношении эвристических решений не предоставляется гарантия того, что они окажутся успешными в такой же степени, как и алгоритмы, обеспечивающие гарантированное решение задачи. Вместо этого эвристики представляют собой эмпирические правила или эмпирические знания, приобретенные на основании опыта, которые могут способствовать достижению решения, но не гарантируют успешную работу. Однако во многих областях, таких как медицина и инженерия, эври-

стики играют важную роль в решении задач некоторых типов. А иногда, даже если известно точное решение, может оказаться непрактичным применение алгоритма для его получения из-за ограничений по стоимости или времени. Эвристики могут обеспечить поиск удобных сокращенных путей, позволяющих уменьшить затраты времени и ресурсов.

Еще одна проблема, с которой сталкиваются экспертные системы, состоит в том, что представленные в них экспертные знания ограничиваются областью знаний, представленной в системе. Типичные экспертные системы не способны обобщать свои знания с использованием **аналогий** для рассуждения о новых ситуациях, а люди на это способны. Безусловно, такое состояние дел немного улучшается благодаря применению индукции правил, но указанный способ позволяет помещать в экспертную систему лишь определенные ограниченные типы знаний. Общепринятый способ создания экспертной системы состоит в том, что инженер по знаниям повторяет цикл, состоящий из этапов проведения интервью с экспертом, создания прототипа, проверки, проведения еще одного интервью и т.д.; поэтому создание системы представляет собой трудоемкую задачу, отнимающую много времени. В действительности указанная проблема переноса человеческих знаний в экспертную систему является столь важной, что получила название **узкого места в процессе приобретения знаний**. Это — описательный термин, поскольку узкое место в процессе приобретения знаний влияет на процесс создания экспертной системы так же, как узкое место в горлышке бутылки влияет на заполнение ее жидкостью.

Несмотря на указанные ограничения экспертных систем, существующие в настоящее время, экспертные системы показали свою способность успешно решать практические задачи, которые невозможно было решить с помощью обычных методологий программирования, особенно в тех условиях, когда приходится пользоваться неопределенной или неполной информацией. Это означает, что очень важно знать преимущества и недостатки любой технологии, чтобы использовать ее должным образом.

1.5 Характеристики экспертной системы

Экспертная система обычно проектируется таким образом, чтобы она обладала описанными ниже общими характеристиками.

- Высокая эффективность. Система должна обладать способностью давать ответы на уровне компетентности, равной или более высокой по сравнению с экспертом в данной области. Это означает, что качество рекомендаций, предоставляемых системой, должно быть неизменно высоким.
- Приемлемое время отклика. Система должна выполнять свою работу за приемлемое время, сопоставимое или лучшее по сравнению с тем, которое

требуется эксперту, чтобы выработать решение. Экспертная система, для которой требуется год, чтобы получить решение, тогда как эксперту-человеку достаточно одного часа, никому не нужна. К тому же **временны́е ограничения**, регламентирующие производительность экспертной системы, могут оказаться особенно жесткими в случае систем реального времени, когда ответ должен быть получен в течение определенного интервала времени, например, как при посадке самолета в тумане.

- Высокая надежность. Экспертная система должна быть надежной и не подверженной сбоям, так как в противном случае ее использование станет невозможным.
- Доступность для понимания. Система должна быть способной объяснить все этапы своих рассуждений, осуществляемых в ходе выработки решения, чтобы ее работа была доступной для понимания. Система не может быть просто "черным ящиком", который вырабатывает загадочный ответ, и должна предоставлять возможность получить объяснение по такому же принципу, как эксперты-люди могут объяснить свои рассуждения. Такая характеристика важна по нескольким описанным ниже причинам.

Одна из причин состоит в том, что от ответов экспертной системы иногда зависят жизнь и собственность человека. Поскольку ошибки могут обойтись очень дорого, экспертная система должна быть способной обосновать свои заключения по такому же принципу, как эксперт-человек может объяснить, на каком основании он пришел к определенному заключению. Таким образом, применение средства объяснения позволяет пользователям провести профилактическую проверку рассуждений системы.

Еще одна причина, по которой желательно предусмотреть средство объяснения, связана с тем, что на этапе разработки экспертной системы необходимо неоднократно убеждаться в том, что система правильно получила знания и правильно их использует. Такая возможность является исключительно важной при отладке, поскольку знания могут быть неправильно введены из-за опечаток или просто оказаться ошибочными из-за того, что инженер по знаниям и эксперт неправильно поняли друг друга. Хорошее средство объяснения позволяет эксперту и инженеру по знаниям проверять точность введенных знаний. Кроме того, учитывая то, какой способ обычно применяется для создания экспертных систем, очень сложно читать длинные листинги программ и разбираться в их работе.

Дополнительным источником ошибок могут оказаться непредвиденные взаимодействия в экспертной системе. Такие взаимодействия могут быть обнаружены путем прогона тестовых примеров, связанных с известными рассуждениями, которым должна следовать система. Как будет подробно описано ниже, к каждой конкретной ситуации, в отношении которой система проводит свои рассуждения, может быть применено множество правил. Поток выполнения в экспертной си-

стеме не является последовательным; например, невозможно просто прочитать код программы строка за строкой и понять, как работает та или иная система. Порядок, в котором в систему были введены правила, не обязательно становится порядком, в котором эти правила выполняются. Экспертная система во многом функционирует как параллельная программа, в которой правила действуют как независимые процессоры знаний.

- Гибкость. Безусловно, любая экспертная система может содержать большой объем знаний, поэтому важно иметь эффективный механизм добавления, модификации и удаления знаний. Одна из причин, по которой системы на основе правил нашли столь широкое распространение, обусловлена предусмотренными в них возможностями эффективного и модульного хранения правил.

В зависимости от системы средство объяснения может быть простым или сложным. Простое средство объяснения в системе, основанной на правилах, может просто предусматривать составление списка всех фактов, которые привели к выполнению самого последнего правила, а более сложные системы могут выполнять действия, описанные ниже.

- Составление списка всех соображений за и против конкретной гипотезы. Термин **гипотеза** обозначает **цель**, которая должна быть доказана; например, в медицинской диагностической экспертной системе гипотезой может стать утверждение: “У пациента имеется столбнячная инфекция”. При решении реальной задачи может быть выдвинуто несколько гипотез, или, как в данном примере, пациент может фактически иметь одновременно несколько заболеваний. Гипотеза представляет собой утверждение, истинность которого находится под сомнением и которое должно быть доказано. После того как выдвинута цель, формируются **подцели**, являющиеся более простыми, и этот процесс продолжается до тех пор, пока не появятся достаточно простые подцели, которые могут быть решены. Такой метод решения задачи представляет собой классический метод, организованный по принципу **разделяй и властвуй**, который является основой обратного логического вывода, рассматриваемого в одной из следующих глав.
- Составление списка всех гипотез, которые позволяют объяснить наблюдаемое свидетельство.
- Поиск объяснений для всех следствий гипотезы. Например, если принято предположение, что у пациента действительно имеется столбняк, то должны быть свидетельства о наличии высокой температуры, возникающей в результате развития инфекционного заболевания. Если этот симптом в дальнейшем действительно обнаруживается, то степень доверия к тому, что гипотеза истинна, увеличивается. Если же этот симптом не наблюдается, то степень доверия к истинности гипотезы уменьшается.

- Выдача **прогноза**, или предсказания того, что произойдет, если гипотеза является истинной.
- Обоснование необходимости предъявления программой вопросов пользователю для получения дополнительной информации. Эти вопросы можно использовать, чтобы направить цепь рассуждений по наиболее перспективным путям получения диагноза. При решении большинства практических задач попытка исследовать все возможности оказывается слишком дорогостоящей или требующей много времени, поэтому должны быть предусмотрены определенные способы, позволяющие направить поиск к правильному решению. Например, достаточно представить себе, какие затраты времени и денег и какие сложности связаны с проведением всех возможных медицинских анализов для пациента, который жалуется на фарингит (очень выгодных для медицинского предприятия, но очень затруднительных для пациента).
- Обоснование знаний, используемых в программе. Например, если программа указывает, что является истинной гипотеза "У пациента имеется столбнячная инфекция", то пользователь может потребовать объяснения. А программа может обосновать свой вывод на основании правила, которое гласит, что если пациент имеет положительные результаты анализа крови на наличие столбняка, то у пациента имеется столбняк. После этого пользователь может потребовать, чтобы программа обосновало это правило. Программа может ответить, указав, что положительные результаты анализа крови на наличие определенного заболевания являются доказательством наличия этого заболевания. К сожалению, при этом могут игнорироваться ложно положительные результаты, как описано в одной из следующих глав.

В данном случае программа фактически ссылается на **метаправила**, представляющие собой знания о правилах. Префикс "мета" означает "свыше" или "дальше". Создаются специальные программы, способные явно выводить новые правила с использованием процесса, называемого **машинным обучением**. Гипотеза обосновывается знаниями, а знания — **гарантией** того, что эти знания являются правильными. Гарантия по существу представляет собой метаобъяснение, которое показывает, почему объяснение экспертной системой своих рассуждений является правильным.

В системе, основанной на правилах, знания могут легко наращиваться **инкрементно**; в этом состоит одна из причин, по которой экспертные системы оказались настолько успешными. Таким образом, база знаний может постепенно наращиваться по мере добавления правил, с тем, чтобы можно было непрерывно проверять производительность и обоснованность системы. Этот способ аналогичен тому, с помощью которого ребенок каждый день усваивает новые знания и проверяет их правильность. Если правила спроектированы должным об-

разом, то взаимодействия между правилами могут быть сведены к минимуму или полностью устранены в целях защиты от непредвиденных побочных эффектов. Возможность инкрементного наращивания знаний способствует упрощению процесса **ускоренного создания прототипа**, чтобы инженер по знаниям мог быстро показать работающий прототип экспертной системы. Эта особенность является очень важной, поскольку поддерживает заинтересованность экспертов и руководителей в успехе проекта. Кроме того, ускоренное создание прототипа позволяет быстро обнаруживать любые пропуски, несовместимости и ошибки в знаниях эксперта или в системе, чтобы можно было немедленно внести исправления.

1.6 Разработка технологии экспертных систем

Истоки экспертных систем лежат во многих научных дисциплинах; особо следует отметить область психологии, посвященную исследованию обработки информации человеком, — **когнитологию**, или науку о познании. Изучение процесса познания — это изучение того, как люди усваивают или обрабатывают информацию. Иными словами, когнитология — это наука о том, как мыслят люди, особенно в ходе решения задач. Для обеспечения лучшего обучения был разработан целый ряд когнитивных инструментальных средств [63].

Еще один важный принцип, который должен быть воплощен в машинах искусственного интеллекта, состоит в том, что такие машины должны обладать способностью распознавать знаки; проведение исследований в этом направлении является основой научной области, называемой **семиотикой** [24]. Семиотика изучает не такие простые, визуально различимые знаки, как знак Stop, а все понятие знака в целом. *Зна́ком* называется то, что представляет нечто другое. Например, если вы смотрите фильм с музыкальным сопровождением, то общепринятым знаком, показывающим, что вскоре произойдет какое-то приятное событие, является повышение тона и ускорение ритма музыки. Аналогичным образом, знаком, свидетельствующим о том, что должно произойти неприятное событие, является замедление музыки и снижение тона. В музыке, фильмах, на телевидении и в повседневной жизни применяется много различных невербальных знаков. Например, если человек лжет или не может откровенно ответить на какой-то вопрос, то обычно опускает глаза. До сих пор нагрузка по восприятию всех этих знаков, возлагаемая на интеллектуальные машины, разрабатываемые для использования в реальном мире, была слишком высока. Простого программирования, позволяющего понимать слова́, недостаточно; такие машины должны также понимать основополагающий смысл знаков.

Изучение процесса познания является очень важным, если мы хотим наделить компьютеры способностью эмулировать экспертов-людей. Эксперты часто не могут объяснить, как они решают ту или иную задачу — решение просто само

приходит к ним в голову. Если эксперт не может объяснить, как он решает задачу, то невозможно представить знания эксперта в экспертной системе в форме явных знаний. В таком случае единственная возможность состоит в использовании программ, которые сами обучаются эмулировать действия эксперта. Таковыми являются программы, основанные на индукции, искусственных нейронных системах и других гибких вычислительных методах, которые будут обсуждаться позднее.

Решение задач человеком и продукционные правила

История разработки технологии экспертных систем является очень обширной. В табл. 1.2 представлены краткие сведения о некоторых важных достижениях, которые привели к созданию языка CLIPS; там, где это было возможно, показаны даты начала проектов. Разработка значительной части проектов продолжалась в течение многих лет. Открытия, сделанные в ходе этого, рассматриваются более подробно в настоящей главе и следующих главах. Целый ряд инструментальных средств экспертных систем, а также современных экспертных систем описан в приложении Ж.

Таблица 1.2. Некоторые важные события и открытия, которые привели к появлению первого выпуска языка CLIPS

Год	События и открытия
1943	Порождающие правила Поста; модель нейрона Маккаллоха и Питтса
1954	Управление выполнением правил с помощью марковского алгоритма
1956	Дартмутская конференция; программа Logic Theorist (Логик-теоретик); эвристический поиск; введение в научный обиход термина “искусственный интеллект”
1957	Изобретение персептрона Розенблаттом; начало работ над программой GPS (General Problem Solver — универсальный решатель задач) (Ньюэлл, Шоу и Саймон)
1958	Язык искусственного интеллекта LISP (Маккарти)
1962	Выход из печати книги Розенблатта *Principles of Neurodynamics*, посвященной восприятию
1965	Применение метода резолюции для автоматического доказательства теорем (Робинсон); применение нечеткой логики для рассуждений о нечетко заданных объектах (Заде); начало работ над первой экспертной системой DENDRAL (Фейгенбаум, Бьюкенен)
1968	Семантические сети, модель ассоциативной памяти (Квиллиан)
1969	Математическая экспертная система MACSYMA (Мартин и Мозес)
1970	Начало работ над языком PROLOG (Колмероэ, Русселл)

Окончание табл. 1.2

Год	События и открытия
1971	Разработка системы распознавания речи HEARSAY I; популярное изложение подхода на основе правил в книге *Human Problem Solving* (Ньюэлл и Саймон)
1973	Создание экспертной системы для медицинской диагностики MYCIN (Шортлифф), повлекшее за собой разработку системы GUIDON, концепции интеллектуального обучения (Клэнси) и системы TEIRESIAS, формулировка принципов применения средства объяснения (Дэвис) и создание EMYCIN, первого командного интерпретатора (Ван Мелле, Шортлифф и Бьюкенен); разработка системы HEARSAY II; применение модели классной доски для организации сотрудничества нескольких экспертов
1975	Фреймы, представление знаний (Минский)
1976	Создание программы AM (Artificial Mathematician — автоматизированный математик), обеспечивающей творческое открытие математических понятий (Ленат); применение теории доказательств Демпстера–Шефера для проведения рассуждений в условиях неопределенности; начало работ над экспертной системой PROSPECTOR, предназначенной для разведки полезных ископаемых (Дуда, Харт)
1977	Разработка командного интерпретатора экспертной системы OPS (Форги), предшественника CLIPS
1978	Начало работ над системой XCON/R1 (Макдермотт, компания DEC), предназначенной для определения конфигураций компьютерных систем DEC; разработка системы Мета-DENDRAL, формулировка понятия метаправил и обоснование принципа вывода правил по методу индукции (Бьюкенен)
1979	Применение rete-алгоритма для быстрого сопоставления с шаблонами (Форги); начало коммерциализации искусственного интеллекта; создание компании Inference Corp. (эта компания выпустила инструментальные средства экспертных систем ART в 1985 году)
1980	Открытия в области символической логики; основана компания LMI для промышленного производства машин LISP
1982	Разработка математической экспертной системы SMP; создание нейронной сети Хопфилда; начало работ в Японии над проектом создания компьютеров пятого поколения, целью которого явилась разработка интеллектуальных компьютеров
1983	Создание инструментального средства экспертных систем KEE (компания IntelliCorp)
1985	Выпуск версии 1 инструментального средства экспертных систем CLIPS (агентство NASA). Эти программы свободно доступны для использования на всех компьютерах, а не только на имеющих специальное назначение и дорогостоящих компьютерах LISP

В конце 1950–начале 1960-х годов было разработано несколько программ в целях обеспечения возможности поиска универсального способа решения задач. Наиболее известной из них была программа GPS, **универсального решателя задач**, созданная Ньюэллом (Newell) и Саймоном (Simon). Появление этой программы вызвало огромную сенсацию, поскольку в прессе за большими компьютерами, которые в то время занимали целую комнату, закрепилось название "гигантские мозги". Люди были охвачены страхом потерять свою работу. Это продолжалось до тех пор, пока крупные компании, подобные IBM, не объявили, что компьютеры будут заниматься только той работой, для выполнения которой тысячам математиков потребовались бы миллионы лет. В те дни, когда вычислительные машины стоили миллионы долларов, а машинное время продавалось по цене 1 доллар за секунду, казалось невозможным то, что люди будут когда-то иметь недорогие персональные компьютеры у себя дома или на работе.

Одним из наиболее значительных результатов, продемонстрированных Ньюэллом и Саймоном, оказалось то, что значительная часть человеческого понимания, или **познания**, можно представить в виде **продукционных правил** IF-THEN (ЕСЛИ-ТО). Например, `IF "создается впечатление, что пойдет дождь" THEN "возьмите с собой зонтик"` или `IF "ваша супруга в плохом настроении" THEN "не показывайте, что вы необычайно довольны жизнью"`. Правило соответствует небольшой, модульной коллекции знаний, называемой **фрагментом**. Фрагменты организованы в свободной форме и снабжены связями, которые ведут к относящимся к ним фрагментам знаний. В одной из теорий так и утверждается, что вся человеческая память организована в виде фрагментов. Ниже приведен пример правила, представляющего фрагмент знаний.

```
IF двигатель автомобиля не запускается и
   бензиномер остановился на нуле
THEN заполните бак бензином
```

Ньюэлл и Саймон популяризировали использование правил для представления человеческих знаний и показали, как могут быть выполнены стандартные рассуждения с помощью правил. А психологи, специализирующиеся в области когнитивной психологии, использовали правила в качестве модели для объяснения процесса обработки информации человеком. Основная идея состоит в том, что мозг выделяет стимулы из сенсорной входной информации. Стимулы активизируют в **долговременной памяти** соответствующие правила, с помощью которых формируется соответствующий отклик. Долговременная память представляет собой место хранения постоянных знаний человека. Например, все люди усвоили примерно такие правила:

```
IF наблюдается пламя THEN происходит пожар
IF наблюдается дым THEN где-то может происходить пожар
IF слышна пожарная сирена THEN где-то может происходить
   пожар
```

Обратите внимание на то, что последние два правила не выражены с полной определенностью. Огонь может потухнуть, а в воздухе еще останется дым. Аналогичным образом, вой пожарной сирены не означает, что где-то происходит пожар, поскольку тревога может оказаться ложной. Стимулы, возникающие, когда человек видит огонь, чувствует запах дыма и слышит вой сирены, могут активизировать эти и другие правила подобного типа.

Долговременная память человека содержит много правил, имеющих простую структуру `IF-THEN`. В действительности шахматный гроссмейстер может хранить в своей памяти 50 тысяч или больше фрагментов знаний о шаблонах шахматных позиций. В отличие от долговременной памяти, **кратковременная память** используется для временного хранения знаний в период решения задачи. Хотя долговременная память может хранить сотни тысяч или даже больше фрагментов, емкость кратковременной, или рабочей, памяти удивительно мала — от четырех до семи фрагментов. В качестве простого примера, подтверждающего этот факт, попытайтесь мысленно представить себе визуально несколько цифр. Большинство людей способны одновременно видеть внутренним взглядом от четырех до семи цифр. Безусловно, люди способны запоминать гораздо больше, чем только цифры в количестве от четырех до семи. Но эти цифры и многие другие сведения успешно хранятся в долговременной памяти, причем для обеспечения их записи на постоянное хранение требуется время.

В одной из теорий предполагается, что кратковременная память представляет то количество фрагментов, которые могут быть активными одновременно; в этой теории процесс решения задач человеком рассматривается как перераспределение таких активизированных фрагментов в мозгу. В конечном итоге фрагмент может быть активизирован с такой интенсивностью, что будет выработана сознательная мысль, и человек, сидящий слишком близко у камина, отметит про себя: “Хм... что-то тлеет. Неужели задымились мои брюки?”

Еще одним элементом, необходимым для решения задач человеком, является **когнитивный процессор**. Когнитивный процессор предпринимает попытки найти правила, которые должны быть **активизированы** с помощью соответствующих стимулов. Для этого недостаточно взять первое попавшееся правило. Например, одно правило указывает, когда следует заполнять бак бензином, а другое — как следует себя вести, услышав звук сирены, но вряд ли следует стремиться поскорее заполнить бак бензином, как только в воздухе раздастся звук сирены. Должно активизироваться только то правило, которое соответствует стимулу. Если имеется много правил, которые могут быть активизированы одновременно, то когнитивный процессор должен выполнить операцию **разрешения конфликтов**, чтобы определить, какое правило имеет наивысший приоритет. После этого должно быть выполнено соответствующее правило с наивысшим приоритетом. Например, предположим, что активизированы два следующих правила:

```
IF происходит пожар THEN покинуть помещение
IF одежда начала тлеть THEN отодвинуться от огня
```

В таком случае необходимо вначале выполнить действие одного правила, а затем — другого. В современных экспертных системах когнитивному процессору соответствует **машина логического вывода**.

Модель решения задач человеком, предложенная Ньюэллом и Саймоном и состоящая из долговременной памяти (правил), кратковременной памяти (рабочей памяти) и когнитивного процессора (машины логического вывода), представляет собой фундамент современных экспертных систем, основанных на правилах.

Правила, подобные этому, относятся к типу правил **продукционных систем**. В наши дни продукционные системы на основе правил представляют собой широко применяемый метод реализации экспертных систем. Отдельные правила, из которых состоит продукционная система, называются **продукционными правилами**. Важным фактором при проектировании экспертной системы является объем знаний, или **степень детализации** правил. Если степень детализации слишком мала, то понимание отдельного правила без изучения других правил становится затруднительным. Если же степень детализации слишком велика, то затрудняется модификация экспертной системы, поскольку в одном правиле чаще всего смешивается несколько фрагментов знаний. Главной положительной особенностью языка CLIPS является возможность использовать не только правила, но и объекты.

До середины 1960-х годов основные исследования в искусственном интеллекте сосредоточивались на создании интеллектуальных систем, которые мало полагались на знания в проблемной области и в основном опирались на мощные методы формирования рассуждений. Даже само название универсального решателя задач характеризует концентрацию усилий на создании машин, не предназначенных для одной конкретной области, а нацеленных на решение задач многих типов. Безусловно, методы рассуждений, используемые в универсальных решателях задач, были очень мощными, но сами эти машины так и не смогли выйти за пределы начального уровня компетентности. После предъявления новой проблемной области этим машинам приходилось заново открывать все знания, начиная от исходных принципов, поэтому такие системы не могли сравниться по своей эффективности с экспертами-людьми, которые полагались на свои знания в проблемной области.

Примером того, какой мощи позволяют достичь знания, является игра в шахматы. Безусловно, компьютеры в наши дни успешно соперничают с людьми, но люди все равно играют хорошо, несмотря на тот факт, что компьютеры выполняют вычисления в миллион раз быстрее. Исследования показали, что опытные игроки в шахматы не обладают чрезвычайными способностями к проведению рассуждений, но вместо этого полагаются на знания шаблонов расположения шахматных фигур, накопленные за многие годы игры. Как уже было сказано выше, согласно

одной из оценок, опытный шахматист обладает знаниями приблизительно о 50 тысячах шаблонов шахматных позиций. Люди очень успешно справляются с задачей распознавания шаблонов, например, показывающих расположение фигур на шахматной доске. Лучшие шахматисты не пытаются рассуждать наперед, рассматривая 50, 100 или больше возможных ходов для каждой фигуры, а анализируют игру в терминах шаблонов, позволяющих обнаруживать долгосрочные угрозы и в то же время оставаться способным заметить краткосрочные отвлекающие маневры. Именно эта стратегия, а не основанный на грубой силе метод опережающего прогнозирования ходов позволяет шахматным программам, таким как Big Blue компании IBM, побеждать всех лучших чемпионов по шахматам среди людей.

Безусловно, знания в проблемной области открывают очень большие возможности, но обычно они ограничиваются только данной конкретной областью. Например, человек, ставший ведущим игроком в шахматы, не становится автоматически экспертом в решении математических задач или даже специалистом по игре в шашки. Хотя некоторые знания могут быть перенесены в другую проблемную область, например, если речь идет о тщательном планировании ходов, но такие знания скорее представляют собой навыки, а не подлинные экспертные знания.

В начале 1970-х годов стало очевидно, что ключом к созданию машинных решателей задач, способных функционировать на уровне эксперта-человека, являются знания в проблемной области. Безусловно, методы формирования рассуждений достаточно важны, но исследования показали, что эксперты при решении задач не опираются в основном на рассуждения. В действительности в ходе решения задач экспертом рассуждения могут играть незначительную роль. Вместо этого эксперты полагаются на огромные знания в части эвристических правил и на опыт, приобретенный в течение многих лет. Если же эксперт не может решить задачу на основе своего опыта, то у него возникает необходимость провести рассуждения, начиная от исходных принципов и теоретических положений (или, что более вероятно, обратиться за помощью к другому эксперту). Способности эксперта к рассуждениям обычно не лучше по сравнению с обычным человеком, которому приходится действовать в полностью незнакомой ситуации. Первые же попытки создания мощных решателей задач, основанных только на рассуждениях, показали, что работа решателя задач оканчивается неудачей, если он должен полагаться только на рассуждения.

Понимание того, что ключом к созданию практически применимых решателей задач являются знания в проблемной области, привело к появлению успешно действующих экспертных систем. Таким образом, в наши дни успешно действующие экспертные системы представляют собой экспертные системы, основанные на знаниях, а не универсальные решатели задач. Причем та же технология, которая привела к разработке экспертных систем, позволила осуществить разработку систем, основанных на знаниях, которые не обязательно содержат экспертные знания человека.

Экспертными знаниями считаются специализированные знания, известные лишь немногим, а обычными знаниями — знания, которые, как правило, можно найти в книгах, в Web, в периодических изданиях и других общедоступных информационных ресурсах. Например, широко доступны знания о том, как решать квадратные уравнения или осуществлять интегрирование и дифференцирование. Широко применяются такие компьютерные программы, основанные на знаниях, как Mathematica и MATLAB, позволяющие выполнять автоматически и эти, и многие другие математические операции как с числовыми, так и с символическими операндами. А другие программы, основанные на знаниях, могут осуществлять управление процессом в производственных установках. В наши дни термины *система, основанная на знаниях*, и *экспертная система* часто используются как синонимы. Фактически в последнее время экспертные системы стали рассматриваться как модель программирования, или **подход** к программированию, альтернативный по отношению к обычному алгоритмическому программированию.

Широкое распространение систем, основанных на знаниях

Подход, основанный на знаниях, был принят на вооружение в 1970-х годах, и с тех пор создано множество успешно действующих экспертных систем. Эти системы описаны в данной книге более подробно, поскольку все они снабжены качественной документацией, а также представлены во многих статьях и книгах с описанием их работы и организации базы знаний. Но в целом при изучении современных экспертных систем, с которыми чаще всего приходится сталкиваться, возникает такая проблема, что заложенные в них знания рассматриваются как частная собственность и являются секретными. Компании используют экспертные системы для переноса в них знаний экспертов-людей, которые работают на эти компании, поэтому стремятся скрыть полученные знания от конкурентов и особенно от представителей закона. Рассмотрим медицинскую экспертную систему, которая ставит такой диагноз, что пациент умирает или получает телесное повреждение, и этот случай становится предметом судебного иска. В действительности истец может даже не испытывать каких-либо отрицательных последствий лечения, а просто присоединиться к групповому иску, поданному теми, кто судится по поводу нанесенного им ущерба. В таком случае и программное обеспечение, и база знаний экспертной системы подлежат проверке другими экспертами. А как показывают многие судебные процессы, всегда приходится сталкиваться с тем, что один эксперт опровергает мнение другого.

К указанной выше категории относятся классические экспертные системы, которые могут интерпретировать масс-спектрограммы для идентификации химических компонентов (DENDRAL), диагностировать заболевания (MYCIN), анализировать геологические данные в целях определения наличия нефти (DIPMETER)

и поиска полезных ископаемых (PROSPECTOR), а также создавать необходимые конфигурации компьютерных систем (XCON/R1). Публикация сведений о том, что с помощью системы PROSPECTOR было обнаружено месторождение минерального сырья стоимостью 100 миллионов долларов, а с применением системы XCON/R1 компании Digital Equipment Corporation (DEC) удалось сэкономить миллионы долларов в год, вызвала в 1980-х годах невероятный интерес к новой технологии экспертных систем. Та ветвь искусственного интеллекта, которая зародилась в 1950-х годах как исследование обработки информации человеком, теперь стала развиваться в направлении достижения коммерческого успеха по принципу разработки конкретных программ для использования на практике.

По нескольким причинам особенно важной оказалась экспертная система MYCIN. Во-первых, эта система продемонстрировала, что искусственный интеллект может успешно применяться для медицинской диагностики. Во-вторых, MYCIN оказалась тестовой площадкой для проверки новых концепций, таких как средства объяснения, автоматическое приобретение знаний и интеллектуальное обучение, которые находят свое применение во многих современных экспертных системах. В-третьих, система MYCIN оказалась важной потому, что в ней была продемонстрирована осуществимость подхода с применением **командного интерпретатора** экспертной системы, в котором программное обеспечение представлено отдельно от данных. Иными словами, данные или знания больше не должны быть жестко закодированными в программе. В действительности командный интерпретатор позволяет легко заменить базу знаний одной предметной области на другую.

Создаваемые ранее экспертные системы, такие как DENDRAL, были единственными в своем роде, поскольку разрабатывались отдельно для каждого приложения. В этих системах база знаний была представлена вместе с программным обеспечением, в котором применялись знания, — с машиной логического вывода. С другой стороны, в системе MYCIN база знаний была явно отделена от машины логического вывода. Такой подход стал чрезвычайно важным этапом в развитии технологии экспертных систем, поскольку он означал, что наиболее существенная часть экспертной системы может быть использована повторно. Таким образом, оказалось, что новая экспертная система может быть создана гораздо быстрее по сравнению с системой типа DENDRAL путем исключения старых знаний и ввода знаний о новой предметной области. Поэтому та часть системы MYCIN, которая относилась к логическому выводу и объяснению (командный интерпретатор), могла быть дополнена знаниями о новой системе. Командный интерпретатор, полученный путем удаления медицинских знаний из системы MYCIN, получил название EMYCIN (сокращение от Essential, или Empty MYCIN — основная часть MYCIN, или пустая MYCIN).

Как показано на рис. 1.5, к концу 1970-х годов сложились три понятия, которые стали основными для большинства современных экспертных систем. Этими понятиями являются правила, командный интерпретатор и знания.



Рис. 1.5. Результаты обобщения трех важных понятий, на базе которых создаются современные экспертные системы, основанные на правилах

В 1980-х годах начали появляться новые компании, которые вывели экспертные системы за пределы университетских лабораторий и приступили к созданию коммерческих программных продуктов. Было создано новое мощное программное обеспечение, а также специализированное программное обеспечение, написанное непосредственно на языке LISP и предназначенное для разработки экспертных систем. К сожалению, из-за высокой стоимости и значительного времени обучения в конечном итоге эти системы начали уходить с арены по мере того, как возрастала мощь персональных компьютеров и были введены такие инструментальные средства экспертных систем, как CLIPS. В машинах LISP собственный язык ассемблера, операционная система и весь остальной фундаментальный код создавались на языке LISP, поэтому требовался большой объем работы по сопровождению.

Система CLIPS в целях повышения быстродействия и обеспечения переносимости на другие платформы была первоначально написана на языке C, и в ней использовался мощный метод сопоставления с шаблонами, получивший название *rete-алгоритма*. Кроме того, в отличие от всех других инструментальных средств экспертных систем, система CLIPS не только предоставляется без ограничений, но и имеет хорошо документированный исходный код, который прилагается к ней. Система CLIPS может быть инсталлирована на любом компьютере с компилятором C, поддерживающим стандартный язык C по определению Кернигана (Kernigan) и Ричи (Richie), поэтому эта система была инсталлирована на компьютерах буквально всех моделей. На основе системы CLIPS был разработан целый ряд языков с такими усовершенствованными средствами, как нечеткая логика и об-

ратный логический вывод. Кроме того, как указано в предисловии, существует даже вариант этой системы, разработанный на языке Java и известный как Jess.

1.7 Приложения и предметные области экспертных систем

Обычные компьютерные программы используются для решения задач многих типов. Но, как правило, эти задачи имеют алгоритмические решения, которые могут быть легко реализованы с помощью обычных программ и языков программирования, таких как C, C++, Java, C# и т.д. Во многих прикладных областях, таких как промышленность и строительство, числовые расчеты имеют первостепенное значение. В отличие от этого, экспертные системы в основном предназначены для осуществления символических рассуждений.

Безусловно, такие классические языки искусственного интеллекта, как LISP и PROLOG, также используются для манипулирования символами, но они имеют более общее назначение по сравнению с командными интерпретаторами экспертных систем. Это не означает, что невозможно создавать экспертные системы на языках LISP и PROLOG. В наши дни термином **логическое программирование** обычно обозначается программирование, осуществляемое на языке PROLOG, хотя и было также разработано много других языков, предназначенных для этой цели. С применением языков PROLOG и LISP создано много экспертных систем. Особенно значительные преимущества язык PROLOG показывает при использовании в диагностических системах, поскольку обладает встроенными средствами обратного логического вывода. Процесс создания крупных экспертных систем становится более удобным и эффективным, если для этого применяются командные интерпретаторы и вспомогательные программы, специально предназначенные для создания экспертных систем. При решении задачи создания новой экспертной системы гораздо более эффективным является использование специализированных инструментальных средств, предназначенных для создания экспертных систем, поскольку при этом не приходится каждый раз “повторно изобретать колесо”, применяя инструментальные средства общего назначения.

Приложения экспертных систем

Экспертные системы нашли свое применение практически в каждой области знаний. Некоторые из таких систем были спроектированы для использования в качестве научно-исследовательских инструментальных средств, тогда как другие предназначались для удовлетворения важных деловых потребностей и потребностей производства. Первым крупным коммерческим успехом применения экспертной системы на обычном деловом предприятии стало создание в 1970-х годах

системы XCON компании DEC. Специалисты этой компании разработали систему XCON (первоначально носившую название R1) в сотрудничестве с Джоном Макдермоттом (John McDermott) из университета Карнеги-Меллона. Экспертная система XCON применялась для разработки конфигураций компьютерных систем, выпускавшихся компанией DEC.

В данном случае под **определением конфигурации** компьютерной системы подразумевалось то, что после получения заявки от заказчика должен быть осуществлен правильный подбор всех необходимых компонентов — программного и аппаратного обеспечения и документации. Задача определения конфигурации все еще не утратила свою важность, поскольку ее всегда приходится решать, когда поступают заказы на компьютеры, автомобили или другие товары, оформляемые с помощью Интернет путем заполнения форм. Было бы чрезвычайно неэффективно, если бы на заводе-изготовителе компьютеров или на автомобильном заводе проверялся вручную каждый заказ на наличие требуемых комплектующих и принималось решение о том, может ли быть выполнен данный конкретный заказ. В наши дни и логические соображения, и финансовые требования подсказывают, что при решении задачи своевременного определения конфигурации и доставки потребителю любого сложного товара невозможно обойтись без экспертной системы.

Уже были созданы тысячи экспертных систем, и сведения о них опубликованы в компьютерных журналах, в Интернет, в книгах и на конференциях. Опубликованные сведения, несомненно, представляют собой только вершину айсберга, поскольку многие компании и правительственные организации не сообщают подробных сведений о своих системах, так как в них содержатся знания, находящиеся в частной собственности, или секретные знания. На основе сведений об экспертных системах, опубликованных в открытой печати, можно выделить несколько широких классов приложений экспертных систем, как показано в табл. 1.3. А в табл. 1.4–1.9 перечислены наиболее широко известные классические системы, поскольку они имеют качественную документацию и стали прообразами современных систем, имеющих аналогичные приложения.

Таблица 1.3. Наиболее общие классы экспертных систем

Класс	Основная область применения
Определение конфигурации	Сборка наиболее подходящих компонентов системы надлежащим способом
Диагностика	Выявление основополагающих причин проблем на основе наблюдаемых свидетельств
Инструктаж	Интеллектуальное обучение, в ходе которого учащийся может задавать вопросы “почему”, “как” и “что, если”, по такому же принципу, как и при обучении с участием преподавателя-человека

Окончание табл. 1.3

Класс	Основная область применения
Интерпретация	Подготовка объяснений для наблюдаемых данных
Текущий контроль	Сравнение наблюдаемых данных с ожидаемыми для оценки производительности
Планирование	Выработка плана действий, позволяющего достичь желаемого результата
Прогнозирование	Предсказание того, к чему приведет указанная ситуация
Устранение нарушения в работе	Поиск способа устранения проблемы
Управление	Обеспечение выполнения процесса. Может потребовать интерпретации, диагностирования, текущего контроля, планирования, прогнозирования и исправления нарушений

Таблица 1.4. Классические экспертные системы, применяемые в химии

Название	Назначение
CRYSALIS	Интерпретация трехмерной структуры белка
DENDRAL	Интерпретация молекулярной структуры
TQMSTUNE	Устранение нарушений в работе тройного квадрупольного масс-спектрометра (поддержка в настроенном состоянии)
CLONER	Проектирование биологических молекул нового типа
MOLGEN	Проектирование экспериментов по клонированию генов
SECS	Проектирование сложных органических молекул
SPEX	Планирование экспериментов в области молекулярной биологии

Таблица 1.5. Классические экспертные системы, применяемые в электронике

Название	Назначение
ACE	Диагностирование причин сбоев в телефонных сетях
IN-ATE	Диагностирование причин ошибок осциллографов
NDS	Диагностирование общегосударственной сети связи
EURISKO	Проектирование трехмерных микроэлектронных приборов
PALLADIO	Проектирование и проверка новых СБИС
REDESIGN	Перепроектирование цифровых схем на основании новых принципов

Окончание табл. 1.5

Название	Назначение
CADHELP	Подготовка инструктивных указаний для участников машинного проектирования
SOPHIE	Подготовка инструктивных указаний по диагностированию причин сбоев в электронных цепях

Таблица 1.6. Классические медицинские экспертные системы

Название	Назначение
PUFF	Диагностирование легочных заболеваний
VM	Текущий контроль над состоянием пациентов в палатах интенсивной терапии
ABEL	Диагностирование характеристик кислотных и щелочных электролитов
AI/COAG	Диагностирование заболеваний крови
AI/RHEUM	Диагностирование ревматических заболеваний
CADUCEUS	Диагностирование заболеваний внутренних органов
ANNA	Текущий контроль над терапевтическим лечением с применением препаратов наперстянки
BLUE BOX	Диагностирование и лечение депрессии
MYCIN	Диагностирование и лечение заболеваний, вызванных бактериальными инфекциями
ONCOCIN	Лечение и наблюдение над пациентами, проходящими курс химиотерапии
ATTENDING	Подготовка инструктивных указаний по проведению анестезирования
GUIDON	Подготовка инструктивных указаний по борьбе с бактериальными инфекциями

Таблица 1.7. Классические технические экспертные системы

Название	Назначение
REACTOR	Диагностирование и устранение сбоев в работе реактора
DELTA	Диагностирование и устранение неисправностей в локомотивах General Electric
STEAMER	Подготовка инструктивных указаний по эксплуатации паровой силовой установки

Таблица 1.8. Классические экспертные системы, применяемые в геологии

Название	Назначение
DIPMETER	Интерпретация данных журналов работы пластового наклономера
LITHO	Интерпретация данных журналов работы нефтяной скважины
MUD	Диагностирование и устранение проблем, возникающих в процессе бурения
PROSPECTOR	Интерпретация геологических данных на предмет обнаружения наличия полезных ископаемых

Таблица 1.9. Классические компьютерные экспертные системы

Название	Назначение
PTRANS	Выдача прогнозов, касающихся успешного управления компьютерами DEC
BDS	Диагностирование неисправных компонентов в коммутируемой сети
XCON	Настройка конфигурации компьютерных систем DEC
XSEL	Подготовка заказов на компьютеры DEC
XSITE	Определение требований к площадке заказчика, предназначенной для установке компьютеров DEC
YES/MVS	Текущий контроль и управление операционной системой MVS компании IBM
TIMM	Диагностирование компьютеров DEC

Наиболее подходящие области применения экспертных систем

Прежде чем приступать к созданию экспертной системы, очень важно определить, является ли приемлемым для решения рассматриваемой задачи подход, который основан на использовании экспертной системы. В частности, необходимо определить, должна ли экспертная система применяться вместо такого альтернативного подхода, как обычное программирование. Для выявления области применения, подходящей для использования экспертной системы, необходимо руководствоваться множеством факторов, перечисленных ниже, и найти ответы на указанные вопросы.

- Может ли задача быть успешно решена с помощью обычного программирования? Если ответ на этот вопрос является положительным, то экспертную систему нельзя назвать наилучшим выбором. Например, рассмотрим задачу диагностирования некоторого оборудования. Если все признаки всех

неисправностей известны заранее, то достаточно применить простой поиск неисправности в таблице или в дереве решений. Экспертные системы в наибольшей степени приспособлены для использования в таких ситуациях, в которых отсутствует приемлемое алгоритмическое решение. Подобные случаи называются **слабо структурированными задачами**, и применение рассуждений позволяет лишь надеяться на то, что будет найдено качественное решение.

В качестве примера слабо структурированной задачи рассмотрим случай, в котором отпускник не может определить, какую путевку следует заказать в оперативном режиме, и решает обратиться к агенту бюро путешествий. В табл. 1.10 подчеркнуты некоторые особенности **слабо структурированной задачи**, которые выражаются в том, как будущий отпускник отвечает на вопросы агента бюро путешествий.

Таблица 1.10. Пример слабо структурированной задачи

Вопросы агента бюро путешествий	Ответы
Могу ли я вам помочь?	Я не уверен
Куда вы хотите поехать?	Пока не знаю
Вы уже выбрали какое-либо конкретное место назначения?	Еще нет
На какую сумму вы рассчитываете?	Я не знаю
Вы успеете собрать деньги на поездку?	Я не знаю
Когда вы хотите отправиться в путешествие?	Еще не решил

Безусловно, это — крайний случай, но он иллюстрирует основные признаки слабо структурированной задачи. Вполне очевидно, что слабо структурированная задача как таковая не поддается алгоритмическому решению, поскольку количество вариантов в ней слишком велико. В данном случае после безуспешного завершения всех прочих попыток следует применить вариант, предусмотренный по умолчанию. Например, агент бюро путешествий может сказать: “Ах да! У меня есть для вас идеальное предложение — круиз вокруг света. Пожалуйста, заполните заявку на получение кредитной карточки, и мы обо всем позаботимся”.

В процессе решения слабо структурированных задач возникает такая проблема, что проект экспертной системы может непреднамеренно перерасти в алгоритмическое решение, а если имеется хороший алгоритм, то потребность в экспертной системе исчезает. Ключом к выявлению такой ситуации становится констатация того, что обнаружено решение, которое требует применения жесткой **структуры управления**. Иными словами, оказалось, что инженером по знаниям, явно задающим приоритеты многих правил, принудительно установлен порядок выполнения правил в определенной последовательности. Принудительное задание жесткой структуры управления в экспертной системе не позволяет использо-

вать важное преимущество технологии экспертных систем, предназначенной для обработки непредсказуемых входных данных, которые не следуют заранее определенному шаблону. Это означает, что экспертная система должна действовать, приспосабливаясь к любым входным данным, какими бы они ни были, а обычная программа, как правило, рассчитана на то, что входные данные будут поступать в определенной последовательности. Экспертная система, в которой применяется мощная структура управления, часто обнаруживает наличие в ней замаскированного алгоритма, поэтому может оказаться приемлемым кандидатом для перекодирования в виде обычной программы.

- Достаточно ли тщательно определены границы предметной области? Очень важно определить тщательно заданные пределы, в которых экспертная система должна обнаруживать свои знания и в которых должны проявляться ее возможности. Например, предположим, что требуется создать экспертную систему для диагностирования причин головной боли. Очевидно, что в базу знаний такой экспертной системы необходимо ввести медицинские знания врача-терапевта. Но для обеспечения глубокого понимания причин головной боли может также потребоваться ввести в базу знания о биохимии нервной системы, затем о той области биохимии, в рамках которой накапливаются знания о нервной системе человека, затем о химии, молекулярной биофизике и т.д., причем, возможно, даже о физике элементарных частиц. Знания, касающиеся причин головной боли, могут также присутствовать в других научных областях, таких как рефлексология, психология, психиатрия, физиология, наука о спорте, йога и самовнушение. Общий вывод из этих рассуждений таков: начав перечислять научные дисциплины, касающиеся любой темы, остановиться просто невозможно. А чем больше знаний из различных научных областей введено в экспертную систему, тем она сложнее.

При этом особенно затрудняется решение задачи координации всех экспертных знаний. Практический опыт показывает, насколько сложно бывает координировать работу групп экспертов разных специальностей, работающих над одними и теми же задачами, особенно если эти группы выдвигают противоречащие друг другу рекомендации. Если бы мы знали, как осуществить с помощью программы координацию экспертных знаний, то могли бы попытаться запрограммировать такую экспертную систему, в которой воплощены знания многих экспертов. Впервые попытки координировать работу многочисленных экспертных систем были предприняты при создании систем HEARSAY II и HEARSAY III. Такие проекты стали наглядной демонстрацией сложности указанной задачи. Например, многие из нас сталкивались с такой ситуацией, когда после доставки автомобиля в разные сервисные центры оказывалось, что в каждом случае дается диагноз, не совпадающий с другими. По мере того как уменьшается объем экспертных знаний,

увеличивается количество разных мнений, поэтому задача выбора правильного способа действий становится весьма затруднительной.

- Есть ли необходимость и потребность в создании экспертной системы? Безусловно, создание экспертной системы — очень увлекательное занятие, но оно теряет смысл, если никто не собирается ее использовать. Например, если на предприятии уже имеется много экспертов-людей, то трудно обосновать необходимость создания экспертной системы по причине нехватки экспертных знаний людей. Кроме того, если эксперты или пользователи не заинтересованы в применении системы, она так и не будет принята на вооружение, даже если есть такая потребность. В частности, многие расчетные модели показали, что управление светофорами с помощью методов искусственного интеллекта позволило бы снизить потребление бензина в крупном городе на 50%, но это также приведет к уменьшению на 50% объема налогов от продажи бензина, поступающих в казну города, штата и федерального правительства.

Особую заинтересованность в поддержке системы должно проявлять руководство. Это требование является еще более важным применительно к экспертным системам, чем к обычным программам, поскольку обычно считается, что развертывание экспертной системы предшествует сокращению количества рабочих мест. Работников необходимо убедить в том, что внедрение экспертной системы приводит не к сокращению работы, а к повышению прибыльности предприятия, поскольку экспертные знания становятся более широко доступными при меньших затратах. В целом научная область, связанная с созданием экспертных систем, заслуживает большей поддержки, поскольку в ней предпринимаются попытки решать такие задачи, которые не могут быть решены с помощью обычного программирования. Возникающие при этом риски гораздо выше, но больше и вознаграждение.

- Есть ли по крайней мере один эксперт-человек, желающий участвовать в этой работе? На предприятии должен быть хотя бы один эксперт, готовый принять участие в проекте, причем даже лучше, если он будет относиться к проекту создания экспертной системы с энтузиазмом. Далеко не все эксперты согласны подвергнуть свои знания проверке на наличие ошибок, а затем перенести их в компьютер. Причем, даже если многие эксперты желают участвовать в разработке, может оказаться целесообразным ограничить количество экспертов, участвующих в создании системы. Разные эксперты могут применять различные способы решения задачи, например, запрашивать разные диагностические тесты. Иногда эксперты могут даже приходить к несовпадающим заключениям. Попытка одновременно закодировать многочисленные методы решения задачи в одной базе знаний может привести к появлению внутренних конфликтов и несовместимостей.

- Может ли эксперт дать пояснения к формулируемым им знаниям таким образом, чтобы это было понятно для инженера по знаниям? Даже если эксперт готов сотрудничать, могут возникнуть трудности при попытке выразить знания в явных терминах. В качестве простого примера возникающих при этом сложностей предлагаем читателю объяснить словами, как происходит движение пальцем. Безусловно, можно сказать, что движение осуществляется путем сокращения мускулов пальца. В таком случае возникает вопрос о том, как происходит сокращение мускулов пальца, и т.д. Еще одна сложность, возникающая в процессе общения эксперта и инженера по знаниям, заключается в том, что инженер по знаниям может не владеть всей технической терминологией, которой пользуется эксперт. Эта проблема становится особенно острой, когда речь идет о медицинской терминологии. Можно потратить целый год или даже больше на то, чтобы инженер по знаниям просто научился понимать, о чем вообще говорит эксперт, не считая того, чтобы преобразовать знания эксперта в явно заданный компьютерный код.
- Являются ли знания в области решения задачи главным образом эвристическими и неопределенными? Экспертные системы становятся наиболее подходящими, если знания эксперта в основном заданы в эвристической и неопределенной форме. Это означает, что знания могут быть основаны на опыте (такие знания называются **практическими знаниями**), а эксперту может потребоваться опробовать другие методы в том случае, если выбранные им подходы не работают. Иными словами, знания эксперта могут предусматривать применение подхода по принципу проб и ошибок, а не подхода, основанного на логике и алгоритмах. Тем не менее эксперт все равно должен быть способен решать такую задачу быстрее по сравнению с теми, кто не является экспертом. Подобная область является хорошей сферой приложения для экспертных систем. Если бы задачу можно было просто решить с помощью логики и алгоритмов, то для ее решения было бы целесообразнее применить обычную программу.

1.8 Языки, командные интерпретаторы и инструментальные средства

Фундаментальным решением, которое должно быть принято при определении задачи, является решение о том, как лучше всего создать модель для этой задачи. Иногда бывает уже накоплен определенный опыт, позволяющий выбрать наилучший подход. Например, опыт показывает, что начисление заработной платы лучше выполнять с помощью обычного процедурного программирования. Кроме того, опыт показывает, что для решения этой задачи лучше всего использовать коммерческий программный пакет, если он имеется, а не создавать подобную программу

с нуля. В качестве общей рекомендации по выбору наиболее приемлемого подхода к решению задачи следует указать, что в первую очередь необходимо рассматривать наиболее традиционный способ — обычное программирование. Причина, по которой следует действовать таким образом, обусловлена тем, что накоплен огромный опыт использования обычного программирования и имеется чрезвычайно разнообразный набор доступных коммерческих пакетов. Если же задача не может быть эффективно решена с помощью обычного программирования, то следует обращаться к таким нетрадиционным подходам, как искусственный интеллект. Но для решения таких задач недостаточно просто пройти небольшой курс по языку C — необходимо изучать теорию.

Как и стандартный язык SQL, применяемый для работы с реляционными базами данных, язык экспертных систем является языком более высокого уровня по сравнению с такими языками третьего поколения, как LISP или C, поскольку он позволяет проще выполнять определенные операции; с другой стороны, язык экспертных систем может использоваться лишь для решения гораздо меньшего круга задач. Это означает, что в силу своего особого характера языки экспертных систем являются более подходящими для создания экспертных систем, а не для разработки программ общего назначения. Во многих ситуациях возникает даже необходимость выходить на время за рамки языка экспертной системы, чтобы выполнить какую-то функцию в процедурном языке. Язык CLIPS спроектирован таким образом, чтобы можно было легко выполнять подобные переходы.

Основное функциональное различие между языками экспертных систем и процедурными языками заключается в том, чтó лежит в основе представления. Процедурные языки нацелены на обеспечение поддержки гибких и надежных методов представления данных. Например, они позволяют легко создавать и манипулировать такими структурами данных, как массивы, записи, связные списки, стеки, очереди и деревья. Современные языки, такие как Java и C#, разработаны для предоставления программисту существенной помощи в достижении **абстракции данных**; для этого предусмотрены такие структуры, обеспечивающие инкапсуляцию данных, как объекты, методы и пакеты. Тем самым достигается необходимый уровень абстракции, который затем реализуется в программе. Данные и методы манипулирования данными тесно переплетаются в объектах. В отличие от этого, разработчики языков экспертных систем сосредоточивают свое внимание на решении задачи поддержки гибких и надежных способов представления знаний. Подход, основанный на использовании экспертных систем, допускает применение двух уровней абстракции — абстракция данных и **абстракция знаний**. В языках экспертных систем специально предусмотрено отделение данных от методов манипулирования данными. Примером подобного отделения является применение фактов (абстракции данных) и правил (абстракции знаний) в языке экспертной системы на основе правил. Кроме того, в языке CLIPS предусмотрено исполь-

зование объектов и всех прочих средств настоящего объектно-ориентированного языка.

Указанное различие в направленности между языками двух типов приводит также к различиям в методологии проектирования программ. Прежде всего, в процедурных языках данные и знания тесно переплетаются, поэтому программисты обязаны тщательно описывать последовательность выполнения операторов программы. С другой стороны, в языках экспертных систем данные явно отделены от знаний, поэтому требуется гораздо менее жесткий контроль над последовательностью выполнения кода программы. Как правило, для применения знаний к данным используется полностью отдельный компонент кода — машина логического вывода. Такое разделение знаний и данных способствует достижению более высокой степени распараллеливания и модульности.

Общепринятый способ определения потребности в разработке программы экспертной системы заключается в определении того, нужно ли представить в программе экспертные знания эксперта-человека. Если есть такой эксперт и готов к сотрудничеству, то подход, основанный на использовании экспертной системы, может оказаться успешным. Аналогичным образом с помощью инструментальных средств экспертных систем могут быть лучше всего решены задачи, требующие очень большого объема знаний и характеризующиеся наличием неопределенности.

На пути к выбору наиболее подходящих инструментальных средств экспертных систем приходится преодолевать много неосуществимых соблазнов, поскольку в наши дни имеется слишком много вариантов выбора таких средств. А что касается языка CLIPS, то он проще других языков в изучении и поэтому хорошо подходит для подготовки вводного учебника, даже несмотря на то, что не обладает абсолютно всеми возможностями других языков. Кроме того, язык CLIPS все еще сохраняет свои первоначальные преимущества, заключающиеся в том, что программы на этом языке имеют небольшие размеры и выполняются очень быстро; это особенно важно, если требуется отклик в реальном времени.

Не считая того, что в наши дни имеется такой выбор языков, что возникает путаница, источником путаницы становится и терминология, используемая для описания этих языков. Некоторые поставщики именуют свои программные продукты как “инструментальные средства”, тогда как другие называют их “командными интерпретаторами”, а остальные ведут речь об “интегрированных вариантах среды”. В настоящей книге для наибольшей ясности соответствующие термины определены, как указано ниже.

- **Язык**. Транслятор команд, записанных с применением конкретного синтаксиса. В языке экспертной системы предоставляется также машина логического вывода, предназначенная для выполнения операторов этого языка. В зависимости от реализации машина логического вывода может обеспечи-

вать прямой логический вывод, обратный логический вывод или оба варианта вывода. В соответствии с этим определением язык LISP не является языком экспертной системы, а PROLOG является таковым. Но существует возможность написать с помощью LISP язык экспертной системы и написать с использованием языка PROLOG систему искусственного интеллекта. Мало того, язык экспертной системы или искусственного интеллекта можно даже написать на языке ассемблера. Целесообразность использования того или иного языкового программного обеспечения зависит от таких показателей, как время разработки, удобство эксплуатации, удобство сопровождения, эффективность и быстродействие.

- **Инструментальное средство**. Сочетание языка и связанных с ним вспомогательных программ (или утилит), позволяющих упростить разработку, отладку и доставку пользователю прикладных программ. В состав утилит могут входить текстовые и графические редакторы, отладчики, системы управления файлами и даже генераторы кода. Некоторые инструментальные средства могут даже допускать использование в одном приложении разных подходов, таких как прямой и обратный логический вывод.

В некоторых случаях инструментальное средство может представлять собой комплекс, в котором все утилиты интегрированы в виде одной среды для предоставления пользователю общего интерфейса. При таком подходе сводится к минимуму необходимость для пользователя выходить за пределы среды в целях выполнения какой-либо задачи. Например, некоторое простое инструментальное средство может не предоставлять возможности управления файлами, поэтому пользователю может потребоваться выходить из этого инструментального средства, чтобы выдать обычные команды, допустим, на базовом языке C. А интегрированная среда обеспечивает удобный обмен данными между утилитами в одной и той же среде. Некоторые инструментальные средства даже не требуют от пользователя, чтобы он писал какой-либо код. Вместо этого инструментальное средство дает возможность пользователю вводить знания, представленные в виде примеров, из простых или электронных таблиц и само вырабатывает соответствующий код.

- **Командный интерпретатор**. Инструментальное средство специального назначения, применяемое в приложениях некоторых типов, требующих от пользователя предоставить только базу знаний. Классическим примером такого инструментального средства является командный интерпретатор EMYCIN, который был создан путем удаления медицинской базы знаний из экспертной системы MYCIN.

MYCIN была спроектирована как система обратного логического вывода для диагностирования заболеваний. В результате простого удаления медицинских знаний была создана система EMYCIN, предназначенная для использования в качестве командного интерпретатора, который может содержать знания, относящиеся

к консультативным системам других типов, в которых применяется обратный логический вывод. Командный интерпретатор EMYCIN продемонстрировал возможность повторного использования таких важных программных средств MYCIN, как машина логического вывода и пользовательский интерфейс. Создание EMYCIN явилось очень важным шагом в разработке современной технологии экспертных систем, поскольку ее появление означало, что нет необходимости создавать экспертную систему для каждого нового приложения с нуля. В настоящее время в области создания инструментальных средств экспертных систем появились весьма достойные конкуренты, обладающие многими возможностями и снабженные графическими интерфейсами пользователя.

Для описания характеристик экспертных систем применяется много способов. В частности, учитывается метод представления знаний, поддержка прямого или обратного логического вывода, поддержка неопределенности, возможность проведения гипотетических рассуждений, наличие средств объяснения и т.д. Способностью оценить по достоинству все эти особенности обладает только разработчик, создавший большое количество экспертных систем, особенно если речь идет о возможностях более дорогостоящих инструментальных средств. Наилучший способ изучения технологии экспертных систем состоит в том, чтобы разработать целый ряд систем с помощью какого-то языка, легкого в изучении, а затем направить свои усилия на освоение более сложного инструментального средства, если вам действительно нужны все его возможности.

1.9 Элементы экспертной системы

Элементы типичной экспертной системы показаны на рис. 1.6. В системе, основанной на правилах, знания в проблемной области, необходимые для решения задач, закодированы в форме правил и содержатся в базе знаний. Безусловно, для представления знаний наиболее широко применяются правила, но, как описано в главе 2, в экспертных системах других типов применяются также иные способы представления.

Экспертная система состоит из описанных ниже компонентов.

- **Пользовательский интерфейс**. Механизм, с помощью которого происходит общение пользователя и экспертной системы.
- **Средство объяснения**. Компонент, позволяющий объяснить пользователю ход рассуждений системы.
- **Рабочая память**. **Глобальная база** фактов, используемых в правилах.
- **Машина логического вывода**. Программный компонент, который обеспечивает формирование логического вывода (принимая решение о том, каким правилам удовлетворяют факты или объекты), располагает выполняемые правила по приоритетам и выполняет правило с наивысшим приоритетом.

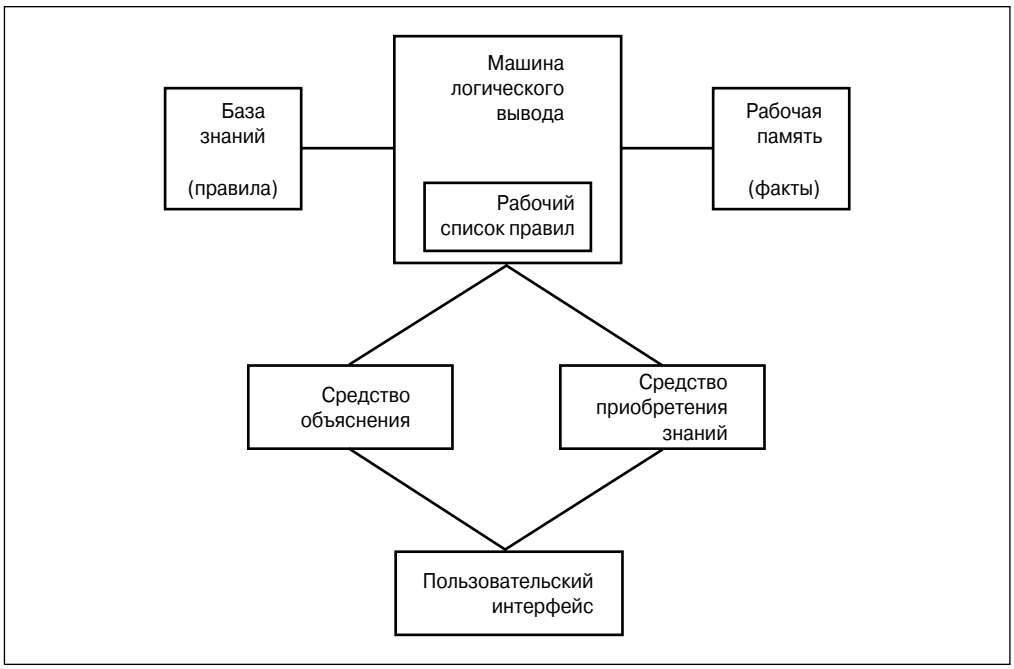


Рис. 1.6. Структура экспертной системы, основанной на правилах

- **Рабочий список правил**. Созданный машиной логического вывода и расположенный по приоритетам список правил, шаблоны которых удовлетворяют фактам или объектам, находящимся в рабочей памяти.
- **Средство приобретения знаний**. Автоматизированный способ, позволяющий пользователю вводить знания в систему, а не привлекать к решению задачи явного кодирования знаний инженера по знаниям.

Во многих системах имеется необязательное средство приобретения знаний. Это инструментальное средство в некоторых экспертных системах способно обучаться, осуществляя вывод правил по методу индукции на основании примеров, и автоматически вырабатывать правила. Для выработки правил в машинном обучении применялись также другие методы и алгоритмы, такие как ID3, C4.5, C5.1, искусственные нейронные сети и генетические алгоритмы. Основная проблема, возникающая при использовании машинного обучения для выработки правил, состоит в том, что отсутствует какое-либо объяснение, почему были созданы эти правила. В отличие от человека, способного объяснить причины, по которым было выбрано то или иное правило, системы машинного обучения никогда не были в состоянии объяснить свои действия, а это может повлечь за собой появление непредсказуемых результатов. Однако в целом для создания деревьев решений лучше всего подходят примеры, представленные в виде простых или электрон-

ных таблиц. Общие правила, подготовленные инженером по знаниям, могут быть намного сложнее по сравнению с простыми правилами, полученными путем вывода правил по методу индукции.

В зависимости от реализации системы в качестве пользовательского интерфейса может использоваться простой текстовый дисплей или сложный растровый дисплей с высокой разрешающей способностью. Дисплеи с высокой разрешающей способностью обычно применяются для моделирования панелей управления с циферблатами и окнами.

В экспертной системе, основанной на правилах, базу знаний называют также **продукционной памятью**. В качестве очень простого примера рассмотрим задачу принятия решения о переходе через дорогу. Ниже приведены продукции для двух правил, в которых стрелки означают, что система осуществит действия справа от стрелки, если условия слева от стрелки будут истинными.

```
горит красный свет → стоять
горит зеленый свет → двигаться
```

Продукционные правила могут быть выражены в эквивалентном формате псевдокода IF-THEN следующим образом:

```
Правило: red_light
IF
   горит красный свет
THEN
   стоять
Правило: green_light
IF
   горит зеленый свет
THEN
   двигаться
```

Каждое правило обозначается именем. Вслед за именем находится часть IF правила. Участок правила между частями IF и THEN правила упоминается под разными именами, такими как **антецедент**, **условная часть**, **часть шаблона** или **левая часть** (**left-hand-side — LHS**). Такое отдельно взятое условие, как

```
горит красный свет
```

называется **условным элементом**, или **шаблоном**.

Ниже приведены некоторые примеры правил из классических систем.

```
Система MYCIN для диагностирования менингита и бактеремий
   (бактериальных инфекций)
IF
   The site of the culture is blood, and
   The identity of the organism is not known with
```

```
   certainty, and
   The stain of the organism is gramneg, and
   The morphology of the organism is rod, and
   The patient has been seriously burned
THEN
   There is weakly suggestive evidence (.4) that
      the identity of the organism is pseudomonas
```

```
Система XCON/R1 для определения конфигурации компьютерных
   систем VAX компании DEC
IF
   The current context is assigning devices to
   Unibus modules and
   There is an unassigned dual-port disk drive and
   The type of controller it requires is known and
   There are two such controllers, neither of
      which has any devices assigned to it, and
   The number of devices that these controllers
      can support is known
THEN
   Assign the disk drive to each of the controllers, and
   Note that the two controllers have been
      associated and each supports one drive
```

В системе, основанной на правилах, машина логического вывода определяет, какие антецеденты правил (если таковые вообще имеются) выполняются согласно фактам. В качестве стратегий решения задач в экспертных системах обычно используются два общих метода логического вывода: **прямой логический вывод** и **обратный логический вывод**. В число других методов, применяемых для выполнения более конкретных методов, могут входить анализ целей и средств, упрощение задачи, перебор с возвратами, метод “запланировать–выработать–проверить”, иерархическое планирование и принцип наименьшего вклада, а также обработка ограничений.

Прямой логический вывод представляет собой метод формирования рассуждений от фактов к заключениям, которые следуют из этих фактов. Например, если перед выходом из дома вы обнаружите, что идет дождь (факт), то должны взять с собой зонтик (заключение).

Обратный логический вывод предусматривает формирование рассуждений в обратном направлении — от гипотезы (потенциального заключения, которое должно быть доказано) к фактам, которые подтверждают гипотезу. Например, если вы не выглядываете наружу, но кто-то вошел в дом с влажными ботинками и зонтиком, то можно принять гипотезу, что идет дождь. Чтобы подтвердить

эту гипотезу, достаточно спросить данного человека, идет ли дождь. В случае положительного ответа будет доказано, что гипотеза истинна, поэтому она становится фактом. Как уже было сказано выше, гипотеза может рассматриваться как факт, истинность которого вызывает сомнение и должна быть установлена. В таком случае гипотеза может интерпретироваться как цель, которая должна быть доказана.

В зависимости от проекта экспертной системы в машине логического вывода осуществляется либо прямой, либо обратный логический вывод, либо обе эти формы логического вывода. Например, язык CLIPS спроектирован в расчете на применение прямого логического вывода, в языке PROLOG осуществляется обратный логический вывод, а в версии CLIPS, называемой Eclipse, разработанной Полом Хэйли (Paul Haley), осуществляется и прямой, и обратный логический выводы. Выбор машины логического вывода зависит от типа задачи. Диагностические задачи лучше всего решать с помощью обратного логического вывода, в то время как задачи прогнозирования, текущего контроля и управления проще всего поддаются решению с помощью прямого логического вывода. Но применение более развитого инструментального средства связано с необходимостью увеличения объема памяти и влечет за собой снижение скорости выполнения, поскольку для создания такого инструментального средства требуется больший объем кода. Язык CLIPS был спроектирован с учетом минимальных потребностей в ресурсах, т.е. таким образом, чтобы программы на этом языке выполнялись как можно быстрее и существовала возможность развертывать приложения в устройствах памяти малой емкости, таких как ПЗУ. Безусловно, люди, применяющие настольные компьютеры, которые имеют ОЗУ емкостью 512 Мбайт, обычно над этим не задумываются, но при разработке такого изделия, как интеллектуальное устройство дистанционного управления или микроволновая печь, необходимо использовать в качестве наименее дорогостоящей постоянной памяти микросхему ПЗУ. Стоимость устройства памяти снижается по мере сокращения объема памяти, поэтому для обеспечения конкурентоспособности желательно применять в потребительских устройствах как можно меньший объем памяти. Язык CLIPS действительно позволяет вырабатывать исполняемые файлы в коде C, которые очень малы и могут быть легко записаны в небольшое ПЗУ.

Рабочая память может содержать факты, касающиеся текущего состояния светофора, такие как “горит зеленый свет” или “горит красный свет”. В рабочей памяти могут присутствовать любой из этих фактов или оба факта одновременно. Если светофор работает нормально, то в рабочей памяти будет находиться только один факт. Но возможно также, что в рабочей памяти будут присутствовать оба факта, если светофор неисправен. Обратите внимание на то, в чем состоит различие между базой знаний и рабочей памятью. Факты не взаимодействуют друг с другом. Факт “горит зеленый свет” не воздействует на факт “горит красный

свет". С другой стороны, наши знания о работе светофоров говорят о том, что если одновременно присутствуют оба факта, то светофор неисправен.

Если в рабочей памяти имеется факт "горит зеленый свет", машина логического вывода обнаруживает, что этот факт удовлетворяет условной части правила `green_light` и помещает это правило в рабочий список правил. А если правило имеет несколько шаблонов, то все эти шаблоны должны быть удовлетворены одновременно для того, чтобы правило можно было поместить в рабочий список правил. В качестве условия удовлетворения некоторых шаблонов можно даже указать отсутствие определенных фактов в рабочей памяти.

Правило, все шаблоны которого удовлетворены, называется **активизированным**, или **реализованным**. В рабочем списке правил может одновременно присутствовать несколько активизированных правил. В этом случае машина логического вывода должна выбрать одно из правил для **запуска**. Термину *запуск* в нейрофизиологии (наука о том, как работает нервная система) соответствует термин *возбуждение*. В результате стимуляции отдельная нервная клетка, или **нейрон**, испускает электрический сигнал. Дальнейшая стимуляция, насколько сильной она бы ни была, не может заставить нейрон снова возбудиться, пока не пройдет какой-то короткий период времени. Этот феномен называется **релаксацией**. Экспертные системы, основанные на правилах, создаются с использованием релаксации в целях предотвращения тривиальных циклов. Дело в том, что если бы правило, касающееся зеленого света, продолжало снова и снова запускаться под воздействием одного и того же факта, то экспертная система так и не смогла бы выполнить какую-либо полезную работу.

Для обеспечения релаксации разработаны различные методы. В языке экспертной системы одного из типов (OPS5) каждому факту после его ввода в рабочую память присваивается уникальный идентификатор, называемый **временно́й отметкой** (timetag). Вслед за тем, как некоторое правило запускается под воздействием некоторого факта, машина логического вывода не запускает его снова под воздействием того же факта, если с момента применения его временно́й отметки прошло слишком мало времени.

Вслед за частью `THEN` правила находится список **действий**, которые должны быть выполнены после запуска правила. Эта часть правила называется **консеквентом**, или **правой частью** (**Right-Hand Side — RHS**). Если происходит запуск правила `red_light`, выполняется его действие "стоять". Аналогичным образом после запуска правила `green_light` его действием становится "двигаться". В состав конкретных действий обычно входит добавление или удаление фактов из рабочей памяти либо вывод результатов. Формат описания этих действий зависит от синтаксиса языка экспертной системы. Например, в языке CLIPS действие по добавлению в рабочую память нового факта, называемого "`stop`" (стоять), принимает вид `(assert stop)`. Поскольку язык CLIPS происходит от языка LISP, то круглые скобки вокруг шаблонов и действий являются обязательными.

Машина логического вывода работает в режиме осуществления **циклов "распознавание–действие"**. Для описания указанного режима работы применяются также другие термины, такие как **цикл "выборка–выполнение"**, **цикл "ситуация–отклик"** и **цикл "ситуация–действие"**. Но как бы ни назывался такой цикл, машина логического вывода снова и снова выполняет некоторые группы задач до выявления определенных критериев, которые вызывают прекращение выполнения. При этом решаются общие задачи, обозначенные в приведенном ниже псевдокоде как **разрешение конфликтов**, **действие**, **согласование** и **проверка условий останова**.

```
WHILE работа не закончена
Разрешение конфликтов. Если имеются активизированные
правила, то выбрать правило с наивысшим приоритетом;
в противном случае работа закончена.
Действие. Последовательно осуществить действия,
указанные в правой части выбранного активизированного
правила. В данном цикле проявляется непосредственное
влияние тех действий, которые изменяют содержимое
рабочей памяти. Удалить из рабочего список правил
только что запущенное правило.
Согласование. Обновить рабочий список правил путем
проверки того, выполняется ли левая часть каких-либо
правил. В случае положительного ответа активизировать
соответствующие правила. Удалить активизированные
правила, если левая часть соответствующих правил больше
не выполняется.
Проверка условий останова. Если осуществлено
действие halt или дана команда break, то работа закончена.
END-WHILE
Принять новую команду пользователя
```

В течение каждого цикла могут быть активизированы и помещены в рабочий список правил многочисленные правила. Кроме того, в рабочем списке правил остаются результаты активизации правил от предыдущих циклов, если не происходит деактивизация этих правил в связи с тем, что их левые части больше не выполняются. Таким образом в ходе выполнения программы количество активизированных правил в рабочем списке правил изменяется. В зависимости от программы ранее активизированные правила могут всегда оставаться в рабочем списке правил, но никогда не выбираться для запуска. Аналогичным образом некоторые правила могут никогда не становиться активизированными. В подобных случаях следует повторно проверять назначение этих правил, поскольку либо такие правила не нужны, либо их шаблоны неправильно спроектированы.

Машина логического вывода выполняет действия активизированного правила с наивысшим приоритетом из рабочего списка правил, затем — действия активизированного правила со следующим по порядку приоритетом и т.д., до тех пор, пока в списке не останется больше активизированных правил. Для инструментальных средств экспертных систем разработаны различные системы приоритетов, но, вообще говоря, все инструментальные средства позволяют инженеру по знаниям определять приоритеты правил.

В рабочем списке правил возникают конфликты, если различные активизированные правила имеют одинаковый приоритет и машина логического вывода должна принять решение о том, какое из этих правил необходимо запустить. В различных командных интерпретаторах для решения этой проблемы применяются разные способы. Ньюэлл и Саймон использовали такой подход, что правила, введенные в систему в первую очередь, приобретают по умолчанию наивысший приоритет. С другой стороны, в языке OPS5 наивысший приоритет автоматически получают правила с более сложными шаблонами. А в языке CLIPS правила имеют по умолчанию одинаковый приоритет, если каким-то из них не присваивается другой приоритет инженером по знаниям.

После завершения выполнения всех правил управление возвращается к интерпретатору команд **верхнего уровня**, чтобы пользователь мог выдать командному интерпретатору экспертной системы дополнительные инструкции. Работа в режиме верхнего уровня соответствует применяемому по умолчанию режиму, в котором пользователь взаимодействует с экспертной системой, и обозначается как задача "`Accept a new user command`" (Принять новую команду пользователя). Прием новых команд происходит именно на верхнем уровне.

Верхний уровень представляет собой пользовательский интерфейс к командному интерпретатору в тот период, когда происходит разработка приложения экспертной системы. Но обычно разрабатываются более сложные пользовательские интерфейсы, позволяющие упростить работу с экспертной системой. Например, экспертная система может иметь пользовательский интерфейс для управления производственной установкой в виде экрана, на котором демонстрируется блок-схема этой установки. На этом экране вместе с условными изображениями приборов с круговыми и линейными шкалами могут отображаться мерцающими цветами предупреждающие сообщения и сообщения о состоянии. В действительности проектирование и реализация пользовательского интерфейса могут потребовать больше усилий, чем создание базы знаний экспертной системы, особенно на стадии разработки прототипа. В зависимости от возможностей командного интерпретатора экспертной системы пользовательский интерфейс может быть реализован с помощью правил или с применением операторов на другом языке, вызываемых из экспертной системы. Например, версия CLIPS на языке Java, называемая Jess, которая была создана Эрнестом Фридманом-Хиллом (Ernest Friedman-Hill) [30], позволяет легко вызывать классы Java для упрощенного создания графиче-

ского интерфейса пользователя, поскольку в языке Java имеется много объектов, предназначенных для этой цели.

Главной особенностью экспертной системы является предусмотренное в ней средство объяснения, которое дает возможность пользователю задавать вопросы о том, как система пришла к определенному заключению и для чего ей требуется определенная информация. Система, основанная на правилах, способна легко ответить на вопрос о том, как было получено определенное заключение, поскольку хронология активизации правил и содержимое рабочей памяти можно сохранять в стеке. Но такая возможность не столь легко достижима при использовании искусственных нейронных сетей, генетических алгоритмов или других систем, разработка которых еще продолжается. Безусловно, были сделаны попытки предусмотреть в некоторых системах возможность объяснения, но созданные при этом средства объяснения не могут сравниться по своей наглядности со средствами любой экспертной системы, спроектированной человеком. Развитые средства объяснения могут дать возможность пользователю задавать вопросы типа “что, если” и изучать альтернативные пути формирования рассуждений по принципу гипотетических рассуждений.

1.10 Продукционные системы

В настоящее время экспертными системами наиболее широко применяемого типа являются системы, основанные на правилах. В системах, основанных на правилах, знания представлены не с помощью относительно декларативного, статического способа (как ряд истинных утверждений), а в форме многочисленных правил, которые указывают, какие заключения должны быть сделаны или не сделаны в различных ситуациях. Система, основанная на правилах, состоит из правил `IF-THEN`, фактов и интерпретатора, который управляет тем, какое правило должно быть вызвано в зависимости от наличия фактов в рабочей памяти.

Системы, основанные на правилах, относятся к двум главным разновидностям: системы с прямым логическим выводом и системы с обратным логическим выводом. Система с прямым логическим выводом начинает свою работу с известных начальных фактов и продолжает работу, используя правила для вывода новых заключений или выполнения определенных действий. Система с обратным логическим выводом начинает свою работу с некоторой гипотезы, или цели, которую пользователь пытается доказать, и продолжает работу, отыскивая правила, которые позволят доказать истинность гипотезы. Для разбиения крупной задачи на мелкие фрагменты, которые можно будет более легко доказать, создаются новые подцели. Системы с прямым логическим выводом в основном являются управляемыми данными, а системы с обратным логическим выводом — управляемыми целями [21]. В одной из следующих глав дано более подробное описание

процесса логического вывода и приведен законченный пример того, как в языке CLIPS может быть осуществлен обратный логический вывод.

Широкое применение систем, основанных на правилах, обусловлено описанными ниже причинами.

- Модульная организация. Благодаря такой организации упрощается представление знаний и расширение экспертной системы по методу инкрементной разработки.
- Наличие средств объяснения. Такие экспертные системы позволяют легко создавать средства объяснения с помощью правил, поскольку антецеденты правила точно указывают, что необходимо для активизации правила. Средство объяснения позволяет следить за тем, запуск каких правил был осуществлен, поэтому дает возможность восстановить ход рассуждений, которые привели к определенному заключению.
- Наличие аналогии с познавательным процессом человека. Согласно результатам, полученным Ньюэллом и Саймоном, правила, по-видимому, представляют собой естественный способ моделирования процесса решения задач человеком. А при осуществлении попытки выявить знания, которыми обладают эксперты, проще объяснить экспертам структуру представления знаний, поскольку применяется простое представление правил `IF-THEN`.

Правила относятся к типу продукций, идея которых восходит к работам, выполненным в 1940-х годах. Поскольку системы, основанные на правилах, являются такими важными, необходимо ознакомиться с тем, как происходило развитие концепции правил. Это позволяет получить лучшее представление о том, почему системы, основанные на правилах, являются столь полезным средством создания экспертных систем.

Продукционные системы Поста

Продукционные системы были впервые использованы в символической логике Постом (Post), поэтому имя этого ученого вошло в название указанных систем. Пост доказал такой важный и неожиданный результат, что любая система математики или логики может быть оформлена в виде системы продукционных правил определенного типа. Этот результат показал огромные возможности применения продукционных правил для представления важных классов знаний, а это означает, что продукционные правила не сводятся к нескольким ограниченным типам. Кроме того, продукционные правила, обозначаемые термином **правила подстановки**, используются также в лингвистике как способ определения грамматики языка. Компьютерные языки обычно определяются с помощью формы продукционных правил, известной как нормальная форма Бэкуса–Наура (Backus-Naur Form — BNF); в качестве примера такого определения можно ознакомиться

с грамматикой языка CLIPS, приведенной в приложении Г. Основная идея Поста заключалась в том, что любая математическая или логическая система представляет собой набор правил, который указывает, как преобразовать одну строку символов в другой последовательный набор символов. Это означает, что продукционное правило после получения входной строки (антецедента) способно выработать новую строку (консеквент). Такая идея является также действительной по отношению к программам и экспертным системам, в которых начальная строка символов представляет собой входные данные, а выходная строка становится результатом определенных преобразований, которым были подвергнуты входные данные.

В качестве очень простого случая можно представить себе, что если входной строкой является "у пациента имеется высокая температура", то выходной строкой может быть "пациент должен принять аспирин". Обратите внимание на то, что за этими строками не закреплен какой-либо смысл. Иными словами, манипуляции со строками основаны на синтаксисе, а не на семантике, т.е. не на понимании того, что скрывается за словами "высокая температура", "аспирин" и "пациент". Люди знают, что означают эти строки в реальном мире, а продукционная система Поста применяется лишь в качестве способа преобразования одной строки в другую. Для данного примера может быть предусмотрено следующее продукционное правило:

```
антецедент → консеквент
у пациента имеется высокая температура →
   пациент должен принять аспирин
```

В этом правиле стрелка означает, что одна строка должна быть преобразована в другую. Указанное правило можно интерпретировать с помощью более знакомой системы обозначений IF-THEN следующим образом:

```
IF у пациента имеется высокая температура THEN пациент
должен принять аспирин
```

Продукционные правила также могут иметь несколько антецедентов, как в следующем примере:

```
у пациента имеется высокая температура AND температура
выше 38,9 градуса → обратиться к врачу
```

Обратите внимание на то, что специальная связка AND не входит в состав ни одной строки. Связка AND указывает, что данное правило имеет несколько антецедентов.

Продукционная система Поста состоит из группы продукционных правил, подобных приведенным ниже (числа, заданные в этих правилах в круглых скобках, упоминаются в дальнейшем обсуждении).

```
(1) двигатель автомобиля не запускается → проверить
    аккумулятор
```

```
(2) двигатель автомобиля не запускается → проверить
    наличие бензина
(3) проверить аккумулятор AND аккумулятор неисправен →
    заменить аккумулятор
(4) проверить наличие бензина AND бензин отсутствует →
    заполнить бак бензином
```

Если обнаруживается строка “двигатель автомобиля не запускается”, то могут использоваться правила (1) или (2) для выработки строк “неисправен аккумулятор” и “проверить аккумулятор”. Но механизм управления не предусматривает применения к строке одновременно обоих правил. Может быть применено только одно правило, оба правила последовательно или ни одного правила. Если есть еще одна строка “проверить аккумулятор”, а также строка “аккумулятор неисправен”, то может быть применено правило (3) для выработки строки “заменить аккумулятор”.

В отличие от обычного языка программирования, такого как C или C++, порядок, в котором записаны правила, не имеет никакого значения. Правила в рассматриваемом примере могут быть записаны в указанном ниже порядке, а система при этом остается неизменной.

```
(4) проверить наличие бензина AND бензин отсутствует →
    заполнить бак бензином
(2) двигатель автомобиля не запускается → проверить
    наличие бензина
(1) двигатель автомобиля не запускается → проверить
    аккумулятор
(3) проверить аккумулятор AND аккумулятор неисправен →
    заменить аккумулятор
```

Безусловно, продукционные правила Поста были необходимы для формирования определенной части фундамента экспертных систем, но они не были достаточными для написания практически применимых программ. Основным ограничением продукционных правил Поста с точки зрения программирования является отсутствие **стратегии управления**, которая позволяла бы регламентировать применение правил. Система Поста позволяет применять правила к строкам в любой форме, поскольку отсутствует какая-либо спецификация, определяющая то, как должны применяться те или иные правила.

В качестве аналогии предположим, что вы отправились в библиотеку, чтобы найти определенную книгу по экспертным системам. В библиотеке вы начинаете бессистемно просматривать книги на полках, пытаясь найти требуемую. Если библиотека довольно большая, то на поиск необходимой книги может потребоваться много времени. Даже если вы найдете отделение с книгами по экспертным системам, то следующая случайная попытка поиска может привести вас в совершенно

другое отделение, например, посвященное французской кулинарии. Ситуация становится еще хуже, если вам требуется материал из первой книги, чтобы вы могли определить, какую вторую книгу необходимо найти. Случайный поиск второй книги также отнимет много времени.

Марковские алгоритмы

Следующий крупный шаг в разработке методов применения продукционных правил был сделан на основании открытия, сделанного Марковым, которое позволило определить структуру управления для продукционных систем. **Марковский алгоритм** — это упорядоченная группа продукций, которые применяются в порядке приоритета к входной строке. Если правило с наивысшим приоритетом является неприменимым, то используется следующее правило в порядке приоритета, и т.д. Марковский алгоритм завершает свою работу после обнаружения одного из следующих условий: во-первых, последняя продукция не применима к строке или, во-вторых, применена продукция, которая заканчивается точкой.

Марковские алгоритмы могут также применяться к подстрокам строк, начиная слева. Например, продукционная система, состоящая из следующего единственного правила:

$$AB \rightarrow HIJ$$

после ее применения к входной строке $GABKAB$ вырабатывает новую строку $GHIJKAB$. Поскольку теперь продукция применяется к новой строке, окончательным результатом становится строка $GHIJKHIJ$.

Специальный символ $\wedge$ обозначает **пустую строку**, не содержащую символов. Например, следующая продукция удаляет все вхождения символа A в строке:

$$A \rightarrow \wedge$$

Другие специальные символы могут представлять любой отдельный символ и обозначаются строчными буквами a, b, c, ..., x, y, z. Эти символы представляют собой односимвольные переменные и являются важной частью современных языков экспертных систем. Например, следующее правило меняет местами символы A и B в строке, где x — любой единственный символ:

$$AxB \rightarrow BxA$$

Греческие буквы α, β и т.д. используются в строках в качестве специальных знаков пунктуации. Греческие буквы применяются потому, что их можно легко отличить от обычных букв алфавита.

Ниже приведен пример марковского алгоритма, который переводит первую букву входной строки в конец выходной строки. Правила упорядочены с учетом того, что правило (1) имеет высший приоритет, правило (2) — приоритет, который

следует за высшим приоритетом, и т.д. Приоритет правил задан в соответствии с порядком ввода правил, как показано ниже.

(1) $$\alpha xy \rightarrow y\alpha x$$

(2) $$\alpha \rightarrow \wedge$$

(3) $$\wedge \rightarrow \alpha$$

В табл. 1.11 показана трассировка выполнения алгоритма применительно к входной строке ABC.

Таблица 1.11. Трассировка выполнения марковского алгоритма

Правило	Успех (S) или неудача (F)	Строка
1	F	ABC
2	F	ABC
3	S	αABC
1	S	$B\alpha AC$
1	S	$BC\alpha A$
1	S	$BCA\alpha$
2	S	BCA

Обратите внимание на то, что символ α действует аналогично вре́менной переменной в обычном языке программирования. Но символ α не хранит значение, а используется как метка-заполнитель для обозначения места внесения изменений во входной строке. После выполнения задачи символ α удаляется с помощью правила 2. Скрытые марковские модели (Hidden Markov Model — HMM) широко используются в приложениях распознавания образов. К числу наиболее известных приложений такого типа относятся программы распознавания речи.

Rete-алгоритм

Обратите внимание на то, что марковские алгоритмы предусматривают применение вполне определенной стратегии управления, поскольку высокоприоритетные правила расположены по порядку на первых местах. При условии, что применимо правило с самым высоким приоритетом, используется это правило. В противном случае в марковском алгоритме предпринимаются попытки использовать правила с более низкими приоритетами. Безусловно, марковский алгоритм может применяться в качестве основы экспертной системы, но он является весьма неэффективным способом создания систем со многими правилами. Если требуется создать экспертную систему для решения реальных задач, содержащую сотни или тысячи правил, то проблема эффективности приобретает наибольшую

важность. Независимо от того, насколько приемлемыми являются все прочие характеристики системы, если пользователю придется долго ожидать ответа, то он не станет работать с такой системой. Поэтому фактически требуется алгоритм, который имеет полную информацию обо всех правилах и может применить любое нужное правило, не предпринимая попытки последовательно проверять каждое правило.

Решением этой проблемы является **rete-алгоритм**, разработанный Чарльзом Л. Форги (Charles L. Forgy) в университете Карнеги–Меллона в 1979 году в рамках диссертации по командному интерпретатору экспертной системы OPS (Official Production System — стандартная продукционная система) на получение степени доктора философии. Термин *rete-алгоритм* происходит от латинского слова rete (по-английски читается "рити", а по-русски — "рете"), которое означает сеть. Rete-алгоритм функционирует как сеть, предназначенная для хранения большого объема информации и обеспечивающая значительное сокращение времени отклика и повышение быстродействия при запуске правил по сравнению с большими группами правил `IF-THEN`, которые должны проверяться один за другим в обычной системе, основанной на правилах. Rete-алгоритм основан на использовании динамической структуры данных, подобной стандартному дереву B+, которая автоматически реорганизуется в целях оптимизации поиска.

Rete-алгоритм представляет собой очень быстрое средство сопоставления с шаблонами, высокое быстродействие которого достигается благодаря хранению в оперативной памяти информации о правилах, находящихся в сети. Этот алгоритм предназначен для повышения быстродействия систем с прямым логическим выводом, основанных на правилах, благодаря ограничению объема работы, требуемой для повторного вычисления конфликтного множества после запуска одного из правил. Недостатком этого алгоритма является его высокие потребности в памяти, но в наши дни, когда микросхемы памяти стали такими дешевыми, этот недостаток не имеет большого значения. В rete-алгоритме воплощены два описанных ниже эмпирических наблюдения, на основании которых была предложена структура данных, лежащая в его основе.

- **Временна́я избыточность**. Изменения, возникающие в результате запуска одного из правил, обычно затрагивают лишь несколько фактов, а каждое из этих изменений влияет только на несколько правил.
- **Структурное подобие**. Один и тот же шаблон часто обнаруживается в левой части больше чем одного правила.

Если в системе заданы сотни или тысячи правил, то подход к организации работы, в котором компьютер последовательно проверяет вероятность того, должен ли быть выполнен запуск каждого правила, становится очень неэффективным. Благодаря разработке rete-алгоритма появилась практическая возможность создания инструментальных средств экспертных систем даже на тех медленных

компьютерах, которые применялись в 1970-х годах. В наши дни rete-алгоритм продолжает оставаться важным средством повышения быстродействия в тех случаях, когда экспертная система содержит много правил.

В rete-алгоритме в каждом цикле контролируются только изменения в согласованиях, поэтому в каждом цикле "распознавание–действие" не приходится согласовывать факты с каждым правилом. Благодаря этому существенно повышается скорость согласования фактов с антецедентами, поскольку статические данные, которые не изменяются от цикла к циклу, могут быть проигнорированы. Эта тема будет обсуждаться более подробно в главах, посвященных языку CLIPS. В результате создания быстрых алгоритмов согласования с шаблонами, таких как rete-алгоритм, был полностью подготовлен фундамент для развертывания практических приложений экспертных систем. На рис. 1.7 приведены общие сведения о технологиях, которые образуют фундамент современных экспертных систем, основанных на правилах.

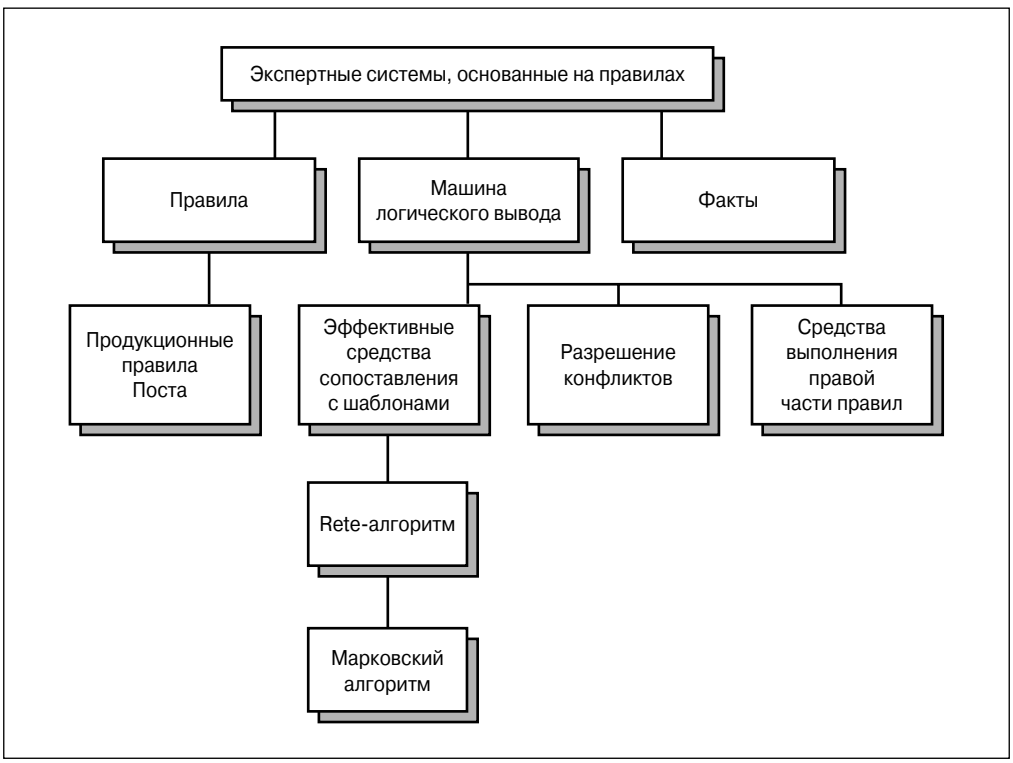


Рис. 1.7. Технологии, образующие фундамент современных экспертных систем, основанных на правилах

1.11 Процедурные подходы

Принципы программирования можно классифицировать как процедурные и непроцедурные. На рис. 1.8 показана **таксономия**, или классификация процедурных подходов, на примере различных языков, а на рис. 1.9 показана таксономия непроцедурных подходов. Эти рисунки иллюстрируют общую связь между экспертными системами и другими подходами к программированию, поэтому должны рассматриваться как общее руководство, а не как строгие определения. В частности, язык CLIPS обозначен как основанный на правилах, но на этом языке можно написать полностью объектно-ориентированную экспертную систему или гибридную систему, в которой используются и правила, и объекты. Некоторые подходы к программированию и языки имеют такие особенности, которые позволяют отнести их больше чем к одной категории. Например, одни ученые рассматривают функциональное программирование как пример процедурного подхода, а другие считают его декларативным.

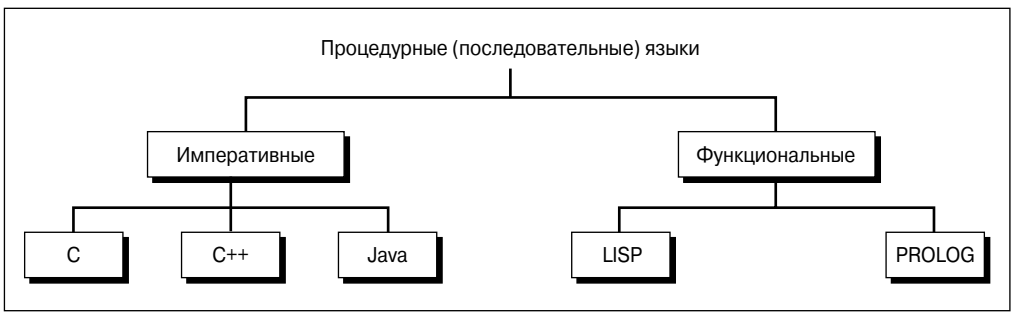


Рис. 1.8. Процедурные языки

Алгоритм представляет собой метод решения задачи за конечное количество шагов. Реализация алгоритма в виде программы называется **процедурной программой**. Термины *алгоритмическое программирование*, *процедурное программирование* и *обычное программирование* часто используются как синонимы для обозначения программ, не относящихся к искусственному интеллекту. В основе процедурной программы лежит такой общий принцип, что выполнение ее осуществляется последовательно, оператор за оператором, если не встречается команда перехода. Для обозначения процедурной программы часто используется еще один синоним — **последовательная программа**. Однако термин *последовательное программирование* подразумевает наличие слишком больших ограничений. Дело в том, что все современные языки программирования поддерживают рекурсию, поэтому программы могут оказаться не строго последовательными.

Отличительной особенностью процедурного подхода является то, что программист обязан точно указывать, как должно быть реализовано в программе

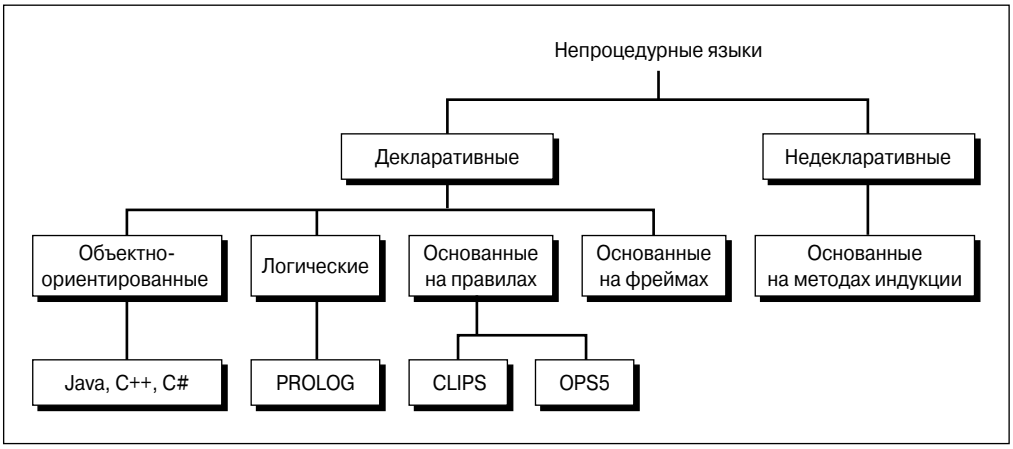


Рис. 1.9. Непроцедурные языки

решение задачи. Процедурный код приходится вырабатывать даже при использовании генераторов кода. Но в определенном смысле программирование с помощью генераторов кода можно считать **непроцедурным** программированием, поскольку генератор кода удаляет основную часть или даже весь процедурный код, написанный программистом. Цель непроцедурного программирования состоит в том, чтобы предоставить программисту возможность указывать, что должно быть сделано, и позволить системе решать, как это сделать.

Императивное программирование

Термины *императивный* и *операторный* используются как синонимы. Наиболее заметной отличительной особенностью такого языка, как C, является применение в нем операторов, которые представляют собой указания или команды компьютеру, сообщающие, что нужно сделать. Но следует отметить, что в объектно-ориентированной версии языка C, т.е. в языке C++, также могут использоваться объекты, а общий принцип работы программы состоит в том, что объекты передают сообщения друг другу и тем самым управляют ходом выполнения программы. Таким образом, выполнение объектно-ориентированной программы в большей степени напоминает работу системы, управляемой событиями, а не императивной системы, в основе которой лежит принцип, что операторы программы выполняются последовательно от начала до конца.

Императивный принцип программирования — это не что иное, как абстрактное представление действительно применяемых компьютеров, архитектура которых, в свою очередь, основана на машине Тьюринга и фон-неймановской машине с регистрами и хранилищем данных (памятью). В основе архитектуры этих машин лежит понятие модифицируемого хранилища данных. В языке программи-

рования аналогом модифицируемого **хранилища данных** являются переменные и результаты присваивания значений переменным, а хранилище данных представляет собой объект, которым управляет программа. В языках императивного программирования предусмотрен богатый набор команд, которые позволяют задавать структуру кода и управлять хранилищем данных. В каждом языке императивного программирования определено конкретное представление об аппаратных средствах. Эти представления настолько отличаются друг от друга, что принято вести речь, например, о виртуальной машине Pascal или виртуальной машине Java, которая выполняет байт-коды с учетом конкретной аппаратной платформы. Благодаря такой особенности виртуальных машин обеспечивается возможность переноса байт-кода без изменений с одной компьютерной платформы на другую. Успешное распространение языка Pascal в 1970–1980-х годах достигнуто в основном за счет переносимости байт-кода. В дальнейшем та же система была применена в языке Java. В действительности в языке программирования, поддерживаемом фактически применяемыми аппаратными средствами и операционной системой, виртуальную машину реализует компилятор, который определяется языком программирования.

В императивном программировании значение может быть обозначено именем, а в дальнейшем это имя может быть присвоено другому значению. Коллекция имен и связанных с ними значений, а также местонахождение в программе оператора, которому передано управление, представляет собой **состояние**. Состояние — это логическая модель памяти, представляющая собой ассоциацию между местонахождениями в памяти и значениями. Программу в ходе ее выполнения можно рассматривать как осуществляющую переход из начального состояния в конечное и при этом проходящую через последовательность промежуточных состояний. Переход от одного состояния в другое определяется операциями присваивания, командами ввода и командами определения хода выполнения.

Императивные языки были разработаны в целях освобождения программиста от необходимости составления кода на языке ассемблера в фон-неймановской архитектуре. Поэтому в императивных языках обеспечивается бóльшая поддержка переменных, операций присваивания и операций повторного выполнения. Все эти операции представляют собой операции низкого уровня, поэтому в современных языках предпринимается попытка их сокрытия путем предоставления таких средств, как рекурсия, процедуры, модули, пакеты и т.д. Императивные языки характеризуются также тем, что в них широко используется жесткая структура управления и в связи с этим реализуются **нисходящие** проекты программ.

Серьезная проблема, связанная с использованием всех языков, заключается в том, что задача доказательства правильности программы является очень сложной. Под понятием правильной программы подразумевается непротиворечивая программа (эта тема обсуждается более подробно в главе 2). С точки зрения искусственного интеллекта еще одной серьезной проблемой при использовании

императивных языков является то, что они не обеспечивают достаточно эффективного манипулирования символами. Дело в том, что архитектура императивных языков была модифицирована таким образом, чтобы они соответствовали фон-неймановской компьютерной архитектуре, поэтому по указанному принципу были созданы языки, которые позволяют очень хорошо осуществлять сложные математические расчеты, но не обеспечивают манипулирование символами. Однако для написания командных интерпретаторов экспертных систем в качестве базовых языков использовались императивные языки, такие как C, и объектно-ориентированные языки, подобные C++ или Java. Эти языки и созданные на их основе командные интерпретаторы работают более эффективно и быстро на компьютерах общего назначения по сравнению с первыми версиями командных интерпретаторов, созданных с использованием языка LISP, для которого требуется специальное аппаратное обеспечение, предназначенное исключительно для LISP.

Итак, императивные языки характеризуются тем, что создаваемые с их помощью программы являются последовательными, поэтому указанные языки не позволяют непосредственно реализовывать достаточно эффективные экспертные системы, особенно основанные на правилах. В качестве иллюстрации этой проблемы рассмотрим способ представления в виде кода информации о реальной задаче с сотнями или тысячами правил. Например, в базе знаний классической системы XCON, применяемой для определения конфигурации компьютерных систем, содержатся приблизительно 7000 правил. Вначале были предприняты безуспешные попытки написать такую программу на языке FORTRAN или BASIC, и только после этого был принят более подходящий подход, основанный на использовании экспертной системы. Такое инструментальное средство экспертной системы, как CLIPS, позволяет создать подобную крупную базу правил намного проще, чем при использовании программирования на языке третьего поколения или объектно-ориентированном языке, таком как Java, C++ или C#. Чтобы полностью оценить преимущества языка экспертной системы, необходимо прежде всего попытаться составить на нем программу. Программирование на языке CLIPS рассматривается далее в этой книге.

Непосредственный способ представления этих знаний на императивном языке мог бы предусматривать использование 7000 операторов `IF-THEN` или очень и очень большого оператора `CASE`. Такой стиль программирования приводит к появлению весьма серьезных проблем с точки зрения производительности, поскольку в каждом цикле “распознавание–действие” приходится выполнять поиск шаблонов, сопоставляемых с фактами, во всех 7000 правил. Следует отметить, что программное обеспечение машины логического вывода и применяемого в ней цикла “распознавание–действие” также приходится создавать на императивном языке. Но при этом ситуация становится намного сложнее, чем при оформлении в виде кода 7000 правил, поскольку многие правила активизируются в результате активизации других правил. Например, если речь идет о том, нужно ли останав-

ливаться, увидев красный свет на светофоре, то необходимо учитывать, что это правило действует, только если вы приближаетесь к светофору, а не после того, как вы его миновали. В системе, основанной на правилах, могут также возникать многие другие варианты взаимодействия. Некоторые правила могут предусматривать добавление фактов в базу знаний, другие — извлечение фактов, а третьи — модификацию фактов. А в случае машинного обучения могут применяться более сложные правила для создания новых правил или удаления существующих.

Эффективность программы можно было бы повысить, если бы существовала возможность упорядочить правила таким образом, чтобы правила, которые должны быть выполнены с наибольшей вероятностью, были перенесены в самое начало списка. Однако для осуществления такого подхода потребовался бы значительный объем работы по настройке системы, причем после добавления новых правил, а также удаления и модификации существующих правил снова требовалась бы настройка. Лучший метод повышения эффективности состоит в создании в памяти динамической сети шаблонов правил, позволяющей уменьшить время поиска для определения того, какие правила должны быть активизированы. К тому же было бы лучше не вынуждать программиста создавать такую древовидную сеть вручную, а формировать ее автоматически с помощью программы, учитывающей синтаксис шаблонов и действий в правилах `IF-THEN`. Было бы также удобно иметь синтаксис правил `IF-THEN`, который в большей степени способствовал бы успешному представлению знаний и допускал применение мощных средств сопоставления с шаблонами. Для этого необходимо разработать синтаксический анализатор, который бы анализировал структуру входных данных, а также интерпретатор или компилятор, позволяющий обрабатывать новые синтаксические конструкции правил `IF-THEN`.

Результатом применения всех указанных методов повышения эффективности становится специализированная экспертная система. На основе одной экспертной системы можно создавать другие экспертные системы, отделив от нее машину логического вывода, синтаксический анализатор и интерпретатор, т.е. компоненты, которые в целом принято называть *командным интерпретатором экспертной системы*. Но в наши дни, безусловно, гораздо проще воспользоваться такими существующими инструментальными средствами, как CLIPS, которые имеют качественную документацию и прошли серьезную проверку, а не выполнять всю указанную разработку с нуля.

Функциональное программирование

По своему характеру **функциональное программирование**, которое иллюстрируется на примере таких языков, как LISP, значительно отличается от программирования на основе операторных языков, поскольку в последних в основном применяются сложные управляющие структуры и нисходящее проектирование.

Решение задач при их разбиении на подзадачи намного упрощается. Но обычные языки программирования не позволяют применять достаточно сложные принципы такого разбиения. С другой стороны, в функциональных языках подобные ограничения во многом устранены. Разбивка программ на модули, соответствующие подзадачам, значительно облегчается в основном благодаря использованию таких двух средств функциональных языков, как функции высокого порядка и отложенное вычисление. В основе функционального программирования лежит фундаментальная идея объединения простых функций для создания более мощных функций. При этом по сути применяется метод **восходящего** проектирования, в отличие от обычных методов **нисходящего** проектирования, применяемых в императивных языках.

Функциональное программирование сосредоточено вокруг понятия **функции**. С точки зрения математики функция представляет собой **отображение**, или правило, по которому устанавливается соответствие между элементами одного множества (**области определения**) и другого множества (**области значений**). Пример определения функции приведен ниже.

$$\text{cube}(x) \equiv x * x * x,$$

где x — вещественное число, а cube — функция с вещественными значениями

Определение функции возведения в куб, cube, состоит из следующих трех частей:

1. отображение — $x * x * x$;
2. область определения — вещественные числа;
3. область значений — вещественные числа.

Символ $\equiv$ означает “является эквивалентным”, или “определен как”. Следующее обозначение представляет собой сокращенный способ записи утверждения, что функция возведения в куб (cube) представляет собой отображение из области определения, состоящей из вещественных чисел и обозначенной как $\Re$, в область значений, также состоящей из вещественных чисел:

$$\text{cube}{:}\Re \rightarrow \Re$$

Общим обозначением для функции f, которая выполняет отображение из области определения S в область значений T, служит $f : S \rightarrow T$. А **проекцией** функции f называется множество всех **образов элементов** $f(s)$, где s — элемент множества S. В случае функции возведения в куб образом элемента s является элемент $s * s * s$, а проекция представляет собой множество всех вещественных чисел. Применительно к функции возведения в куб проекция и область значений совпадают. Но это условие может не соблюдаться применительно к другим функциям, таким как функция возведения в квадрат $x * x$, областью определения и областью значений которой являются вещественные числа. Дело в том,

что проекцией по отношению к функции возведения в квадрат являются только неотрицательные вещественные числа, поэтому проекция и область значений не совпадают.

С использованием обозначения, применяемого для описания множества, проекции функции можно определить таким образом:

$$R \equiv \{f(s) \mid s \in S\}$$

Фигурные скобки ($\{\}$) обозначают **множество**, а **вертикальная черта** ($|$) читается "где". Приведенное выше выражение можно интерпретировать так, что проекция R эквивалентна множеству значений $f(s)$, где каждый элемент s принадлежит к множеству S. Отображение — это множество упорядоченных пар (s,t), где $s \in S$, $t \in T$ и $t = f(s)$. Каждому элементу в множестве S должен соответствовать один и только один элемент множества T. Но одному элементу s может соответствовать несколько значений t. В качестве простого примера можно указать, что каждое положительное число n имеет два квадратных корня, $\pm\sqrt{n}$.

Функции могут быть также определены рекурсивно, как в следующем примере:

$$\text{factorial}(n) \equiv n * \text{factorial}(n-1),$$

где n — целое число и factorial — целочисленная функция

Рекурсивные функции очень широко применяются в таких функциональных языках, как LISP.

Математические понятия и выражения являются **ссылочно прозрачными**, поскольку значение всего выражения полностью определяется его частями, а между частями отсутствует какое-либо скрытое взаимодействие. Например, рассмотрим функциональное выражение $x + (2 * x)$. Очевидно, что его результатом является $3*x$. Это означает, что оба выражения, $x+(2*x)$ и $3*x$, дают одинаковый результат, независимо от того, какие значения будут подставлены вместо x. В качестве x можно даже подставить другие функции, и результат сравнения двух выражений останется неизменным. Допустим, что $h(y)$ — некоторая произвольная функция. И в таком случае выражение $h(y) + (2 * h(y))$ будет по-прежнему эквивалентно выражению $3 * h(y)$.

Теперь рассмотрим следующий оператор присваивания в таком императивном языке программирования, как C:

$$\text{sum} = f(x) + x$$

Если параметр x передается по ссылке и его значение изменяется в вызове функции $f(x)$, то какое значение будет использоваться для второго вхождения переменной x в это выражение? В зависимости от того, как написан компилятор, x может иметь первоначальное значение, если этот элемент выражения сохраняется

в стеке, или новое значение, если x не сохраняется. Еще один источник путаницы возникает, если один компилятор вычисляет выражения справа налево, а другой — слева направо. В таком случае при использовании разных компиляторов результат вычисления выражения $f(x) + x$ не будет равен результату вычисления выражения $x + f(x)$, даже если применяется один и тот же язык. Вследствие применения глобальных переменных могут возникать и другие побочные эффекты. Поэтому программные функции, в отличие от математических функций, не являются ссылочно прозрачными.

Функциональные языки программирования создавались с учетом обеспечения требования к ссылочной прозрачности. Функциональный язык состоит из следующих пяти частей:

- **объекты данных**, с которыми должны оперировать языковые функции;
- **примитивные функции**, которые должны применяться к объектам данных;
- **функциональные формы**, предназначенные для создания новых функций на основе существующих функций;
- **прикладные операции** на функциях, которые возвращают значения;
- **процедуры именования**, предназначенные для присваивания имен новым функциям.

Функциональные языки, как правило, реализуются с помощью интерпретаторов для обеспечения простоты конструирования и быстрого отклика на ввод команд пользователем.

В языке LISP (сокращение от LISt Processing — обработка списков) объекты данных представляют собой **символические выражения** (S-выражения), которые являются либо **списками**, либо **атомами**. Ниже приведены примеры списков, которые демонстрируются в связи с тем, что стиль их представления аналогичен тому стилю, с помощью которого в языке CLIPS программируются шаблоны.

```
(milk eggs cheese)
(shopping (groceries (milk eggs cheese) clothes (pants)))
()
```

Шаблоны, подобные этим, могут задаваться в программе в условном выражении либо в **левой части** (**Left-Hand-Side — LHS**) правила и представлять факты или вложенные списки, как в примере списка "`shopping`". Если условное выражение имеет истинное значение, то выполняется **правая часть** (**Right-Hand-Side — RHS**) правила и создаются новые списки. В программе может быть предусмотрено выполнение в правой части правила других действий, таких как извлечение фактов. Списки всегда заключены в парные круглые скобки, а элементы списков разделены пробелами. Элементами списков могут быть атомы, такие как `milk`, `eggs` и `cheese`, или вложенные списки, такие как `(milk eggs cheese)` и `(pants)`. Списки можно разбивать на элементы, а атомы не под-

лежат разбиению. **Пустой список**, `()`, не содержит элементов и обозначается как **nil**.

Первоначальная версия языка LISP именовалась чистым языком LISP, поскольку была чисто функциональной. Но эта версия не была достаточно эффективным средством написания программ. Поэтому в LISP были введены нефункциональные дополнения в целях повышения эффективности написания программ. В качестве примера можно назвать оператор `SET`, который действует как оператор присваивания, а операторы `LET` и `PROG` могут использоваться для создания локальных переменных и выполнения последовательности S-выражений. Безусловно, эти операторы действуют аналогично функциям, но не являются функциональными в строго математическом смысле.

По истечении определенного времени после его создания LISP стал ведущим языком искусственного интеллекта в Соединенных Штатах. На языке LISP было написано много оригинальных командных интерпретаторов экспертных систем, поскольку с помощью LISP можно очень легко проводить эксперименты по созданию программ. Но обычные компьютеры недостаточно эффективно выполняют программы LISP, а эксплуатация командных интерпретаторов, созданных с помощью LISP, осуществляется еще менее эффективно. Безусловно, по мере усовершенствования процессоров и повышения их тактовой частоты возрастает и производительность программ LISP.

Таким образом, с внедрением программ LISP связана проблема значительных издержек, которая оказала свое влияние на разработку таких программ и стала причиной возникновения так называемой **проблемы доставки**. Дело в том, что недостаточно лишь разработать великолепную программу, если ее невозможно *доставить* пользователю для последующей эксплуатации из-за слишком больших издержек. Даже если разработка ведется с помощью превосходной рабочей станции, компьютер с аналогичными характеристиками не всегда становится приемлемым средством доставки программы пользователю в связи с наличием ограничений по быстродействию, мощности, размерам, весу, стойкости к воздействию окружающей среды или стоимости. Для некоторых приложений может даже потребоваться, чтобы окончательный вариант кода был помещен в ПЗУ в целях уменьшения стоимости и обеспечения энергонезависимости. Но при использовании определенных инструментальных средств искусственного интеллекта и экспертных систем, для эксплуатации которых требуются специальные аппаратные средства, возможность помещения кода в ПЗУ может стать проблематичной. Поэтому лучше обеспечить такую возможность заранее, чем в дальнейшем сталкиваться с необходимостью повторной разработки кода программы.

Еще одна проблема состоит в том, как обеспечить при создании интеллектуальных систем сопряжение языков искусственного интеллекта с такими обычными языками программирования, как C, C++, C# и Java. Приложения, в которых должен осуществляться большой объем сложных математических расчетов,

лучше всего разрабатывать с помощью обычных языков, а не инструментальных средств экспертных систем. Именно поэтому в языке CLIPS предусмотрена возможность легко создавать собственные функции на базовом языке. С дополнительными сведениями по этим и другим сложным темам можно ознакомиться в оперативном режиме с помощью справочного руководства *CLIPS Reference Manual* (`http://www.ghgcorp.com/clips/CLIPS.html`).

Обычно экспертные системы, написанные на языке LISP, сложно внедрить в программы на языках, отличных от LISP, если для этого не предусмотрены специальные конструкции. Поэтому основным требованием при выборе языка искусственного интеллекта является анализ возможности применения того языка, на котором написано само инструментальное средство. В настоящее время исходя из соображений переносимости, эффективности и быстродействия код многих инструментальных средств экспертных систем разрабатывается на языке C или преобразуется в код на языке C. Благодаря этому устраняется также проблема, связанная с тем, что для приложений, основанных на языке LISP, требуются дорогостоящие специальные аппаратные средства.

1.12 Непроцедурные подходы

Суть **непроцедурных подходов** состоит в том, что они не требуют от программиста описания точных подробностей того, как должна быть решена задача. В этом непроцедурные подходы коренным образом отличаются от процедурных подходов, которые требуют указания того, как должна вычисляться последовательность функций или операторов. При использовании непроцедурных подходов основной объем работы заключается в том, чтобы указать, *что* должно быть сделано, и позволить системе определить, *как* это сделать.

Декларативное программирование

В **декларативном подходе** путь достижения **цели** рассматривается отдельно от методов, используемых для достижения этой цели. Пользователь задает цель, а основополагающий механизм реализации предпринимает попытку достичь данной цели или, как принято говорить, *выполнить эту цель*. Для реализации такой декларативной модели было создано множество подходов и связанных с ними языков программирования.

Объектно-ориентированное программирование

Еще одним подходом, который может отчасти рассматриваться как императивный и отчасти как декларативный, является **объектно-ориентированный** подход. В настоящее время термин *объектно-ориентированный* используется для обозна-

чения таких языков программирования, как C++, Java и C#. Основная идея этого подхода состоит в том, что программа разрабатывается по принципу представления данных, используемых в программе, в виде объектов, с последующей реализацией операций над этими объектами. Такой способ разработки программы является противоположным по отношению к способу, применяемому при нисходящем проектировании, который осуществляется путем поэтапного уточнения управляющей структуры программы. Один из широко применяемых способов объектно-ориентированного проектирования основан на языке UML (Unified Modeling Language — универсальный язык моделирования).

В качестве примера объектно-ориентированного проекта рассмотрим задачу написания программы для управления расчетным счетом с помощью интерактивного меню. Наиболее важными объектами данных являются текущий остаток, суммы списания и зачисления. После определения соответствующих классов могут быть также определены различные методы, предназначенные для осуществления операций с объектами данных. В состав таких операций могут входить зачисление, списание, а также начисление месячных процентов. После определения всех объектов данных, операций и интерфейса меню можно приступать к составлению программы. Такая методология объектно-ориентированного проектирования хорошо приспособлена для создания программ, имеющих слабую управляющую структуру. С другой стороны, подобная методология может оказаться неприменимой при создании программ, имеющих сильную управляющую структуру, таких как приложения для учета заработной платы. Но в настоящее время объектно-ориентированное программирование применяется весьма широко, поскольку позволяет снизить затраты на сопровождение и обеспечить повторное использование кода объектов.

Термин **объектно-ориентированное программирование** первоначально использовался применительно к таким языкам, как Smalltalk, которые были специально предназначены для работы с объектами. В настоящее время этот термин иногда применяется для обозначения способа разработки объектно-ориентированного проекта даже на языке, не имеющем подлинной объектной поддержки.

Такие современные объектно-ориентированные языки, как C++, Java и C#, обладают средствами поддержки объектов, встроенными непосредственно в сам язык. Язык Smalltalk происходит от языка SIMULA 67, разработанного для моделирования (simulation, отсюда название языка). В языке SIMULA 67 реализовано понятие **класса**, которое легло в основу понятия *сокрытия информации*; реализация этого понятия позволяет достичь такой ситуации, в которой программисту требуется знать лишь то, как используется объект, и не учитывать подробности внутреннего устройства объекта и того, как он запрограммирован. Класс — это не тип. **Экземпляром** класса является объект данных, с которым могут проводиться необходимые манипуляции. Термин *экземпляр* был перенесен на экспертные системы, в которых он означает факт, согласующийся с шаблоном. Аналогичным

образом в проблематике экспертных систем принято применять формулировку, что использован экземпляр правила, в тех случаях, когда выполняется левая часть этого правила. В системах, основанных на правилах, термины *активизированный* и *используемый в качестве экземпляра* рассматриваются как синонимы.

Еще одним важным понятием, заимствованным из языка SIMULA 67, является **наследование**. С помощью наследования может быть определен **подкласс**, который наследует свойства одного или нескольких классов. Например, может быть определен один класс, состоящий из объектов, которые используются в стеке, и другой класс, предназначенный для применения в качестве класса комплексных чисел. После этого может быть легко определен подкласс, состоящий из объектов, которые представляют собой комплексные числа, используемые в стеке. Таким образом, эти объекты имеют свойства, унаследованные от вышестоящих классов, называемых **суперклассами**. Понятие наследования может быть расширено для организации объектов в виде иерархии, в которой объекты могут наследовать свойства от своих классов, а эти классы, в свою очередь, могут наследовать свойства от вышестоящих классов, и т.д. Принцип наследования является очень полезным, поскольку позволяет создавать объекты, которые наследуют свойства от своих классов; при этом программист не обязан повторно задавать каждое свойство, наследуемое объектом. Но задача обеспечения множественного наследования является более сложной, поэтому такая возможность в языках Java и C# не предусмотрена. Но язык CLIPS создан на основе языка C++ и в связи с этим позволяет воспользоваться преимуществами множественного наследования. В язык CLIPS встроен полноценный объектно-ориентированный язык, называемый **COOL**. Внутреннее функционирование языка COOL осуществляется незаметно для пользователя, поэтому пользователю достаточно лишь понимать суть объектно-ориентированного программирования.

Логическое программирование

Одним из первых приложений искусственного интеллекта, в которых использовались компьютеры, было доказательство логических теорем с помощью программы Logic Theorist (Логик-теоретик) Ньюэлла (Newell) и Саймона (Simon). Впервые отчет о создании этой программы был опубликован на Дартмутской конференции по искусственному интеллекту в 1956 году. Данный отчет вызвал сенсацию, поскольку до этого компьютеры применялись только для числовых расчетов. А в данном случае компьютер механически проводил рассуждения, необходимые для доказательства математических теорем, тогда как до сих пор считалось, что эту задачу способны выполнять только математики. Термин *механический* означает *автоматический*; впервые этот термин был применен для обозначения механического вычислительного устройства, предложенного Бэббиджем (Babbage) в XIX столетии.

При разработке программы Logic Theorist и ее преемника, программы GPS (General Problem Solver — универсальный решатель задач), Ньюэлл и Саймон сосредоточились на осуществлении попытки реализовать мощные алгоритмы, позволяющие решать любые задачи. Программа Logic Theorist была предназначена только для доказательства математических теорем, а программа GPS разрабатывалась для решения логических задач любого типа, включая игры и головоломки, такие как шахматы, задача с ханойской башней, задача с миссионерами и каннибалами и криптоарифметические задачи. Кроме того, ниже приведен пример известной криптоарифметической головоломки (числового ребуса), которая рассматривалась Ньюэллом и Саймоном. По условию этой головоломки известно, что $D = 5$.

$$\begin{array}{rr} & \text{DONALD} \\ + & \underline{\text{GERALD}} \\ & \text{ROBERT} \end{array}$$

Задача состоит в том, чтобы подобрать вместо букв разные цифры от 0 до 9 и преобразовать этот ребус в правильный пример сложения чисел.

Программа GPS оказалась первой программой решения задач, в которой знания в области решения задач были явно отделены от знаний в проблемной области. Теперь такой подход, предусматривающий явное отделение знаний в области решения задач от знаний в проблемной области, используется в качестве основы экспертных систем. В современных экспертных системах решения о том, какие знания должны использоваться и как их следует применить, принимает машина логического вывода.

Усилия в направлении усовершенствования автоматического доказательства теорем продолжались и в дальнейшем. К началу 1970-х годов было обнаружено, что вычисление представляет собой частный случай механической, логической дедукции. После того как к высказываниям в форме "заключение `if` условия" был применен обратный логический вывод, обнаружилось, что он является достаточно мощным для доказательства весьма сложных теорем. В этой форме условия могут рассматриваться как представляющие шаблоны, с которыми должно быть найдено соответствие по такому же принципу, как в продукционных правилах, рассматривавшихся ранее. Высказывания, представленные в такой форме, принято называть **хорновскими выражениями** в честь Альфреда Хорна (Alfred Horn), который впервые исследовал их. В 1972 году Ковальский (Kowalski), Колмероэ (Colmerauer) и Руссел (Roussell) создали язык PROLOG для реализации принципа логического программирования на основе обратного логического вывода с использованием хорновских выражений.

Обратный логический вывод может применяться не только для выражения знаний с помощью декларативного представления, но и для управления процессом формирования рассуждений. Как правило, обратный логический вывод

осуществляется путем определения меньших **подцелей**, которые должны быть удовлетворены для того, чтобы могла быть удовлетворена первоначальная цель. Затем эти подцели могут дальше подвергаться разбиению на меньшие подцели и т.д. В качестве примера декларативных знаний можно привести следующий классический пример:

```
Все люди смертны
Сократ – человек
```

Это высказывание может быть представлено с помощью хорновских выражений таким образом:

```
некто смертен
   IF некто – человек
Сократ – человек
   IF (во всех случаях)
```

В данном высказывании, касающемся Сократа, условие `IF` является истинным во всех случаях. Иными словами, для использования знаний о Сократе не требуется согласования фактов с какими-либо шаблонами. Сравните это с таким случаем, где речь идет о смертных, и некто должен быть человеком, для того чтобы был удовлетворен шаблон условия `IF`.

Обратите внимание на то, что хорновское выражение может интерпретироваться как процедура, которая указывает, как должна быть удовлетворена цель. Таким образом, для определения того, что некто является смертным, необходимо определить, является ли хоть кто-либо человеком. Ниже приведен немного более сложный пример.

```
Условиями нормальной эксплуатации автомобиля является
наличие бензина, масла и надутых шин
```

Этот пример может быть представлен с помощью хорновского выражения следующим образом:

```
x является исправным автомобилем и может нормально
   эксплуатироваться
   IF x снабжен бензином и
   IF x снабжен маслом и
   IF x имеет надутые шины
```

Здесь заслуживает внимания то, что проблема определения, пригоден ли автомобиль к нормальной эксплуатации, сведена к трем более простым подпроблемам, или подцелям. А теперь предположим, что имеются некоторые дополнительные декларативные знания наподобие следующих:

```
Бензиномер показывает ненулевое значение
   IF автомобиль снабжен бензином
Указатель уровня масла показывает ненулевое значение
```

```
   IF автомобиль снабжен маслом
Воздушный манометр показывает по меньшей мере 20
   IF автомобиль имеет надутые шины
Бензиномер показывает ненулевое значение
Указатель уровня масла показывает нулевое значение
Воздушный манометр показывает 15
```

Эти знания могут быть представлены с помощью таких хорновских выражений:

```
x снабжен бензином
   IF бензиномер показывает ненулевое значение
x снабжен маслом
   IF указатель уровня масла показывает ненулевое значение
x имеет надутые шины
   IF воздушный манометр показывает по меньшей мере 20
Бензиномер показывает ненулевое значение
   IF (во всех случаях)
Указатель уровня масла показывает нулевое значение
Воздушный манометр показывает 15
   IF (во всех случаях)
```

На основании этих выражений программа автоматического доказательства теорем может доказать, что автомобиль непригоден к нормальной эксплуатации, поскольку в нем нет масла, а давление воздуха является недостаточным.

Одним из преимуществ систем обратного логического вывода является то, что выполнение заложенных в них программ может осуществляться параллельно. Это означает, что при наличии многочисленных процессоров они могли бы одновременно работать над выполнением подцелей. В языке PROLOG предусмотрена машина обратного логического вывода, поэтому он представляет собой больше чем просто язык. Как минимум PROLOG можно рассматривать как командный интерпретатор, поскольку он требует применения следующих компонентов:

- интерпретатор или машина логического вывода;
- база данных (факты и правила);
- определенная форма сопоставления с шаблонами, называемая **унификацией**;
- механизм **перебора с возвратами**, позволяющий переходить к исследованию альтернативных подцелей, если текущая попытка поиска, предпринятая для выполнения некоторой цели, оказалась неудачной.

В качестве примера обратного логического вывода предположим, что вы можете купить масло для того, чтобы обеспечить нормальную эксплуатацию своего автомобиля, если у вас есть наличные или кредитная карточка. В данном случае

одной из подцелей становится проверка того, есть ли у вас наличные. Если тот факт, что у вас есть наличные, не обнаруживается, то механизм перебора с возвратами может затем перейти к исследованию другой подцели для определения того, если ли у вас кредитная карточка. Если кредитная карточка имеется, то цель покупки масла может быть достигнута. Следует отметить, что отсутствие некоторого факта, требуемого для доказательства целевого утверждения, приводит к такому же действенному дальнейшему развитию логического вывода (хотя, возможно, менее эффективному), чем наличие отрицательного факта, такого как "Указатель уровня масла показывает нулевое значение". К недостижению цели могут приводить и отрицательные, и отсутствующие факты (очевидно, что в политике часто наблюдается другая логика).

Если по условиям задачи механизмы перебора с возвратами и сопоставления с шаблонами не требуются, то программист должен исключить возможность их использования или разработать код на другом языке. Одним из преимуществ логического программирования является возможность применения исполняемых спецификаций. Это означает, что для создания исполняемой программы достаточно представить требования к решению задачи в виде хорновских выражений. В этом состоит весьма существенное отличие от обычного программирования, в котором документ с описанием требований отнюдь не напоминает окончательный исполняемый код.

В отличие от систем, основанных на применении продукционных правил, в языке PROLOG порядок, в котором заданы подцели, факты и правила, существенно влияет на работу этой программы. С другой стороны, способ поиска данных в базе данных, применяемый в программе PROLOG, влияет на эффективность программы и поэтому на ее быстродействие. Более того, в некоторых случаях программы выполняются правильно, если подцели, факты и правила введены в одном порядке, но входят в бесконечный цикл или вырабатывают ошибки этапа прогона, если этот порядок становится другим.

Экспертные системы

Экспертные системы могут рассматриваться как направление **декларативного программирования**, поскольку программист не указывает на уровне алгоритма, как программа должна достичь заданной цели. Например, в экспертной системе, основанной на правилах, любое из правил может стать активизированным и быть помещено в рабочий список правил, если левая часть этого правила согласуется с фактами. Порядок, в котором были введены правила, не влияет на то, какие правила должны быть активизированы. Таким образом, порядок операторов в программе не задает жесткий порядок управления ходом выполнения. Экспертные системы других типов основаны на **фреймах**, как описано в главе 2, и **сетях логического вывода**, которые рассматриваются в главе 4.

Между экспертными системами и обычными программами имеется целый ряд различий. Некоторые из этих различий перечислены в табл. 1.12.

Таблица 1.12. Некоторые различия между обычными программами и экспертными системами

Характеристика	Обычная программа	Экспертная система
Способ управления ходом выполнения	С учетом порядка операторов	С использованием машины логического вывода
Средства управления и данные	Неявная интеграция	Явное разделение
Детерминированность управления	Сильная	Слабая
Способ принятия решений	С помощью алгоритма	На основе правил и логического вывода
Поиск решения	Отсутствует или применяется в небольших масштабах	Применяется в крупных масштабах
Решение задач	С помощью правильного алгоритма	С применением правил
Входные данные	Подразумевается наличие правильных данных	Возможно применение неполных или неправильных данных
Обработка непредвиденных входных данных	Связана с возникновением значительных сложностей	Отличается высокой степенью приспособляемости
Выходные данные	Всегда правильные	То, насколько решение приближается к оптимальному, зависит от задачи
Объяснение причин получения конкретных результатов	Отсутствует	Обычно предусмотрено
Приложения	Обработка числовых данных, файлов и текста	Символические рассуждения
Выполнение	Как правило, последовательное	Под влиянием правил
Проект программы	Структурированный проект	Проект с минимально заданной структурой или вообще без структуры
Возможность модификации приложения	Связана с затруднениями	Приемлемая
Расширение возможностей приложения	Осуществляется в виде крупных этапов	Происходит постепенно

Экспертные системы обычно используются с той целью, чтобы легче было справиться с неопределенностью. Неопределенность может обнаруживаться во входных данных экспертной системы и даже в самой базе знаний. На первый взгляд это может показаться удивительным для тех специалистов, которые привыкли работать в рамках обычного программирования. Но бóльшая часть человеческих знаний является эвристической. Это означает, что знания могут оказаться правильными не во всех ситуациях, поэтому вместо использования имеющихся знаний приходится применять какой-то другой подход. Кроме того, входные данные могут оказаться неправильными, неполными, несогласованными, а также содержать ошибки разных типов. Алгоритмические решения не позволяют справляться с подобными ситуациями, поскольку алгоритм обязан гарантировать решение задачи с помощью конечной последовательности шагов.

В зависимости от входных данных и состояния базы знаний экспертная система может предложить правильный ответ, приемлемый ответ, неприемлемый ответ или вообще не дать ответа. Безусловно, на первый взгляд такая ситуация может показаться шокирующей, но это лучше, чем отсутствие возможности получить ответ при некоторых обстоятельствах. Кроме того, следует учитывать еще одно важное соображение — хорошая экспертная система способна действовать не хуже по сравнению с самым лучшим решателем задач соответствующего класса (экспертом-человеком) и даже более успешно. Если бы был известен эффективный алгоритмический метод, действующий лучше, чем любая экспертная система, то достаточно было бы просто воспользоваться этим методом. Но важнее всего иметь возможность применить самый лучший, причем, вероятно, единственный инструмент для выполнения работы.

Недекларативное программирование

При использовании таких языков, как PROLOG и SQL, очень широко применяются недекларативные подходы. Для решения задач искусственного интеллекта были разработаны новые версии языка PROLOG, а язык SQL стал стандартным средством работы с реляционными базами данных. Языки подобного типа используются для создания приложений весьма разнообразных типов, начиная с автономных приложений и заканчивая приложениями, применяемыми в сочетании с другими подходами.

Программирование на основе индукции

Одним из приложений искусственного интеллекта, которое привлекло значительный интерес, является программирование **на основе индукции**. К этому направлению относится классический алгоритм ID3, применяемый для машинного обучения, а также более новые алгоритмы C4.5 и C5.1. При указанном подходе программа обучается путем формирования обобщений на основе представленно-

го образца, подобно тому, как человек мог бы обучиться наращивать последовательность 2,4,6,“?”. Одним из направлений, в котором был успешно применен данный подход, является доступ к базе данных. При этом пользователь может не указывать конкретные значения для одного или нескольких полей, по которым должен быть выполнен поиск, поскольку для него достаточно выбрать лишь один или несколько подходящих примеров полей с искомыми характеристиками. После этого программа доступа к базе данных выявляет логическим путем характеристики данных и выполняет поиск соответствующих записей в базе данных. В качестве наглядных примеров использования распознавания образов при поиске могут служить Oracle и другие системы управления реляционными базами данных, в которых применяется язык SQL (Structured Query Language — язык структурированных запросов), ключевые слова которого взяты из английского языка.

Некоторые инструментальные средства экспертных систем предоставляют возможность использовать обучение по индукции, в процессе которого они принимают для анализа примеры и практические задания, а затем автоматически вырабатывают правила (подробнее об этом — в приложении Ж).

1.13 Искусственные нейронные системы

В 1980-х годах в число широко применяемых принципов программирования вошло новое направление разработок, получившее название **искусственных нейронных систем** (**Artificial Neural System — ANS**). Это направление основано на открытии одного из способов обработки информации мозгом. Указанный подход иногда упоминают под названием **коннекционизма** (от английского слова connection — соединение), поскольку в нем процесс решения задач моделируется путем обучения моделей нейронов, соединенных в сеть. В настоящее время искусственные нейронные системы применяются во многих приложениях, начиная с распознавания лиц, медицинской диагностики, ведения игр, распознавания речи и заканчивая управлением двигателями автомобилей.

Исследователи, придерживающиеся коннекционистского направления, полагают, что размышления о вычислениях, проводимые в терминах модельного представления мозга, а не модельного представления компьютера, ведут к лучшему пониманию характера интеллектуального поведения. Таким образом, в основе коннекционизма лежит такая идея, что в области искусственного интеллекта можно добиться существенного прогресса, подходя к решению задач с точки зрения организации вычислений в том стиле, который применяется в мозгу, а не с помощью манипулирования символами на основе правил. Нейронные сети представляют собой способ обработки информации, базирующийся на том принципе, по которому осуществляется обработка информации в таких биологических нервных

системах, как мозг. Фундаментальным понятием нейронных сетей является структура системы обработки информации. В системе нейронной сети, состоящей из большого количества обрабатывающих элементов (или нейронов), соединенных многочисленными связями, для решения задач используется такой же способ обучения на примерах, каким руководствуются люди. Конфигурация нейронной сети настраивается на реализацию конкретного приложения, такого как классификация данных или распознавание образов, с помощью процесса усвоения знаний, называемого **обучением**. При этом, как и в биологических системах, обучение предусматривает внесение изменений в характеристики синаптических соединений, существующих между нейронами.

Предусмотрено много способов классификации типов искусственных нейронных систем, но самый полезный отличительный классификационный признак состоит в том, должно ли быть предусмотрено для такой системы обучающее множество входных и выходных данных, или нет. Если обучающее множество предусмотрено, то искусственная нейронная система называется *основанной на контролируемой модели*. А если система должна обучаться классификации входных данных, не зная каких-либо выходных данных, то она именуется *неконтролируемой*. Хорошим примером контролируемой искусственной нейронной системы является система, применяемая при распознавании лиц. В отличие от этого, если точно не известно, какими должны быть выходные данные, то искусственная нейронная система служит в качестве хорошего классификатора для группирования входных данных, как при распознавании больных и здоровых людей в случае обнаружения вспышек заболевания.

Нейронные сети могут классифицироваться по способу соединения нейронов; отличаться от других тем, что в них применяются определенные типы вычислений, выполняемых нейронами; обеспечивать различные способы передачи шаблонов активности по сети; а также характеризоваться определенным способом обучения и скоростью обучения. Нейронные сети применялись для решения практических задач всех типов. Основным преимуществом нейронных сетей является то, что они позволяют решать задачи, слишком сложные для обычных технологий, т.е. такие задачи, которые не имеют алгоритмического решения или для которых алгоритмическое решение является слишком сложным, чтобы его можно было определить аналитически. Вообще говоря, нейронные сети хорошо подходят для решения задач, которые успешно решают и люди, но не могут объяснить, как они это делают. В число таких задач входят распознавание образов и прогнозирование (в последнем случае требуется также обнаруживать тенденции изменения данных). В настоящее время искусственный интеллект широко используется при **анализе скрытых закономерностей в данных** для обнаружения в исторических данных таких шаблонов, которыми компания могла бы руководствоваться в будущем. Например, анализ скрытых закономерностей в данных может применяться

в компании для определения того, когда следует накапливать на складе определенные товары с учетом сезонных вариаций спроса.

Кроме того, нейронные сети используются в качестве интерфейсной части таких экспертных систем, для которых требуются большие объемы входных данных от датчиков, а также необходим отклик в реальном времени. В разделе `FAQ` группы новостей `comp.ai.neural-nets` предоставляются без ограничений для загрузки свыше пятидесяти искусственных нейронных систем; другие информационные ресурсы упоминаются в приложении Ж.

Задача коммивояжера

Искусственные нейронные системы позволили добиться замечательных успехов в обеспечении отклика в реальном времени при решении сложных задач распознавания образов. В 1980-х годах нейронная сеть, эксплуатируемая на обычном микрокомпьютере, позволила получить очень качественное решение задачи коммивояжера за 0,1 секунды, т.е. намного быстрее по сравнению с оптимальным решением, для получения которого потребовался целый час процессорного времени на мэйнфрейме. Задача коммивояжера является очень важной, поскольку представляет собой классическую задачу, с которой приходится сталкиваться при оптимальной маршрутизации пакетов в системе передачи данных. Задача поиска оптимальных маршрутов является важным средством минимизации времени прохождения пакетов, поскольку от этого зависят эффективность и быстродействие, достигаемые как при маршрутизации пакетов через Интернет, так и при доставке посылок по почте многочисленным адресатам.

В своей основной постановке задача коммивояжера сводится к вычислению самого короткого маршрута через города, указанные в некотором списке. В табл. 1.13 показаны возможные маршруты для городов, количество которых составляет от одного до четырех. Обратите внимание на то, что количество маршрутов пропорционально факториалу количества городов за вычетом единицы, $(N-1)!$.

Количество маршрутов для 10 городов равно $9! = 362880$, а для 30 городов количество возможных маршрутов составляет примерно $29! = 8.8^{30}$. Задача коммивояжера представляет собой классический пример **комбинаторного взрыва**, поскольку количество возможных маршрутов увеличивается настолько быстро, что для какого-либо количества городов, которое может встретиться в реальном случае, невозможно найти практически применимые решения. В частности, если решение задачи для 30 городов потребует одного часа процессорного времени, то для 31 города потребуется 30 часов, а для 32 городов — 330 часов. Но фактически указанные размерности задачи очень малы по сравнению с применяемыми в действительности тысячами телекоммуникационных коммутаторов и сотнями городов, которые рассматриваются в задачах маршрутизации пакетов данных и реальных предметов транспортировки.

Таблица 1.13. Маршруты, обнаруженные в процессе решения задачи коммивояжера

Количество городов	Маршруты
1	1
2	1 – 2 – 1
3	1 – 2 – 3 – 1
	1 – 3 – 2 – 1
4	1 – 2 – 3 – 4 – 1
	1 – 2 – 4 – 3 – 1
	1 – 3 – 2 – 4 – 1
	1 – 3 – 4 – 2 – 1
	1 – 4 – 2 – 3 – 1
	1 – 4 – 3 – 2 – 1

Нейронная сеть позволяет найти решение для случая с 10 городами почти за такое же короткое время, как и для случая с 30 городами, тогда как для обычной программы требуется намного больше времени. В случае с 10 городами с помощью нейронной сети был найден один из двух оптимальных маршрутов, а в случае с 30 городами — один из лучших 100 000 000 маршрутов. Это достижение становится еще более впечатляющим, если учесть, что найденный маршрут относится к числу самых лучших из оптимальных решений. Безусловно, нейронные сети могут не всегда давать оптимальный ответ, но способны предложить наилучший вариант в режиме реального времени. А во многих случаях ответ, правильный на 99,99999999999999999%, полученный за 0,1 секунды, лучше, чем ответ, правильный на 100%, который получен через 30 часов. В качестве других способов решения этой задачи использовались генетические алгоритмы, как показано в приложении Ж, и алгоритм Evolutionary Ant [23]. Еще один успешно опробованный подход основан на реальных результатах изучения ДНК [87].

Элементы искусственной нейронной системы

Искусственная нейронная система может рассматриваться как аналоговый компьютер, в котором используются простые обрабатывающие элементы, соединенные друг с другом главным образом параллельно. Обрабатывающие элементы выполняют очень простые логические или арифметические операции над своими входными данными. Основой функционирования искусственной нейронной системы является то, что с каждым элементом такой системы связаны **весовые коэффициенты**. Эти весовые коэффициенты представляют информацию, хранимую в системе.

Типичный искусственный нейрон показан на рис. 1.10. Нейрон может иметь несколько входов, но только один выход. Человеческий мозг содержит приблизительно 10^{11} нейронов, а каждый нейрон может иметь тысячи соединений с другими. Входные сигналы нейрона умножаются на весовые коэффициенты и складываются для получения суммарного входа нейрона, I. Весовые коэффициенты могут быть представлены в виде матрицы и обозначены подстрочными индексами.

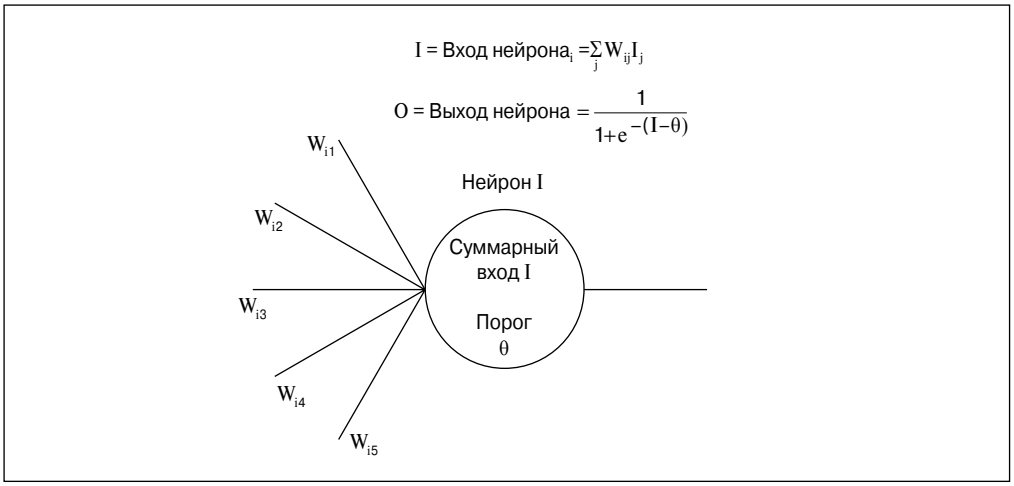


Рис. 1.10. Обрабатывающий элемент нейрона

Выходной сигнал нейрона часто формируется с помощью **сигмоидальной функции** от входного. Реакция, которая может быть описана с помощью сигмоидальной функции, является типичной для реальных нейронов; эта реакция заключается в том, что выходной сигнал стремится к одному из пределов при получении очень малых и очень больших входных сигналов. Функция, которая связывает выход нейрона с его входами, называется **функцией активизации**. В качестве нее обычно используется сигмоидальная функция $(1 + e^{-x})^{-1}$. Кроме того, с каждым нейроном связано **пороговое значение**, θ, которое вычитается из общего входного сигнала, I. На рис. 1.11 показана искусственная нейронная система, способная вычислять значение **исключительного ИЛИ (exclusive-OR — XOR)** от ее входов с использованием способа, называемого **обратным распространением**. Сеть, вычисляющая значение исключительного ИЛИ, выдает истинный выход, только если не на всех ее входах имеются истинные значения или не на всех входах имеются ложные значения. Количество узлов в скрытом слое изменяется в зависимости от приложения и проекта.

Нейронные сети не требуют программирования в обычном смысле этого слова. Для обучения нейронных сетей применяются многие другие алгоритмы обучения нейронных сетей, такие как **встречное распространение** и обратное распростра-

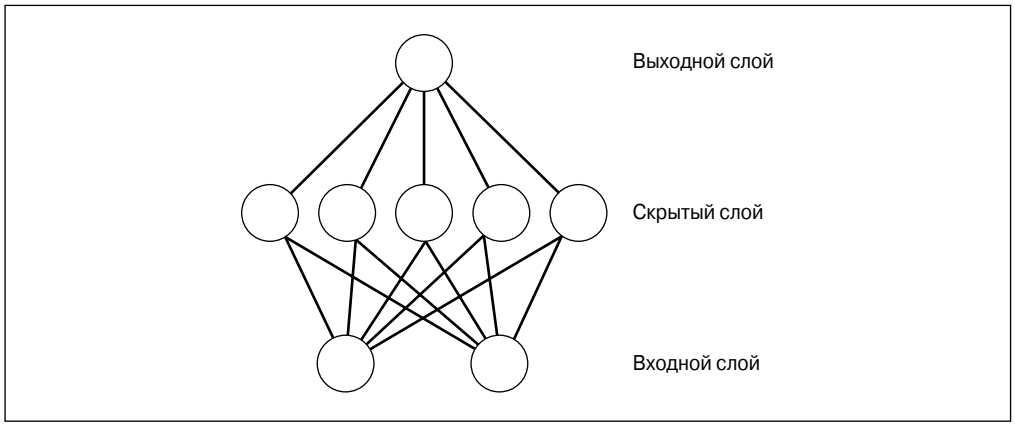


Рис. 1.11. Сеть с обратным распространением

нение. Программист "программирует" сеть, задавая входные данные и соответствующие выходные данные. Сеть обучается, автоматически корректируя весовые коэффициенты в сети, соединяющей нейроны. Весовые коэффициенты и пороговые значения нейронов определяют характер распространения данных через сеть и тем самым задают правильный отклик на обучающие данные. Обучение сети в целях получения правильных ответов может потребовать многих часов или дней, в зависимости от того, какое количество образов должно быть изучено в ходе обучения сети, а также от необходимых аппаратных и программных средств. Но после завершения такого обучения сеть выдает ответы очень быстро.

Если программное моделирование сети не обеспечивает достаточно быстрого получения ответов, то в целях получения отклика в реальном времени искусственная нейронная система может быть изготовлена на основе микросхем. А сами микросхемы могут быть сконструированы сразу после обучения сети и определения весовых коэффициентов.

Характеристики искусственной нейронной системы

По своей архитектуре искусственная нейронная система во многом отличается от обычной вычислительной системы. В обычном компьютере предусмотрена возможность связывать с ячейками памяти дискретную информацию. Например, компьютер позволяет хранить номера карточек социального обеспечения, представленные с помощью кода ASCII, в группах смежных ячеек памяти. А исследование содержимого каждой группы смежных ячеек памяти позволяет непосредственно реконструировать значение одного из номеров карточек социального обеспечения. Такое восстановление данных является возможным, поскольку имеется взаимно-

однозначная связь между каждым символом номера карточки социального обеспечения и ячейкой памяти, содержащей код ASCII этого символа.

С другой стороны, модели искусственных нейронных систем разрабатывались на основе современных теорий функционирования мозга, согласно которым информация представлена в мозгу с помощью весовых коэффициентов. Но непосредственной корреляции между конкретным значением весового коэффициента и конкретным элементом хранимой информации не существует. Такое распределенное представление информации аналогично применяемому в голограмме, поскольку линии голограммы действуют как дифракционная решетка, с помощью которой восстанавливается хранимое изображение при прохождении через нее лазерного луча, но сами по себе не поддаются непосредственной интерпретации.

Нейронная сеть представляет собой приемлемое средство решения задачи, если имеется много эмпирических данных, но нет алгоритма, который обеспечил бы получение достаточно точного решения с достаточно высоким быстродействием. Средства представления данных искусственной нейронной системы обладают несколькими описанными ниже преимуществами по сравнению с памятью обычных компьютеров.

- Память нейронной сети является **отказоустойчивой**. При удалении отдельных частей нейронной сети происходит лишь снижение качества хранимых данных, но не полное исчезновение данных. Это происходит потому, что информация хранится в распределенной форме.
- Качество хранимого изображения снижается постепенно, пропорционально той части сети, которая была удалена. Катастрофической потери информации не происходит. Кроме того, такая форма хранения и обеспечения качества хранимых данных характерна для голограмм.
- Данные хранятся естественным образом, будучи представленными с помощью **ассоциативной памяти**. Ассоциативной памятью называют такую память, в которой достаточно выполнить поиск частично представленных данных, чтобы полностью восстановить всю хранимую информацию. В этом состоит отличие ассоциативной памяти от обычной памяти, в которой восстановление данных осуществляется путем указания точного адреса восстанавливаемых данных. А в ассоциативной памяти вся первоначальная информация может быть извлечена даже при наличии неполных или зашумленных входных данных.
- Сети позволяют выполнять экстраполяцию и интерполяцию на основе хранимой в них информации. А обучение позволяет придать сети способность осуществлять поиск важных особенностей или связей в данных. После этого сеть становится способной к экстраполяции и обнаружению связей во вновь поступающих данных. В одном эксперименте было проведено обуче-

ние нейронной сети на гипотетическом примере семейных связей между 24 людьми. В результате этого сеть приобрела способность правильно отвечать на такие вопросы о связях, в отношении которых обучение не проводилось.

- Сети обладают **пластичностью**. Даже после удаления определенного количества нейронов может быть проведено повторное обучение сети до ее первоначального уровня навыков, если осталось достаточное количество нейронов. Такая особенность является также характерной для мозга, в котором могут быть повреждены некоторые части, но со временем с помощью обучения достигнуты первоначальные уровни навыков.

Благодаря таким особенностям искусственные нейронные системы становятся очень привлекательными для применения в роботизированных космических аппаратах, нефтепромысловом оборудовании, подводных аппаратах, средствах управления технологическими процессами и в других технических устройствах, которые должны функционировать продолжительное время без ремонта в неблагоприятной среде. Искусственные нейронные системы не только позволяют решить проблему надежности, но и предоставляют возможность уменьшить эксплуатационные расходы благодаря своей пластичности. Причем, даже если есть возможность выполнять ремонт аппаратных средств, то, по-видимому, более экономически эффективным является перепрограммирование нейронной сети, чем ее замена.

Но в целом искусственные нейронные системы не очень хорошо подходят для создания приложений, в которых требуются сложные математические расчеты или поиск оптимального решения. Кроме того, искусственная нейронная система не является наилучшим вариантом, если существует практически применимое алгоритмическое решение (безусловно, если нет возможности создать менее дорогостоящую искусственную нейронную систему на основе микросхемы).

Новейшие достижения в технологии искусственных нейронных систем

Для *коннекционистов*, или специалистов по нейронным сетям, основное направление разработок развивается вокруг идеи о том, что в своих психологических исследованиях мы должны “относиться к мозгу серьезно”. Безусловно, истоки этой идеи можно проследить до тех философских работ древнегреческих ученых, в которых рассматривались действия жизненных духовных начал в нервной системе. В дальнейшем Рене Декарт изложил в своих трудах важные предположения, касающиеся этой темы. А разработка самих искусственных нейронных систем началась с исследований по математическому моделированию нейронов, выполненных Маккаллохом и Питтсом в 1943 году. Объяснение процесса обучения с помощью нейронов предложено Хеббом (Hebb) в 1949 году. В хеббовском

обучении эффективность активизации одних нейронов другими возрастает с увеличением количества запусков. Термин **запуск** означает, что нейрон испускает электрохимический импульс, способный стимулировать другие связанные с ним нейроны. Повышение эффективности активизации свидетельствует о том, что проводимость соединений между нейронами на участках их сопряжения, называемых **синапсами**, возрастает с увеличением количества запусков. А в искусственной нейронной системе для моделирования изменений проводимости синапсов естественных нейронов применяется способ корректировки весовых коэффициентов соединений между нейронами [90].

В 1961 году Розенблатт опубликовал свою книгу, оказавшую значительное влияние на дальнейший ход исследований, в которой рассматривалась новая исследованная им разновидность искусственной нейронной системы, получившая название **персептрон**. После этого стало очевидно, что персептрон — замечательное устройство, которое обнаруживает способности к обучению и распознаванию образов. По существу, он состоит из двух слоев нейронов и предоставляет возможность использовать простой алгоритм обучения. Но весовые коэффициенты должны устанавливаться в нем вручную, в отличие от современных искусственных нейронных систем, которые сами способны устанавливать свои весовые коэффициенты на основе обучения. Под влиянием этих достижений в течение 1960-х годов многие исследователи перешли в область искусственных нейронных систем и приступили к изучению персептронов.

Эта ранняя эпоха исследования персептронов подошла к концу в 1969 году, когда Минский (Minsky) и Пейперт (Papert) публиковали книгу *Perceptrons*, в которой показали теоретические ограничения персептронов, рассматриваемых как вычислительная машина общего назначения. Эти авторы подчеркнули то, что персептроны способны вычислять только 14 из 16 основных логических функций, поэтому обладают неустранимыми недостатками. Это означает, что персептрон нельзя считать вычислительным устройством общего назначения. В частности, они доказали, что персептрон не способен вычислять функцию исключительного ИЛИ. Безусловно, эти ученые не проводили серьезных исследований многослойных искусственных нейронных систем, но высказали пессимистическое мнение о том, что даже многослойные сети не будут способны решить задачу вычисления функции исключительного ИЛИ. Бюджетное финансирование исследований в области искусственных нейронных систем было сокращено в пользу символического подхода к разработке искусственного интеллекта с использованием таких языков, как LISP, и алгоритмов. В 1970-х годах получили широкое распространение новые способы представления символической информации в приложениях искусственного интеллекта на основе фреймов, предложенные Минским. В дальнейшем на основе фреймов был создан современный подход, в котором применяются сценарии. Но современная технология интегральных схем позволяет легко конструи-

ровать персептроны и другие искусственные нейронные системы, поскольку эти технические устройства являются достаточно простыми.

Исследования в области искусственных нейронных систем продолжались в небольших масштабах и в 1970-х годах, а в конце 1970-х годов Джеффри Хинтон (Geoffrey Hinton), Джеймс Макклелланд (James McClelland), Дэвид Рамелхарт (David Rumelhart), Пол Смоленский (Paul Smolensky) и другие члены Parallel Distributed Processing Research Group проявили интерес к теориям познания, основанным на нейронных сетях. Основополагающая книга, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, опубликованная этими учеными в 1986 году, ознаменовала собой возврат к коннекционизму как к принципиально значимой теории познания. В дальнейшем Хопфилд (Hopfield) разработал надежное теоретическое основание искусственных нейронных систем на примере Hopfield Net (сети Хопфилда) и показал, что с помощью искусственной нейронной системы можно решать самые разнообразные задачи. Общая структура сети Хопфилда приведена на рис. 1.12. В частности, Хопфилд показал, что с помощью искусственной нейронной системы задачу коммивояжера можно решать за постоянное время, тогда как в обычных алгоритмических решениях обнаруживается комбинаторный взрыв. Искусственная нейронная сеть, выполненная в форме электронной схемы, способна решить задачу коммивояжера за 1 микросекунду или за меньшее время. Кроме того, искусственные нейронные системы способны легко решать другие комбинаторные задачи оптимизации, такие как раскраска карты четырьмя цветами, реконструкция формы объектов в евклидовом пространстве и поиск перестановочного кода.

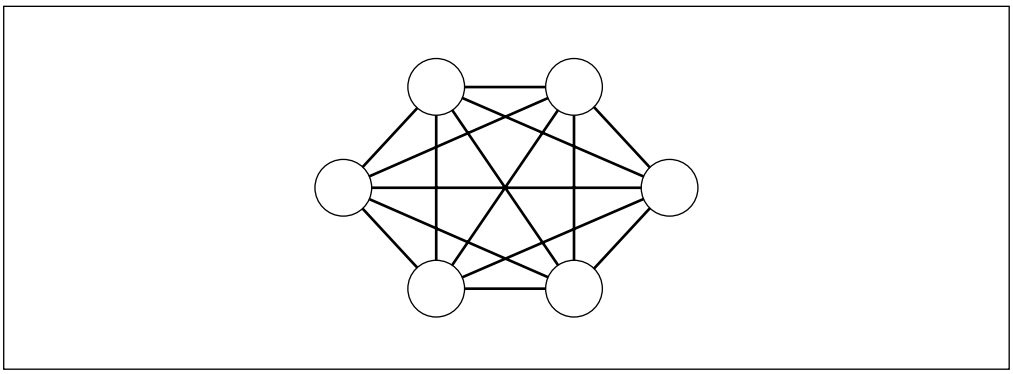


Рис. 1.12. Искусственная нейронная сеть Хопфилда

Одной из искусственных нейронных систем, позволяющих легко решить задачу вычисления исключительного ИЛИ, является сеть **с обратным распространением**, известная также как сеть, основанная на **обобщенном дельта-правиле**. Сеть с обратным распространением, как правило, реализуется в виде сети

с тремя слоями, хотя могут быть также заданы дополнительные слои. Слои, находящиеся между входным и выходным слоями, называются **скрытыми слоями**, поскольку для внешнего мира видимы только входной и выходной слои. Еще одним широко применяемым типом искусственной нейронной системы является сеть со встречным распространением, которая была предложена Хехтом-Нилсеном (Hecht-Nielsen) в 1986 году. А один из важных теоретических результатов в математике, теорему Колмогорова, можно интерпретировать как доказательство того, что трехслойная сеть с n входами и $2n + 1$ нейронами в скрытом слое позволяет представить любую непрерывную функцию.

Приложения технологии искусственных нейронных систем

Важный пример обучения с помощью обратного распространения был продемонстрирован с помощью нейронной сети, которая обучалась правильному произношению слов, представленных в виде текста. Эта искусственная нейронная система обучалась путем корректировки своего выхода с помощью устройства преобразования текста в речь, называемого DECTalk. Для разработки правил грамотного произношения, используемых в устройстве DECTalk, потребовалось двадцать лет лингвистических исследований, а искусственная нейронная система освоила эквивалентные навыки произношения за одну ночь, просто прослушивая правильное произношение речи во время чтения текста. При этом в самой этой искусственной нейронной системе не было заложено путем программирования никаких лингвистических навыков.

Искусственные нейронные системы использовались для распознавания радарных целей с помощью электронных и оптических компьютеров. В перспективе на основе новых реализаций нейронных сетей с применением оптических компонентов могут быть созданы оптические компьютеры с быстродействием, превышающим в миллионы раз быстродействие электронных компьютеров. Привлекательность реализаций искусственных нейронных систем на основе оптических компонентов обусловлена тем, что свет характеризуется свойством параллельного распространения. Это означает, что не происходит обоюдное воздействие совместно распространяющихся световых лучей. А такие оптические компоненты, как зеркала, линзы, высокоскоростные программируемые пространственные модуляторы света, массивы оптических устройств с двумя устойчивыми состояниями (способные действовать как оптические нейроны) и дифракционные решетки, позволяют легко вырабатывать и осуществлять манипуляции с огромными количествами фотонов. Складывается впечатление, что оптические компьютеры, спроектированные как искусственные нейронные системы, и сами ИНС являются дополнительными по отношению друг к другу.

Классические приложения искусственных нейронных систем обсуждаются в [35]. В частности, искусственные нейронные системы являются полезным компонентом таких систем управления, в которых обычные подходы являются неприменимыми [38]. В действительности, как показано в [48], искусственные нейронные системы широко используются во многих промышленных системах управления; в этом можно также убедиться, перейдя по ссылкам, приведенным в приложении Ж.

1.14 Коннекционистские экспертные системы и индуктивное обучение

С использованием искусственных нейронных систем могут быть также созданы экспертные системы. В одной из экспертных систем искусственная нейронная система представляет собой базу знаний в области диагностики, сформированную путем обучения на основе примеров, взятых из практической медицины. Эта экспертная система предпринимает попытки распознать заболевание по его симптомам как одно из известных заболеваний, по данным о которых была обучена система. Кроме того, была спроектирована машина логического вывода, называемая MACIE (Matrix Controlled Inference Engine), в которой применяется база знаний на основе искусственной нейронной системы. В этой системе используется прямой логический вывод для формирования логических заключений и обратный логический вывод — для передачи пользователю запросов на предоставление тех дополнительных данных, которые требуются для выработки решений. Безусловно, сама эта искусственная нейронная система не способна объяснить, почему ее весовым коэффициентам присвоены те или иные значения, но машина MACIE способна интерпретировать ответы искусственной нейронной системы и вырабатывать правила `IF-THEN`, применимые для объяснения ее знаний.

В подобной экспертной системе на основе искусственной нейронной системы используется **индуктивное обучение**. Это означает, что система **логически выводит** информацию, содержащуюся в ее базе знаний, с помощью примеров. Индукция — это процесс логического вывода общего случая из частных. Как показано в приложении Ж, кроме искусственных нейронных систем, имеется целый ряд коммерчески доступных программных средств принятия решений, позволяющих явно вырабатывать правила с применением примеров. Индуктивное обучение используется для уменьшения значимости или устранения узкого места, связанного с приобретением знаний. Вся нагрузка по приобретению знаний возлагается на экспертную систему, поэтому появляется возможность сократить время разработки и повысить надежность, если система логическим путем выводит правила, которые не были известны человеку. Сведения о многих экспертных системах,

применяемых в сочетании с искусственными нейронными системами, приведены в [36].

1.15 Современное состояние разработок в области искусственного интеллекта

В течение 1990-х годов в области искусственного интеллекта были реализованы значительные достижения, и эта тенденция продолжается в XXI столетии. А приверженцы таких взглядов, что единственным приемлемым направлением разработок искусственного интеллекта является создание сильного искусственного интеллекта, основанного на логике и рассуждениях, сумели добиться лишь очень ограниченного успеха в создании систем, которые так и не вышли за пределы замкнутого мира лабораторий. Успешно действующие системы на основе искусственного интеллекта, которые смогли противостоять грубой реальности жесткого, конкурентного рынка, обычно создавались на базе систем, в основе которых лежат биологические принципы. Эволюция искусственного интеллекта с 1950-х годов показана на рис. 1.13. На этом рисунке течение времени показано сверху вниз. Первоначально разработки в области искусственного интеллекта подразделялись на два основных подхода. В одном из них использовались модели, созданные на базе символической логики, а в основу другого важного подхода легли результаты изучения биологических процессов. Между этими двумя направлениями всегда существовала жестокая конкуренция, но исходя из результатов опыта, полученного при реализации решений на базе искусственного интеллекта, теперь можно утверждать, что в общем ни один из этих отдельно взятых подходов не позволяет получить правильный ответ на сложную задачу. Самым лучшим, на что можно надеяться, является оптимальное решение, но чаще всего приходится довольствоваться просто хорошим решением.

В правой части схемы, приведенной на рис. 1.13, показаны новые подходы, основанные на достижениях физики, особенно тех, в которых используются квантовые компьютеры. Безусловно, в наши дни считается, что применение квантовых компьютеров приведет лишь к ускорению поиска на несколько порядков величины, но такой способ использования указанных устройств слишком упрощен. Сразу после открытия в XX веке квантовой механики благодаря анализу уравнения Шрёдингера и эквивалентного ему формализма, матричной механики Гейзенберга, многие ученые, включая Альберта Эйнштейна, ее не признавали. Сам Эйнштейн любил повторять: “Бог не играет в кости со Вселенной”. Тем не менее квантовая механика неразрывно связана с вероятностью, и поэтому, в отличие от классической ньютоновской теории, теперь ни одно утверждение не может рассматриваться как абсолютная истина. Фактически предложена даже та-

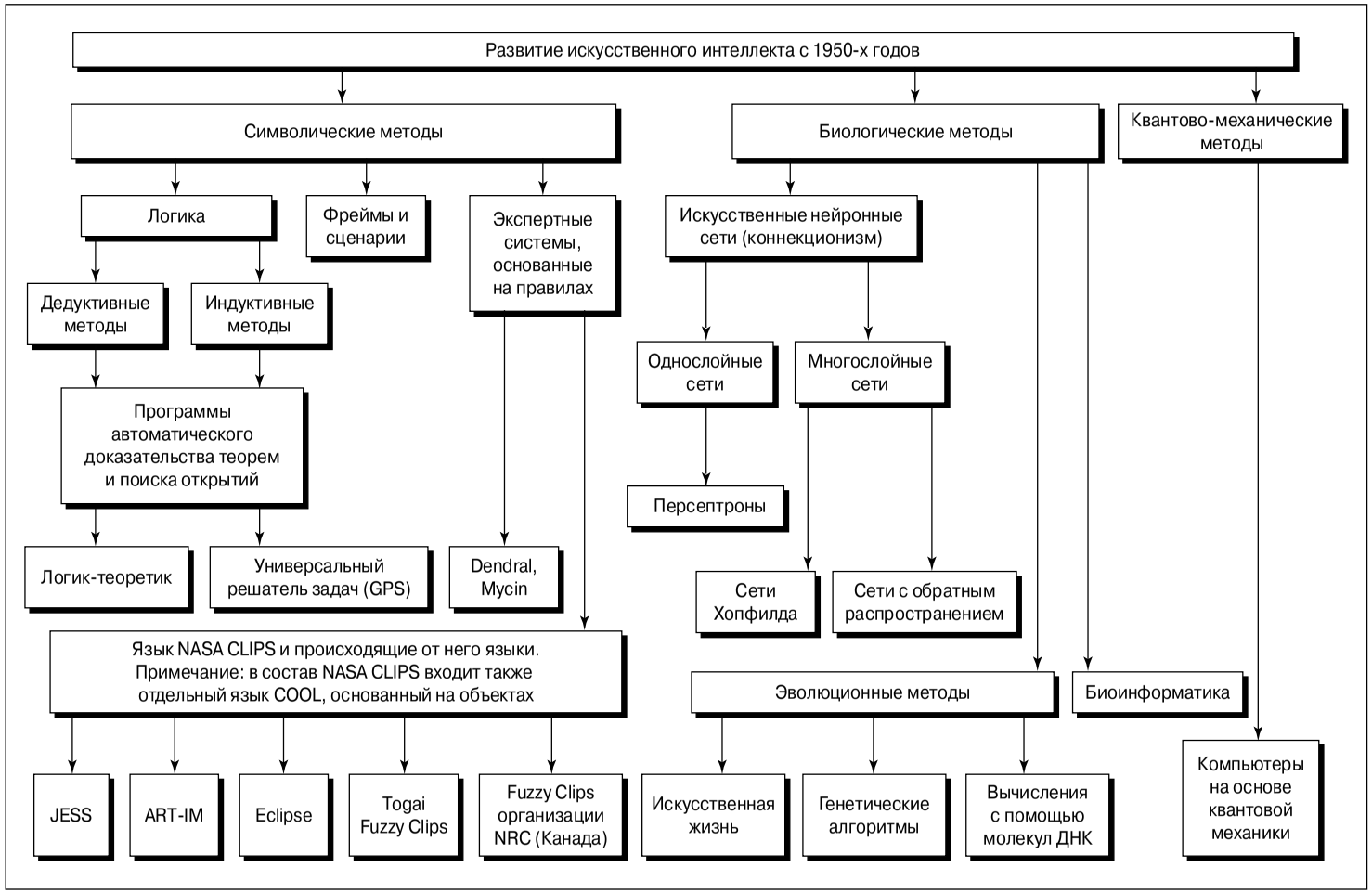


Рис. 1.13. Эволюция искусственного интеллекта

кая теория, согласно которой сознание рассматривается как феномен квантовой механики [85].

Многие ученые размышляли на тему о том, что разум и сознание сами могут рассматриваться как эмерджентное (от слова emergent — появляющийся) свойство мозга, поскольку мозг состоит из нейронов, которые в конечном итоге состоят из атомов и субатомных частиц. В свою очередь, эти частицы могут рассматриваться как квантовая пена, лежащая в основе всего существующего. Пространство субатомных частиц является дискретным, а не непрерывным, как принято считать в математике и классической ньютоновской физике, а расстояния в нем настолько малы, что не превышают 10^{-60} метров. По мнению физиков, в масштабах столь малых расстояний пространство имеет не 3 обычных измерения, а 11 измерений. Безусловно, для человека очень лестно считать, что все истины могут быть установлены лишь с помощью человеческих рассуждений и размышлений, но, как установили еще философы Древней Греции, т.е. примерно 2500 лет тому назад, в исследованиях, не подкрепленных обоснованным экспериментированием, игнорируются законы реального мира, поэтому они могут приводить к заключениям, подобным приведенным ниже.

```
Посылка. Я неподвижен, поскольку не чувствую своего
   перемещения
Посылка. Я вижу, как солнце встает и садится
Заключение. Следовательно, солнце движется вокруг меня
```

При обсуждении эволюционных алгоритмов термин **эмерджентность** приобретает особое значение. Как эмерджентное характеризуется неожиданно возникающее поведение, которое невозможно было предвидеть заранее. Например, может показаться, что вода в ручье течет очень плавно, но это впечатление внезапно нарушается после падения в воду камня. В зависимости от скорости воды, размера камня и других факторов может возникнуть турбулентность. Вода обнаруживает нелинейное поведение в случае газифицированного потока, разрывного течения и в других случаях, в которых вступают в силу динамические изменения. Эти изменения не могут быть предсказаны с помощью обычной гидродинамики и требуют применения для своего описания дифференциальных уравнений второго или более высокого порядка. Чаще всего получение аналитического решения этих уравнений не представляется возможным, поэтому самый лучший способ действия, который возможен в таких случаях, состоит в получении приближенных числовых решений. Было выявлено много важных примеров эмерджентного поведения и созданы приложения, предназначенные для управления городами, руководства предприятиями, а также для использования в других весьма разнообразных областях [54].

Классическим примером эмерджентного поведения является поведение колонии муравьев, в которой действует интеллект децентрализованного типа, называемый **коллективным интеллектом** [59]. Коллективный интеллект является распределенным и в этом коренным образом отличается от централизованного интеллекта, который проявляют люди и другие млекопитающие, но весьма успешно действует в сообществах насекомых, живущих колониями. Исследования в области приложений коллективного интеллекта проводились для создания надежно функционирующих колоний роботов на других планетах, в которых каждый отдельный робот может не расходовать значительный объем ресурсов на обеспечение своей интеллектуальности (что позволяет экономить затраты и энергию), но вся колония может процветать. Представляются перспективными и другие приложения, касающиеся заводов, сетей и распределенных систем. Интернет также может рассматриваться как проявление коллективного интеллекта, в котором используется распределенный интеллект коммутаторов пакетов низкого уровня для маршрутизации пакетов с высокой степенью выживаемости. Указанное свойство вполне соответствует одной из первоначальных целей создания Интернет (или Arpanet, как первоначально называлась эта сеть), сформулированных в 1960-х годах.

Еще одним важным новым направлением, возникшим в рамках работ по искусственному интеллекту, является создание эволюционных алгоритмов [26]. Такие алгоритмы, как правило, используются для повышения надежности конечных результатов наряду с другими методами искусственного интеллекта, такими как искусственные нейронные системы. Проблема, возникающая при использовании искусственных нейронных сетей, генетических алгоритмов и других схем, состоит в том, что конечные решения могут становиться ограниченными в локальном минимуме, а не сходиться к истинному решению в глобальном максимуме. По этим причинам в некоторых методах намеренно сохраняются для репродукции некоторые “непригодные” решения, а не выбираются лишь наиболее пригодные, поскольку именно первоначально непригодное решение может оказаться способным дальше всего отойти от локального минимума и продвинуться вплоть до глобального максимума.

Эволюция коннекционистских систем — это мощный способ оптимизации архитектуры искусственной нейронной сети, позволяющий принимать лишь самые основные предположения, касающиеся того, какой должна быть архитектура, или вообще обходиться без каких-либо предположений [58]. Безусловно, в искусственных нейронных сетях применяются способы параллельной обработки, поэтому их реализация требует большого объема компьютерных мощностей, но эти способы становятся все более легко осуществимыми по мере того, как компьютеры становятся мощнее и расширяется возможность получения доступа ко многим компьютерам для вычислений, поскольку стремительно развивается Grid-технология

Интернет, и частные компании предоставляют общий доступ ко многим собственным компьютерам для проведения вычислений [29].

Строго говоря, эволюционные алгоритмы того типа, который используется в компьютерах, не основаны на теории эволюции Дарвина, реализуемой в виде небольших последовательных шагов в течение многих тысяч или миллионов лет, а базируются на конкурирующей теории Ламарка. Ламарк был современником Дарвина, жившим в XIX веке, который считал, что если животное получало дополнительную тренировку каких-то конечностей в результате физических усилий, то потомки этого животного наследовали более сильные конечности. Применение компьютеров позволяет обеспечить ход эволюции как в случае, рассматриваемом Ламарком, за счет выбора для воспроизводства только тех экземпляров объектов, которые подошли ближе других к заданным критериям пригодности.

Кроме того, для создания форм искусственной жизни в составе компьютерной программы применялись эволюционные алгоритмы. Эти алгоритмы не используются по такому же принципу, как генетические алгоритмы или нейронные сети, т.е. просто для решения какой-либо конкретной задачи. Как правило, эволюционные алгоритмы применяются в более широком контексте таких научных областей, как экология или экономика, для изучения связей “хищник–жертва” в дикой природе или, что в большей степени затрагивает интересы самого человека, для изучения любой среды, например отношений “изготовитель–потребитель” в мировой экономике. К тому же в видеоиграх широко используются алгоритмы, позволяющие моделировать окружающую среду, с помощью которой определяются новые виртуальные создания, вступающие в борьбу с игроком. Классическим примером применения таких алгоритмов является чрезвычайно мощная игра Sims; сетевая версия этой игры позволяет людям вести игру друг с другом в интерактивном режиме во всем мире.

При использовании эволюционных методов в сочетании с коннекционистскими системами могут возникать удивительные явления, которые не были заранее запланированы. На этом основан еще один повод для критики коннекционистских систем теми, кто поддерживает методы искусственного интеллекта, базирующиеся на логике, поскольку искусственные нейронные сети не позволяют найти объяснение того, как были достигнуты реализованные в них правила создания соединений и получены весовые коэффициенты. Безусловно, во многих проектах были предложены искусственные нейронные системы, предусматривающие возможность хотя бы частично “объяснить” сформированные в них правила, но остается фактом то, что такие правила и весовые коэффициенты не проектируются людьми, а формируются как “условный рефлекс”, с учетом того, какими являются входные данные и желаемые выходные данные системы. В определенном смысле искусственная нейронная сеть или эволюционный алгоритм либо “выращивают” правильное решение, либо “не выживают”.

В процессе решения задачи декодирования генома человека все большее значение приобретала область **биоинформатики**, поскольку в ней постепенно создавались компьютерные технологии, позволяющие обрабатывать поистине колоссальные объемы информации. В этой области открылось много вакансий для специалистов по искусственному интеллекту. Кроме того, постепенно были созданы такие новые области, как **геномика** и **протеомика**. Геном человека состоит приблизительно из 30 000 генов. Каждый ген по существу представляет собой живую фабрику, которая производит белки для выполнения физиологических функций, либо путем непосредственного осуществления действий в клетках, либо путем передачи сообщений в клетки. В настоящее время очередная грандиозная задача состоит в определении того, какие функции выполняет каждый из этих генов и белков; это важно не только для изучения здоровых генов, но и для распознавания генетических заболеваний. Для решения подобных задач требуются огромные объемы вычислительных мощностей из-за колоссального количества данных и огромных размерностей возможных решений. При этом жизненно важным инструментальным средством становится искусственный интеллект.

Искусственный интеллект доказал также свою значимость в области обороны. По окончании первой войны в Персидском заливе в 1990 году было заявлено, что экономия от использования средств искусственного интеллекта в планировании материально-технического обеспечения военных операций во много раз окупила все затраты на исследования по искусственному интеллекту, взятые на себя агентством DARPA (Defense Advanced Research Projects Agency — Агентство перспективных исследовательских программ) с 1950-х годов (`http://www.au.af.mil/au/aul/school/acsc/ai02.htm`). В настоящее время искусственный интеллект позволяет экономить многие миллиарды долларов благодаря применению компьютерных игр для военного обучения вместо проведения настоящих учений, в которых нельзя обойтись без использования топлива и боеприпасов.

В 1950-х годах очень трудно было себе представить, насколько значительным окажется коммерческий успех, достигнутый с помощью средств искусственного интеллекта. В этот период, а также в течение следующей четверти столетия исследователи в области искусственного интеллекта в основном направляли свои усилия на поиск гуманных, научно значимых приложений, таких как доказательство теорем и автоматизация процесса совершения открытий. Безусловно, программы с искусственным интеллектом использовались также для ведения игр в шашки, шахматы и других настольных игр, но в целом такие программы разрабатывались с серьезными целями понимания человеческого мышления, а не ради их развлекательной ценности.

Для этих исследователей, работавших в ранний период развития искусственного интеллекта, было бы весьма удивительно то, насколько велик современный

коммерческий успех искусственного интеллекта в создании видеоигр и специальных эффектов в фильмах. В настоящее время индустрия видеоигр на равных ведет борьбу с киноиндустрией и музыкальной индустрией за деньги потребителя. Люди получают огромное удовольствие от участия в интерактивных играх, особенно если при этом есть возможность вступать в игру с людьми, находящимися в других странах, городах или штатах. В развитых видеоиграх широко используются методы искусственного интеллекта для создания виртуальных существ, которые ведут себя вполне реалистично, а не действуют в соответствии с простыми предсказуемыми шаблонами.

Все больше университетов теперь предлагает не только курсы, но и дипломы специалистов по видеоиграм. Это может служить признанием того, что в этой области уже создана многомиллиардная индустрия, предоставляющая хорошие возможности трудоустройства. Кроме того, средства искусственного интеллекта широко применяются в бизнесе. Наиболее ярким примером такого применения является создание интеллектуальных помощников, позволяющих с помощью базы знаний отделять простые задачи от сложных, поскольку, как известно, 90% задач являются простыми. К тому же на многих деловых предприятиях используются экспертные системы, оснащенные реальными экспертными знаниями. Единственная проблема, сдерживающая развитие в этой области, заключается в том, что подобные деловые системы основаны на внутрифирменной информации компании, поэтому подробные сведения о данных системах, как правило, не публикуются, чтобы конкуренты не могли благодаря этому получить преимущество.

Безусловно, имеется еще много примеров экспертных систем, описанных в приложении Ж, которые можно найти в Интернет, но еще больше сведений по этой теме приведено в World Wide Web, в некоторых статьях, опубликованных на таких узлах, как CiteSeer, и в сообщениях групп новостей, подобных `comp.ai.shells`, которые в основном посвящены описанию экспертных систем CLIPS. Преимуществом группы новостей является то, что в нее можно отправить вопрос и надеяться на то, что кто-то на него ответит.

Перспективы создания систем искусственного интеллекта и экспертных систем не только открывают дополнительные возможности трудоустройства, но и становятся причиной появления новых специальностей, таких как **инженер по онтологии**. В действительности эта специальность может рассматриваться как одно из направлений использования вычислительной техники в фундаментальных науках, таких как философия (причем фактически в настоящее время на коммерческих предприятиях находят свое применение лишь инженеры по онтологии). В системах искусственного интеллекта и экспертных системах термин *онтология* имеет смысл, отличный от обычной традиционной трактовки этого термина в философии.

Онтология — это явная формальная спецификация терминов проблемной области и отношений между ними [42]. Безусловно, средний пользователь Web этого не осознает, но онтологии незаметно для постороннего взгляда широко применяются в Web. К ним относятся не только крупные онтологии, организованные в виде таксономий (т.е. иерархических нисходящих коллекций взаимосвязанной информации) на таких узлах, как Yahoo!, но и более мелкие онтологии, применяемые на таких узлах, как Amazon.com, eBay и другие, которые предназначены для классификации товаров с учетом отпускной цены или времени аукциона и начального предложения. По существу онтология представляет собой стандартный, согласованный набор терминов, применяемых для описания определенной прикладной области, будь то книги, товары, продаваемые на аукционе, или что-то другое. Если отсутствует общепринятый словарь, то разработка крупного приложения проходит по сценарию создания вавилонской башни, в котором никто не знает, правильно ли применяются термины, касающиеся рассматриваемой области. Многие организации разработали свои собственные онтологии, чтобы можно было проще объяснить, о чем идет речь, не сталкиваясь с неоднозначным толкованием.

Одним из наибольших преимуществ стандартизированной онтологии является то, что она может быть приспособлена для обработки с помощью компьютера, после чего компьютеры приобретают возможность помогать людям в поиске необходимых элементов данных. Аналогичным направлением разработок является создание языков для различных научных и прикладных областей на основе языка XML (Extensible Markup Language — расширяемый язык разметки). В настоящее время действительно велика потребность в специалистах, имеющих подготовку в области искусственного интеллекта, которые пожелали бы стать инженерами по онтологии и заняться классификацией огромных объемов знаний. Кроме того, онтологическая инженерия используется для создания и сопровождения баз данных, применяемых в экспертных системах. Это — нетривиальная задача, особенно с учетом того, что базы знаний становятся все больше по объему, а пользователю разрешается вводить новые знания, предназначенные для использования в экспертных системах.

1.16 Резюме

В настоящей главе приведен общий обзор научных задач и достижений, которые привели к созданию экспертных систем. Задачи, для решения которых обычно применяются экспертные системы, как правило, не поддаются решению с помощью обычных программ, поскольку для них отсутствует известный или достаточно эффективный алгоритм. С другой стороны, экспертные системы основаны на знаниях, поэтому могут эффективно использоваться для реальных задач,

которые являются слабо структурированными и с трудом поддаются решению с помощью других средств. Кроме того, в данной главе рассмотрены различные подходы к представлению знаний и описаны их преимущества и недостатки. Более подробная информация на эту тему приведена в [34].

Кроме того, были описаны преимущества и недостатки экспертных систем в контексте выбора соответствующей предметной области для конкретных приложений экспертных систем. Приведены также критерии выбора соответствующих приложений.

В главе представлены основные сведения о командных интерпретаторах экспертных систем и показано, как они применяются в экспертных системах, основанных на правилах. Описан основной цикл функционирования машины логического вывода, базирующийся на принципе “распознавание–действие”. Применение такого цикла проиллюстрировано на примере простого правила. Наконец, была описана связь экспертных систем с другими подходами к программированию с точки зрения выбора предметных областей, наиболее подходящих для каждого из этих подходов. Сделан принципиальный вывод, согласно которому экспертные системы должны рассматриваться как инструментальное средство программирования определенного типа, которое подходит для создания одних приложений и не подходит для создания других. (Характерные особенности и вопросы применимости экспертных систем будут рассматриваться более подробно в следующих главах.) Кроме того, преимущества и недостатки экспертных систем описаны в контексте выбора предметной области, в наибольшей степени соответствующей специализации конкретного эксперта.

Дальнейшие перспективы развития искусственного интеллекта, несомненно, будут связаны с использованием результатов новых революционных достижений в разработке квантовых компьютеров [11], с применением колоссальной вычислительной мощи, достигаемой благодаря связыванию миллионов компьютеров в Интернет на основе Grid-технологии, а также биотехнологии WETLAB, в которой используются нити РНК. Указанные методы мобилизации вычислительных мощностей могут использоваться для решения очень сложных вариантов задачи коммивояжера всего лишь за несколько суток, т.е. намного быстрее, чем при применении любого отдельно взятого суперкомпьютера.

В приложении Ж приведено много ссылок, по которым можно найти общую текущую информацию об искусственном интеллекте. В оперативном режиме можно получить доступ к огромному объему информации и программного обеспечения. Обсуждению таких тем, как нечеткая логика, нейронные сети, командные интерпретаторы экспертных систем и многих других, посвящены многочисленные группы новостей. Эти группы новостей позволяют не только получать информацию и программное обеспечение, но и передавать свои вопросы и получать ответы от других людей. Авторы настоятельно рекомендуют ознакомиться

с оперативными ресурсами, относящимися к данной главе, которые перечислены в приложении Ж.

Задачи

1.1. Найдите какого-либо человека, который считается либо экспертом, либо специалистом, обладающим очень обширными знаниями в какой-то области (вы сами не должны брать на себя роль эксперта). Проведите собеседование с экспертом и обсудите, насколько успешно экспертные знания этого человека можно было бы смоделировать с помощью экспертной системы, учитывая каждый критерий, приведенный в разделе “Преимущества экспертных систем”.

1.2. Выполните следующие задания.

а) Запишите 10 нетривиальных правил, выражающих знания эксперта, выявленные в процессе решения задачи 1.1.

б) Напишите программу, которая давала бы советы вашему эксперту. Предусмотрите использование проверочных результатов, с помощью которых можно было бы показать, что каждое из десяти правил дает правильный совет. Для упрощения программирования можно предусмотреть ввод данных пользователем с помощью меню.

1.3. Выполните следующие задания.

а) В классической книге Ньюэлла и Саймона *Human Problem Solving* упоминается задача с девятью точками. Соедините девять точек, показанных ниже, четырьмя отрезками прямых, составляющих одну ломаную линию, во-первых, не отрывая карандаш от бумаги и, во-вторых, проведя линию через каждую точку. (*Подсказка.* Линии могут выходить за пределы квадрата, образованного точками.)

o o o
o o o
o o o

б) Объясните, какими рассуждениями вы руководствовались при поиске решения этой задачи (если эти рассуждения были сформулированы явно), и обсудите вопрос о том, могла ли быть экспертная система или программа какого-то другого типа подходящим средством решения подобных задач.

1.4. Напишите программу, позволяющую решать криптоарифметические задачи. Покажите результат решения следующей задачи, где $D = 5$:

$$\begin{array}{rr} & \text{DONALD} \\ + & \underline{\text{GERALD}} \\ & \text{ROBERT} \end{array}$$

1.5. Составьте набор продукционных правил, которые могут применяться при гашении пламени, возникающего в результате горения веществ пяти различных типов, таких как нефть, химикаты и т.д, учитывая тип вещества.

1.6. Выполните следующие задания.

а) Составьте набор продукционных правил для диагностирования отравлений ядами трех различных типов на основании симптомов.

б) Модифицируйте программу так, чтобы после идентификации яда она рекомендовала также лечение.

1.7. Составьте 10 эвристических правил типа `IF-THEN` для планирования отпуска.

1.8. Составьте 10 эвристических правил типа `IF-THEN` для покупки подержанного автомобиля.

1.9. Составьте 10 эвристических правил типа `IF-THEN` для планирования собственного расписания занятий.

1.10. Составьте 10 эвристических правил типа `IF-THEN` для покупки джипа или легкового автомобиля.

1.11. Составьте 10 эвристических правил типа `IF-THEN` для покупки акций или облигаций.

1.12. Составьте 10 эвристических правил типа `IF-THEN` для составления объяснительной по поводу отсутствия на последнем совещании.

1.13. Напишите отчет по результатам изучения одной из современных экспертных систем. Приемлемыми источниками информации на эту тему являются журналы *PCAI* и *IEEE Expert*, а также приложение Ж.

Глава 2

Представление знаний

2.1 Введение

Настоящая и следующая главы посвящены **логике**. Большинство людей считают, что слово “логичный” означает “обоснованный”. Таким образом, если человек рассуждает логично, то его рассуждения обоснованны, поэтому он не допускает поспешных выводов. Но прежде чем перейти к изучению искусственного интеллекта и экспертных систем на основе логики, необходимо дать более строгое определение терминов. Все, кто использует компьютеры, знают, что преимущество компьютеров состоит в том, что они точно выполняют данные им задания. Но и недостаток компьютеров заключается в том, что они действуют в точном соответствии с инструкциями. В настоящее время экспертные системы применяются для оценки кредитоспособности; определяют, должна ли быть проведена ревизия компании налоговым управлением; обеспечивают бесперебойное функционирование атомных электростанций; а также являются ключевыми элементами другой столь же важной деятельности, поэтому необходимо обеспечить, чтобы эти системы действовали чрезвычайно логично и не были подвержены двусмысленным толкованиям. С формальной точки зрения логика — это наука о формировании действительных логических выводов. Это означает, что при наличии необходимого количества истинных фактов вывод всегда должен быть истинным. С другой стороны, если логический вывод недействителен, это означает, что на основании истинных фактов получено ложное заключение (но не стоит пытаться это доказать инспектору дорожной службы, который остановит вас за превышение скорости).

Необходимо также четко провести различие между **формальной логикой** и **неформальной логикой**. Отличительная особенность неформальной логики состоит в том, что ею пользуются в обычной жизни и особенно в адвокатской практике при попытке выиграть в словесном состязании, например, в судебном

разбирательстве. В последнем случае такое состязание обычно не принимает вид ожесточенного обмена репликами, который заканчивается перестрелкой, а выглядит как юридическое доказательство в зале суда, в ходе которого слова, несущие эмоциональную нагрузку, используются для того, чтобы убедить присяжных в том, какая из противоборствующих сторон имеет лучшего адвоката, и тем самым выиграть дело. Сложное логическое доказательство представляет собой цепь логических выводов, в которой одно заключение ведет к другому, и т.д. В суде построение такой цепи может привести к одному из нескольких заключений, которое становится основой судебного решения о виновности, невиновности, невменяемости или необходимости дополнительного расследования, после чего объявляется приговор.

В **формальной логике**, называемой также **символической логикой**, исключительную важность имеет то, как осуществляется логический вывод и как учитываются другие факторы, которые обеспечивают доказательство истинности или ложности окончательного заключения допустимым способом. Идеальным примером компьютерной программы, осуществляющей недействительный символический логический вывод, может служить программа, содержащая программную ошибку. (К счастью, пользователи, потратившие сотни долларов на обновление операционной системы своего компьютера с переходом на новейшую версию, всегда могут получить немедленное удовлетворение, отправив изготовителю ОС отчет об ошибке.) Логика нуждается также в **семантике**, позволяющей придать смысл символам. В формальной логике используется семантика, не основанная на применении слов, несущих эмоциональную нагрузку, как в следующем примере: “Вы предпочитаете пепси-колу или кока-колу?” Вместо этого область применения семантики в формальной логике ограничивается направлением, подобным выбору осмысленных имен для переменных в обычном программировании.

В настоящей главе приведено вводное описание логики, а также даны основные сведения о наиболее широко используемых способах представления знаний. Тематика представления знаний (Knowledge Representation — KR) уже давно считается одним из основных направлений работ в области искусственного интеллекта, поскольку выбор правильного способа представления знаний является не менее значимым фактором, от которого зависит успешное создание системы, чем разработка самого́ программного обеспечения, в котором используются эти знания. С тематикой представления знаний тесно связана не менее важная тематика представления данных, которая рассматривается в такой области компьютерных наук, как проектирование баз данных. Безусловно, базы данных в основном рассматриваются как репозитарии текущих данных, таких как данные инвентарного учета товарно-материальных запасов на складах, данные о кредиторской задолженности, дебиторской задолженности и т.д., а не знаний, но в настоящее время многие компании проводят активную деятельность в направлении **анализа скрытых закономерностей в данных** для извлечения знаний.

Анализ скрытых закономерностей в данных направлен на использование **архивных данных**, находящихся в **хранилищах данных**, для предсказания будущих тенденций. Например, компания может заняться изучением своих данных о сбыте в последние пять лет за декабрь месяц для прогнозирования того, какие товары и в каком количестве следует запасти на складах. Например, специалисты компании с помощью анализа скрытых закономерностей в данных могут обнаружить, что в декабре хорошо продаются рождественские открытки, а поздравления ко дню Святого Валентина мало кого интересуют. Безусловно, этот пример немного надуман, а несколько более реалистичным примером может служить обнаружение того, что одежда в красных и зеленых тонах лучше продается зимой, а не весной (поскольку красный и зеленый цвета ассоциируются с Рождеством), а одежда в коричневых, оранжевых и желтых тонах находит более активный сбыт осенью. Несомненно, об этом догадываются и администраторы магазинов, но анализ скрытых закономерностей в данных может использоваться для получения количественных оценок того, сколько предметов одежды должна закупить торговая компания и когда следует объявить их распродажу в связи с окончанием сезона. Безусловно, истинная ценность анализа скрытых закономерностей в данных состоит в том, что он позволяет обнаружить тенденции, неочевидные для человека, но доступные обнаружению путем анализа огромных объемов исторических данных, которые хранятся в архиве компании. В процессе анализа скрытых закономерностей в данных применяются не только классические статистические методы, но и такие методы искусственного интеллекта, как искусственные нейронные системы, генетические алгоритмы, эволюционные алгоритмы и экспертные системы, не только отдельно взятые, но и в различных комбинациях [94].

Выбор правильных способов представления знаний имеет очень важное значение для экспертных систем по двум причинам. Первая причина состоит в том, что экспертные системы рассчитаны на использование представления знаний определенного типа, основанного на **правилах логики**, называемых способами **логического вывода**. Обычно под термином **умозаключение** подразумевается получение логических заключений на основании фактов. К сожалению, люди не очень хорошо справляются с указанной задачей, поскольку им свойственно путать семантику с самим процессом формирования рассуждений, поэтому им не всегда удается прийти к действительному заключению. Наглядные примеры подобных недостатков часто обнаруживаются при изучении предвыборных плакатов, применяемых политическими противниками. Основой логики, которая используется политиками, являются методы убеждения, не базирующиеся на фактах, а построенные на ненадежных сведениях или применяющие одни и те же факты для достижения полностью противоположных выводов.

Формирование логических выводов — это формальный термин, используемый для обозначения рассуждений специального типа, которые не опираются на семантику (т.е. в них не учитывается смысл слов). Разумеется, в реальном мире невоз-

можно обойтись без учета семантики, но экспертные системы проектируются для проведения рассуждений на основе логики, поэтому не должны подвергаться влиянию той эмоциональной окраски, которая может быть внесена в рассуждения под влиянием семантики. Цель логического вывода состоит в достижении действительного заключения на базе фактов с использованием доказательства в допустимой форме. Еще раз отметим, что в логике термин *доказательство* обозначает формальный способ, в котором применяются факты и правила логического вывода для обоснования действительного заключения. С другой стороны, просматривая телевизионное шоу, в котором тележурналист проводит оживленную дискуссию с человеком, представленным как отставной генерал, бывший дипломат, отстраненный от дел присяжный заседатель или другой подобный деятель, можно быть вполне уверенным в том, что целью этого шоу является не достижение действительных логических заключений, а привлечение интереса телезрителя.

Итак, логическое рассуждение — это формирование действительных логических выводов. Как известно, в реальном мире невозможно обойтись без здравого смысла и вероятностных рассуждений, но такие формы умственной деятельности связаны с неопределенностью, поскольку в действительности ни в чем нельзя быть уверенным на все 100%. Если кто-то скажет, что сейчас небо голубое, несомненно, немного позже оно может стать серым или даже, что еще больше расходится со сказанным, приобрести зеленый оттенок! Рассуждения в условиях неопределенности — очень важная тема, которая будет обсуждаться более подробно в главах 4 и 5.

Вторая причина, по которой представление знаний является важным, состоит в том, что от правильного выбора способа такого представления зависит весь ход разработки, а также эффективность, быстродействие и удобство сопровождения системы. В этом указанное положение полностью аналогично тому положению, которое складывается в обычном программировании, где выбор правильной структуры данных имеет принципиальную значимость при разработке программы. Качественный проект программы всегда начинается с правильного выбора способа представления данных, будь то простые именованные переменные, массивы, связные списки, очереди, деревья, графы, сети или даже такие автономные внешние базы данных, как Microsoft Access, SQL Server или Oracle. В языке CLIPS в качестве средств представления знаний могут использоваться правила, конструкции `deftemplate`, объекты и факты.

В следующей главе будет описано то, как на основе знаний формируются логические выводы в целях выработки действительных заключений и каких распространенных логических ошибок следует избегать. *Логической ошибкой* называется умозаключение, которое на первый взгляд кажется логически обоснованным, но не является таковым. Соблюдение этих требований становится особенно важным в процессе приобретения знаний, в ходе которого проводится собеседование с экспертом-человеком, предоставляющим знания для разрабатываемой эксперт-

ной системы (речь об этом пойдет в главе 6). Прежде всего необходимо отделить истинные знания от семантической окраски, влияние которой может привести к недействительным заключениям. Но при этом не следует слишком много спорить с экспертом по знаниям, предъявлять ему невыполнимые требования, и тем более нельзя добиваться получения тех логических заключений, которые требуются по условиям задания, поскольку это равносильно неудачному завершению проекта!

Как показано в приложении Ж, с применением методов искусственного интеллекта для формирования рассуждений, доказательства теорем и даже обучения логике написано много компьютерных программ.

2.2 Смысл знаний

"Знания", как и "любовь", является одним из тех слов, смысл которого понимает каждый, но затрудняется найти ему определение. Кроме того, трудно найти двух таких людей, которые испытывали бы в любви одинаковые чувства. В действительности единственные проявления истинной любви, которые у всех людей выглядят почти одинаковыми, относятся к любимым кошкам или собакам (приносим свои извинения тем, кто любит змей и тарантулов). Как и любовь, знания имеют много толкований, которые зависят от взглядов того, кто наблюдает за их проявлениями. В качестве взаимозаменяемых со словом "знания" часто используются другие слова, такие как данные, факты и информация [83].

Как правило, в процессе поиска решения задач применяются логические рассуждения и опыт. Общим термином, которым обозначается использование опыта для решения некоторой задачи, является **эвристика**. Уже давно принято называть конкретные примеры эвристического опыта **рассуждениями на основе прецедентов**. Это — один из основных типов рассуждений, применяемых в юридической практике, медицине и автосервисе; с помощью таких рассуждений юристы, врачи и автомеханики пытаются найти решение задачи по данным о встречавшихся ранее аналогичных случаях, называемых **прецедентами** [66]. Безусловно, клиенты часто жалуются на то, что предложенное им решение на основе прецедента обходится слишком дорого, а это несправедливо, поскольку аналогичная задача уже была успешно решена ранее, но они должны представлять себе, как много приходится платить за то, чтобы создать новый прецедент в области медицины, юриспруденции или автосервиса!

Экспертная система может содержать сотни или тысячи небольших фрагментов знаний о прецедентах, на которые она опирается. Каждое правило в экспертной системе может рассматриваться как своего рода микропрецедент, который может способствовать решению задачи или использоваться в цепи логических выводов в надежде на то, что с его помощью удастся получить решение.

Наука о знаниях называется **эпистемологией**. В рамках этой науки рассматриваются характер, структура и происхождение знаний. Некоторые категории эпистемологии показаны на рис. 2.1. Кроме философских разновидностей знаний, сформулированных Аристотелем, Платоном, Декартом, Юмом, Кантом и другими учеными, на этом рисунке представлены две особые разновидности, называемые **априорными** и **апостериорными** знаниями. Латинский термин “a priori” означает предшествующий. Априорные знания предшествуют знаниям, полученным с помощью органов чувств, и не зависят от них. В частности, примерами априорных знаний является утверждение: “Все имеет свою причину” и “Сумма всех углов треугольника на евклидовой плоскости равна 180 градусам”. Априорные знания рассматриваются как универсально истинные, и эти знания невозможно опровергнуть, не впадая в противоречия. К примерам априорных, неопровержимых знаний относятся логические утверждения, математические законы, а также знания, которыми владеют подростки.

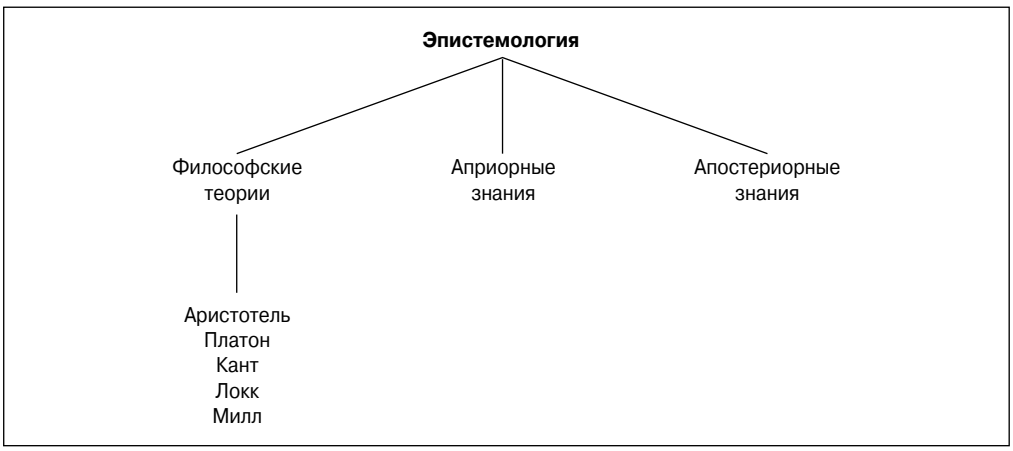


Рис. 2.1. Некоторые категории эпистемологии

Противоположными по отношению к априорным знаниям являются знания, полученные с помощью органов чувств, — апостериорные знания. Истинность или ложность апостериорных знаний, например, представленных с помощью утверждения: “На светофоре горит зеленый свет”, может быть проверена на основании чувственного опыта. Но чувственный опыт не всегда может оказаться надежным, поэтому существует вероятность того, что апостериорные знания будут опровергнуты на основе новых знаний. Однако это не всегда приводит к противоречию. Например, встретив человека с карими глазами, можно поверить в то, что его глаза действительно карие. В дальнейшем, увидев, как этот человек снимает коричневые контактные линзы, после чего обнаруживаются его синие глаза, необходимо просто пересмотреть полученные ранее знания.

Знания могут дополнительно подразделяться на **процедурные**, **декларативные** и **неявные**. Типы процедурных и декларативных знаний соответствуют процедурному и декларативном подходам, которые рассматриваются в главе 1 и более подробно описаны в [10].

Процедурные знания часто называют знаниями о том, как сделать то или другое. К примерам процедурных знаний относятся знания о том, как вскипятить кастрюлю воды. Но человек, который считает, что его знаний, касающихся способов доведения воды до кипения на обычных высотах, будет достаточно, чтобы успешно вскипятить воду на любой высоте над уровнем моря, обнаруживает, что его попытка использовать знания, основанные на здравом смысле, оканчивается неудачей. Но истинной причиной этой неудачи становится не то, что рассуждения начинающего альпиниста основаны на здравом смысле, а то, что у него недостает практического опыта для проведения правильных рассуждений (см. приведенную в конце данной главы задачу, посвященную анализу процесса кипячения воды). Декларативными знаниями называют знания о том, является ли некоторое утверждение истинным или ложным. Термин *декларативный* применяется к знаниям, выраженным в форме декларативных утверждений, подобных следующему: “Не опускайте пальцы в кастрюлю с кипящей водой”. Как показано в [88], разработано много различных способов представления знаний с использованием логики.

Неявные знания иногда называют **подсознательными знаниями**, поскольку они не могут быть выражены с помощью языка. В качестве примера можно указать знания о том, как двинуть рукой. Давая наиболее общий ответ, можно было бы сказать, что движение рукой осуществляется за счет напряжения или расслабления определенных мускулов и сухожилий. Но в таком случае приходится рассматривать этот вопрос на более низком уровне и объяснять, откуда вы знаете, как следует напрягать или ослаблять свои мускулы и сухожилия. В качестве других примеров можно указать ходьбу или езду на велосипеде. А если речь идет о компьютерных системах, то знания, представленные в искусственной нейронной системе, напоминают неявные знания, поскольку обычно нейронная сеть неспособна непосредственно объяснить суть содержащихся в ней знаний, но могла бы приобрести такую способность при наличии соответствующей программы (см. раздел 1.14).

Знания играют в экспертных системах очень важную роль. В действительности можно провести аналогию с приведенным ниже классическим выражением, которое привел Николас Вирт (Nicholas Wirth) в своей оригинальной книге по языку Pascal.

$$\text{Алгоритмы} + \text{Структуры данных} = \text{Программы}$$

Применительно к экспертным системам это выражение выглядит следующим образом:

$$\text{Знания} + \text{Логический вывод} = \text{Экспертные системы}$$

Согласно определению знаний, которое используется в настоящей книге и иллюстрируется на рис. 2.2, знания входят в состав иерархии способов представления информации. На нижнем уровне этой иерархии находится шум, состоящий из информационных элементов, которые не представляют интереса и могут лишь затруднить восприятие и представление данных. На более высоком уровне находятся бесформатные данные, содержащие элементы данных, которые в принципе могут представлять определенный интерес. На следующем уровне находится информация, т.е. обработанные данные, явно представляющие интерес для пользователей. За этим уровнем следует уровень знаний, на котором представлена настолько важная информация, что ее следует надежно хранить и обеспечить выполнение над ней необходимых операций. В главе 1 знания в экспертной системе на основе правил определены как правила, которые активизируются фактами или другими правилами в целях выработки новых фактов или заключений. Заключение рассматривается как конечный продукт цепи рассуждений, называемой *логическим выводом*, но при условии, что эти рассуждения осуществляются в соответствии с формальными правилами. Процесс формирования логических выводов является существенной частью процесса функционирования экспертной системы. Сам термин **формирование логических выводов**, как правило, используется применительно к таким механическим системам, как экспертные системы. А термин *рассуждения* применяется применительно к продуктивным человеческим размышлениям.



Рис. 2.2. Пирамида знаний

Искусственная нейронная система не формирует логические выводы. Вместо этого такая система осуществляет поиск в целях обнаружения в данных основополагающих образов, которые не являются очевидными для человека. По существу искусственная нейронная система представляет собой классификатор образов. Например, способность человека к чтению основана на возможностях распознавания образов, заложенных в нейронной сети мозга, которая была обучена распознаванию образов букв. В другом участке мозга эти образы преобразуются в слова, которые человек мысленно слышит во время чтения, поскольку именно так про-

исходит обучение чтению детей — путем озвучивания слов, произносимых по буквам. В качестве эксперимента можно, например, попытаться перевернуть книгу или газету на 180 градусов и приступить к чтению. Большинство людей на первых порах испытывают затруднения при чтении перевернутого текста, но способны переобучить свою нейронную сеть, применяемую во время чтения, чтобы она распознавала буквы, представленные в необычном ракурсе. Еще в одном классическом психологическом эксперименте людям предлагали носить очки, которые переворачивают изображение, воспринимаемое глазами. Но через несколько дней мозг у таких людей адаптируется, и они начинают снова видеть мир так, как будто его изображение не перевернуто. А в действительности хрусталики в глазах человека проектирует изображение на сетчатку в перевернутом виде, а мозг снова его переворачивает, поэтому мир воспринимается в нормальном виде.

Еще с одним примером гибкости нейронных сетей можно ознакомиться, попытавшись приступить к чтению книги, повернутой на какой-то угол, скажем, на 30° [84]. При этом человек не теряет способности к чтению, просто чтение происходит немного медленнее. Эта задача становится сложнее после поворота книги на 180°. А после определенной практики некоторые люди приобретают способность читать перевернутый текст так же быстро, как и обычный, что показывает поразительную приспособляемость нейронных сетей человека. Аналогичным образом искусственная нейронная система, обученная распознаванию букв, расположенных под различными углами, приобретает способность к распознаванию и чтению текста, который демонстрируется в условиях, отличающихся от обычных. Чем больше число различных вариантов поворота, на примере которых была обучена сеть, тем быстрее и точнее она будет работать. Кроме того, обучение искусственной нейронной системы чтению различных стилей почерка позволяет использовать эту систему для чтения рукописных текстов, оформленных с помощью еще более разнообразных стилей, по аналогии с тем, как люди читают рукописи различных авторов.

Термином **факты** обозначается информация, рассматриваемая как надежная. Экспертные системы формируют логические выводы с использованием фактов. Факты, ложность которых будет продемонстрирована в дальнейшем, могут быть исключены с помощью средств поддержания истинности системы CLIPS; в ходе этого автоматически изымаются все выводы, правила и другие факты, сформированные на основании ложного факта. Кроме того, экспертные системы могут выполнять следующие действия: во-первых, отделять данные от шума, во-вторых, преобразовывать данные в информацию и, в-третьих, преобразовывать информацию в знания. В экспертной системе, которая рассчитана на получение фактов, чрезвычайно опасно использовать бесформатные данные, поскольку надежность полученных в результате заключений может оказаться полностью неприемлемой. Безусловно, и в экспертных системах оправдывается поговорка: “Мусор на входе — мусор на выходе”, которой руководствуются программисты. Но, безусловно,

кто-то может сознательно принять решение о применении информационного мусора для поддержки конкретного рабочего списка правил.

В качестве примера применения представленных выше понятий рассмотрим следующую последовательность из 24 цифр:

137178766832525156430015

При отсутствии знаний об этой последовательности она может показаться просто проявлением шума. Но если есть основания полагать, что эта последовательность имеет смысл, или это достоверно известно, то указанная последовательность рассматривается как данные. При определении того, что́ является данными и что́ является шумом, вполне можно руководствоваться старой рекомендацией, касающейся сельского хозяйства: “Сорняком следует считать все, что выросло вопреки вашему желанию”.

Определенные знания могут относиться к тому, как нужно преобразовывать данные в информацию. Например, следующий алгоритм показывает, как обработать приведенные выше данные для получения информации:

```
Разбить представленные цифры на пары.
Игнорировать те из полученных двухзначных чисел, которые
   меньше 32.
Подставить вместо оставшихся двухзначных чисел символы
   кода ASCII.
```

Применение этого алгоритма к приведенным выше 24 цифрам приводит к получению следующей информации:

```
GOLD 438+
```

После этого к полученной информации можно применить знания. Например, допустим, что существует такое правило:

```
IF цена на золото меньше 500
   и цена растет (+)
THEN
   покупать золото
```

Очевидно, что на рис. 2.2 это явно не показано, но **экспертные знания** представляют собой специализированную разновидность знаний и навыков, которыми обладают эксперты. Экспертные знания могут относиться к показанным на этом рисунке уровням знаний, метазнаний и мудрости. Хотя весьма специализированные знания можно найти в таких общедоступных источниках информации, как книги и статьи, недостаточно просто прочитать книгу, чтобы стать экспертом. Например, в медицинских учебниках можно найти подробную информацию о том, как следует проводить хирургические операции. Но вряд ли найдется такой человек, который согласится подвергнуться хирургической операции на головном

мозге, если кто-то постучится в его дверь и заявит, что окончил заочные курсы и готов предложить свои услуги по сниженным ценам. На это вряд ли может даже соблазнить бесплатный набор ножей Ginsu-2000 (немного попорченный после проведения очередной хирургической операции на головном мозге).

Экспертные знания — это неявные знания и навыки эксперта, которые должны быть извлечены и преобразованы в явные с тем, чтобы их можно было представить в экспертной системе. Причина, по которой знания являются неявными, состоит в том, что истинный эксперт владеет этими знаниями настолько хорошо, что они превратились в его вторую натуру и не требуют размышлений. В качестве примера можно указать, что после окончания медицинского училища практиканты служат в лечебном учреждении примерно один год, работая, как правило, 80 или больше часов в неделю, до тех пор, пока не приобретают способность выполнять медицинские процедуры даже без размышлений. Разумеется, такая организация обучения специалистов подвергалась критике, но она позволяет усваивать знания настолько глубоко, что они становятся второй натурой. (Безусловно, пациентам иногда перед проведением очередной процедуры не мешает также спросить практиканта, как долго ему удалось поспать на этой неделе.) Почти на самом верхнем уровне иерархии (см. рис. 2.3) над уровнем знаний находится уровень **метазнаний**. Префикс **мета** означает “свыше” или “дальше”.

Метазнания представляют собой знания об обычных и экспертных знаниях. Безусловно, экспертная система может быть спроектирована с учетом знаний о нескольких различных проблемных областях, но, как правило, это нежелательно, поскольку в результате система становится менее качественно определенной. Опыт показывает, что наиболее успешно работают такие экспертные системы, применение которых ограничивается наименьшей проблемной областью из всех возможных. Например, если экспертная система спроектирована для выявления заболеваний, вызванных бактериями, то нет смысла применять ее также для диагностирования неисправностей в автомобилях. В качестве практического примера можно указать, что сами врачи специализируются только в одной небольшой области, а не во всей медицине. Даже семейные врачи (в качестве которых чаще всего выступают терапевты) направляют своих пациентов в случае необходимости к соответствующему специалисту.

В экспертных системах *онтология* представляет собой метазнания, которые описывают все, что известно о рассматриваемой предметной области. В идеальном случае онтология должна быть представлена в формальном виде для того, чтобы можно было легко обнаруживать несовместимости и несоответствия. Для построения онтологий может применяться целый ряд бесплатных и коммерческих инструментальных средств. Построение онтологии должно быть закончено до реализации экспертной системы, поскольку в противном случае может потребоваться пересматривать правила по мере поступления дополнительной информации о данной предметной области, что приводит к повышению издержек, увеличению

продолжительности разработки и возрастанию вероятности появления программных ошибок.

Например, экспертная система может иметь базы знаний о ремонте легковых автомобилей компании GM (General Motors), джипов (Sport Utility Vehicle — SUV) GM и дизельных грузовиков GM. В зависимости от того, к какому типу относится автомобиль, требующий ремонта, должна использоваться соответствующая база знаний. В связи с необходимостью снижения потребности в памяти и повышения быстродействия было бы неэффективно держать в памяти одновременно все базы знаний, поскольку в процессе эксплуатации rete-сети непрерывно происходит модификация в памяти всех правил, находящихся в сети. Кроме того, могут возникать конфликты, если антецеденты какого-либо правила для грузовиков и легковых автомобилей содержат одинаковый шаблон, а заключения являются различными. Например, если датчик измерения уровня топлива показывает, что бак пуст, то экспертная система для легковых автомобилей может дать указание: "Заполнить бак бензином", а экспертная система для грузовиков — указание: "Заполнить бак дизельным топливом". Такая путаница, в результате которой бензобак легкового автомобиля будет заполнен дизельным топливом или бак грузовика — бензином, весьма нежелательна. Кроме того, работа экспертной системы замедляется при возрастании количества правил в системе, поскольку rete-сеть становится больше. А метазнания могут использоваться для принятия решения о том, какая база знаний должна быть загружена в память, а также служить в качестве общего руководства по проектированию и сопровождению самой экспертной системы и ее онтологии.

Наконец, вершиной всех знаний является **мудрость**, рассматриваемая в ее философском толковании. Мудрость — это метазнания, позволяющие определять наилучшие цели в жизни и находить пути их достижения. Например, одно из правил мудрости можно выразить следующим образом:

```
IF мне удастся заработать достаточно денег для того, чтобы
   моя семья ни в чем не нуждалась
THEN я уволюсь с работы и буду наслаждаться жизнью
```

Объем работ по искусственному интеллекту на основе инженерии знаний, достигающих уровня мудрости, постоянно возрастает. Однако человечество и так сформулировало чрезвычайно много мудрых мыслей, но прислушиваются к ним лишь единицы. В настоящей книге мы ограничимся рассмотрением систем, основанных на знаниях, и оставим проблематику создания систем на основе мудрости для политических деятелей и других экспертов того же рода.

2.3 Продукции

До настоящего времени предложен целый ряд различных методов представления знаний. К ним относятся правила, семантические сети, фреймы, сценарии, логика, концептуальные схемы и др. В частности, было предложено много языков представления знаний, таких как классический язык KL-ONE (Knowledge Language ONE — язык знаний номер один) и происходящий от него язык на основе фреймов CLASSIC [8]. Кроме того, были предложены многие другие языки, включая языки на основе визуальных средств.

Как было описано в главе 1, в качестве основы базы знаний в экспертных системах широко используются правила, поскольку преимущества правил намного перевешивают их недостатки.

Одной из формальных систем обозначений, применяемых для определения продукций, является нормальная форма Бэкуса–Наура (Backus-Naur form — BNF). Эта система обозначений представляет собой **метаязык**, применимый для определения **синтаксиса** любого языка. Синтаксис определяет форму, а **семантика** обозначает смысл. Метаязык — это язык, предназначенный для описания других языков. Поскольку префикс "мета" обозначает "свыше" или "дальше", метаязык находится на более высоком уровне по сравнению с обычным языком.

Языки подразделяются на несколько типов, таких как естественные языки, логические языки, языки математики, компьютерные языки и т.д. Ниже приведено продукционное правило, в котором система обозначений на основе нормальной формы Бэкуса–Наура используется для формулировки простого правила английского языка, согласно которому предложение (*sentence*) состоит из подлежащего (*subject*), сказуемого (*verb*), за которыми следует знак пунктуации, обозначающий конец предложения (*end-mark*).

```
<sentence>::= <subject> <verb> <end-mark>
```

В этом правиле **угловые скобки** (<>) и символ `::=` представляют собой символы метаязыка, а не определяемого языка. Символ `::=` означает "определено как" и представляет собой эквивалент стрелки (→), применяемой в нормальной форме Бэкуса–Наура. Как описано в главе 1, в продукционных правилах используется стрелка.

Термы, заключенные в угловые скобки, принято называть **нетерминальными символами** (nonterminal). Нетерминальный символ — это переменная, представляющая другой терм. В свою очередь, в качестве подобного "другого" терма может использоваться нетерминальный символ или **терминальный символ** (terminal). Терминальный символ не может быть заменен каким-либо иным символом и поэтому представляет собой константу.

Нетерминальный символ `<sentence>` является специальным, поскольку относится к числу **начальных символов**, применяемых для определения других

символов. В определениях языков программирования начальный символ обычно носит имя <program>. Ниже приведено продукционное правило, которое указывает, что предложение состоит из подлежащего, за которым следует сказуемое, а в конце находится маркер конца предложения.

```
<sentence> → <subject> <verb> <end-mark>
```

А следующие правила позволяют раскрывать значения нетерминальных символов, поскольку указывают терминальные символы, которыми они могут быть заменены. В этом метаязыке **вертикальная черта** означает "или".

```
<subject> → I | You | We
<verb> → left | came
<end-mark> → . | ? | !
```

Все возможные предложения языка, т.е. продукции, могут быть произведены путем последовательной замены каждого нетерминального символа соответствующими ему нетерминальными или терминальными символами, взятыми из правой части правил, до тех пор, пока не будут устранены все нетерминальные символы. Некоторые продукции рассматриваемого языка приведены ниже.

```
I left.
I left?
I left!
You left.
You left?
You left!
We left.
We left?
We left!
```

Ряд терминальных символов называется **строкой** символов языка. Если строка может быть получена из начального символа путем замены нетерминальных символов с применением определяющих их правил, то строка называется действительным **предложением**. Например, такие строки, как "We", "WeWe" и "leftcamecame", являются действительными строками языка, но не действительными предложениями.

Грамматикой называется полный набор продукционных правил, который однозначно определяет язык. Безусловно, приведенные выше правила определяют некоторую грамматику, но она является очень ограниченной, поскольку допускает производство слишком малого количества возможных продукций. Более сложная грамматика может, в частности, предусматривать использование прямых дополнений, как показано в следующих продукциях:

```
<sentence> → <subject> <verb> <object> <end-mark>
<object> → home | work | school
```

Хотя эта грамматика является действительной, она также слишком проста для практического использования. А ниже показана грамматика, в большей степени применимая на практике, в которой в целях упрощения был исключен маркер конца.

```
<sentence> → <subject phrase> <verb> <object phrase>
<subject phrase> → <determiner> <noun>
<object phrase> → <determiner> <adjective> <noun>
<determiner> → a | an | the | this | these | those
<noun> → man | eater
<verb> → is | was
<adjective> → dessert | heavy
```

Нетерминальный символ `<determiner>` используется для замены конкретного члена предложения. Используя эту грамматику, можно вырабатывать предложения, подобные приведенным ниже.

```
the man was a dessert eater
an eater was the heavy man
```

В качестве графического представления предложения, декомпонованного на все терминальные и нетерминальные символы, применявшиеся для вывода этого предложения, используется **дерево синтаксического анализа**, или **дерево вывода**. На рис. 2.3 показано дерево синтаксического анализа для предложения "`the man was a heavy eater`" (этот человек любил поесть). Это предложение является действительным, а строка "`man was a heavy eater`" не представляет собой действительное предложение, поскольку в ней отсутствует определяющее слово (`determiner`) в группе подлежащего (`subject phrase`). Предпринимая попытку определить, соответствует ли оператор программы действительному синтаксису языка, компилятор создает дерево синтаксического анализа.

Дерево, приведенное на рис. 2.3, показывает, что предложение "`the man was a heavy eater`" может быть получено из начального символа путем применения подходящих продукционных правил. Ниже показаны этапы процесса создания рассматриваемого предложения; **двойные стрелки** (=>) указывают на результаты применения продукционных правил.

```
<sentence> => <subject phrase> <verb> <object phrase>
<subject phrase> => <determiner> <noun>
<determiner> => the
<noun> => man
<verb> => was
<object phrase> => <determiner> <adjective> <noun>
<determiner> => a
<adjective> => heavy
<noun> => eater
```

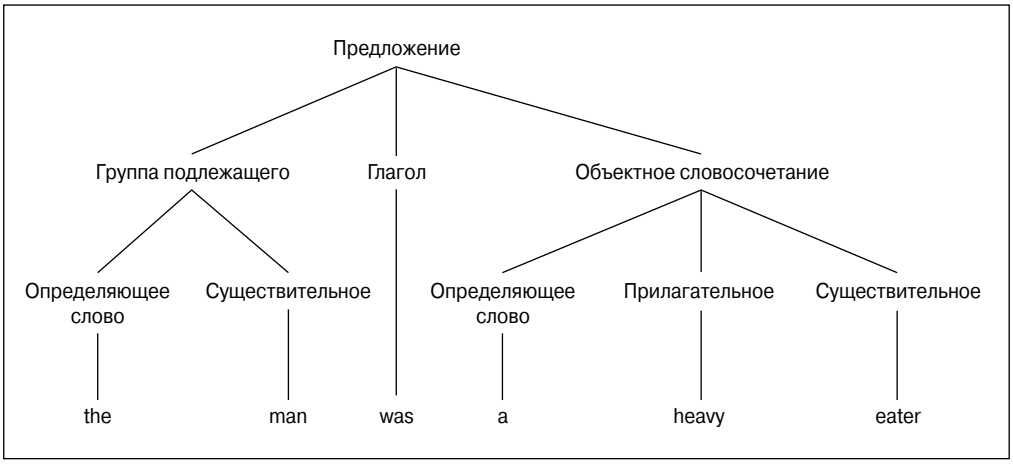


Рис. 2.3. Дерево синтаксического анализа предложения

Альтернативный способ использования продукционных правил состоит в формировании действительных предложений путем подстановки всех допустимых терминальных символов вместо нетерминальных символов, как было описано выше. Безусловно, смысл имеют не все создаваемые при этом продукции, в частности "`the man was the dessert`" (разумеется, речь не идет о том, что каннибалы нашли бы смысл в этом предложении).

Для распознавания структуры предложения очень хорошо подходят конечные автоматы (Finite State Machine — FSM). Например, конечные автоматы используются в компиляторах, которые транслируют исходный код компьютерных языков для **синтаксического анализа** и разбиения исходного кода на меньшие смысловые единицы, называемые **лексемами**. В таких приложениях, как компиляторы, транслирующие исходный код в код на языке ассемблера, и программы, применяемые для точного распознавания речи, невозможно обойтись без синтаксического анализа и использования конечных автоматов. А со времени появления языка Java термин **компилятор** приобрел более широкое значение и стал обозначать средство трансляции исходного кода не только в код ассемблера, но и в независимый от платформы байт-код (эти функции выполняет компилятор javac). Впервые такая возможность появилась с введением в действие программного обеспечения языка Pascal в 1970-х годах, поскольку байт-код этого языка мог эксплуатироваться на любом микропроцессоре. Программное обеспечение, позволяющее преобразовывать код на языке Pascal в байт-код, правильнее считать интерпретатором, а не компилятором, поскольку этот байт-код не содержит команд языка конкретного компьютера, подобных командам языка ассемблера. Но интерпретаторы обладают меньшим быстродействием по сравнению с компиляторами, поэтому разработчи-

ки языка Pascal решили, что лучшей рекламой для программного обеспечения служит его обозначение как компилятора, а не интерпретатора.

С оперативной демонстрационной версией конечного автомата, предложенной компанией Xerox, можно ознакомиться по адресу `http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fsinput.html`. С примерами таких конечных автоматов, как автомат по продаже безалкогольных напитков, можно ознакомиться, перейдя по ссылкам, которые представлены по адресу `http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fsexamples.html`. Практически исчерпывающий справочник по конечным автоматам приведен по адресу `http://odur.let.rug.nl/alfa/fsa_stuff/`.

Безусловно, конечные автоматы успешно действуют применительно к грамматике с сокращенным множеством символов, например, если в число этих символов входят цифры от `0` до `9` или буквы алфавита, но при их использовании в таких областях, как распознавание речи, где могут обнаруживаться неоднозначности, возникают проблемы. Например, рассмотрим следующие два предложения:

```
(1) No one has let us read
    и
(2) No one has lettuce red
```

В предложении (1) группа людей жалуется, что им не дают возможности читать, а в предложении (2) говорится о том, что ни у кого нет красного салата. В одном из удобных способов, позволяющих однозначно отличить друг от друга такие похожие слова и словосочетания, как “`lettuce`” и “`let us`”, “`read`” и “`red`”, используется **скрытая марковская машина** (Hidden Markov Machine — HMM), в которой вероятности присваиваются действиям, осуществимым в конечном автомате. Скрытая марковская машина позволяет рассмотреть всю структуру предложения или проанализировать дополнительные предложения, чтобы выявить правильный контекст (либо прочитав всю книгу, в которой встретилось предложение, либо проведя поиск названий овощей в каталогах овощных магазинов). Безусловно, экспертные системы не являются наиболее подходящим вариантом программного обеспечения, с помощью которого могла бы быть создана скрытая марковская машина, но они могут использоваться в качестве внешнего интерфейса для системы распознавания речи, для того чтобы в дальнейшем такая экспертная система, как CLIPS, могла применяться для инициирования соответствующих правил.

В действительности в языке CLIPS предусмотрена возможность легко связывать с помощью интерфейса код на языке C или C++, поэтому программное обеспечение скрытой марковской машины, написанное на C или C++, может вызываться полностью аналогично любой другой функции, определяемой ключевым словом языка CLIPS. Одной из сильных сторон языка CLIPS является то,

что этот язык — расширяемый, и пользователь может легко вводить новые ключевые слова на этапе компиляции для достижения наибольшей эффективности. Кроме того, благодаря наличию объектно-ориентированных средств языка COOL для расширения возможностей языка CLIPS с использованием всей мощи средств множественного наследования могут применяться объекты. Другое программное обеспечение, такое как искусственные нейронные системы, генетические алгоритмы и прочие программы, написанные на C или C++, может вводиться либо в левые части правил, для инициирования правил, либо в правые части правил, для обеспечения вывода. Из программы на языке CLIPS можно, например, обращаться к синтезатору речи, эффекторам и актюаторам робота, определив с помощью ключевых слов соответствующие функции. Исходный код языка CLIPS представлен в общее пользование, а это означает, что компиляция вызовов новых ключевых слов в программе CLIPS может осуществляться без потери быстродействия, которое обнаруживается при использовании других экспертных инструментальных средств, для которых исходный код не предоставлен.

2.4 Семантические сети

Семантические сети, или просто сети, — это классический способ представления пропозициональной информации, используемый в искусственным интеллекте (`http://www.pcai.com/web/T6110H2/R6.o1.h8/pcai.htm`), поэтому семантические сети иногда называют **пропозициональными сетями**. Как было описано выше, пропозициональным утверждением (или высказыванием) называется предложение, которое может быть либо истинным, либо ложным, такое как “все собаки — млекопитающие” и “треугольник имеет три стороны”. Высказывания имеют форму декларативных знаний, поскольку в них утверждаются факты. С точки зрения математики семантическая сеть представляет собой помеченный ориентированный граф. Высказывание всегда является либо истинным, либо ложным и называется **атомарным**, так как его истинностное значение не подлежит дальнейшей декомпозиции. Здесь термин *атомарный* применяется в классическом смысле, который применялся для обозначения неделимого объекта в трудах древнегреческих ученых. В отличие от этого, нечеткие утверждения, описанные в главе 5, не должны обязательно быть только истинными или только ложными.

Семантические сети впервые были разработаны для исследований в области искусственного интеллекта как способ описания человеческой памяти и языка Квиллианом (Quillian) в 1968 году. Квиллиан использовал семантические сети для анализа смысла слов в предложениях. (Обратите внимание на то, что выявление смысла предложения не сводится к синтаксическому анализу и разбиению предложения на лексемы и лексические структуры по такому принципу, который осуществляется с использованием конечных автоматов и скрытых марковских ма-

шин, рассматриваемых в предыдущем разделе.) С тех пор семантические сети успешно применялись для решения многих задач, связанных с представлением знаний. Понимание смысла, достигаемое с помощью семантических сетей, позволяет выйти за рамки возможностей программного обеспечения простых экспертных систем или искусственного интеллекта для устранения неоднозначности. В следующей главе, посвященной логическому выводу, будет показано, как экспертные системы выводят из фактов заключения, которые могут использоваться другими правилами в цепи логического вывода до тех пор, пока не будет достигнуто действительное заключение. А если не учитывается семантика, то работа экспертных систем может оканчиваться неудачей, как и размышления человека, сбитого с толку из-за обнаружившейся неоднозначности.

Структура семантической сети отображается графически с помощью **узлов** и соединяющих их **дуг**. Узлы иногда именуются **объектами** а дуги — **связями**, или **ребрами**.

Связи в семантической сети применяются для представления отношений, а узлы, как правило, служат для представления физических объектов, концепций или ситуаций. На рис. 2.4, *а* показана обычная сеть (фактически ориентированный граф), в которой связи обозначают авиационные маршруты, проложенные между городами. Узлы обозначены кружками, а связи — линиями, соединяющими узлы. Стрелки показывают направление, или ориентацию, самолетов, совершающих полеты (именно поэтому граф, на котором указаны направления, называется *ориентированным*). На рис. 2.4, *б* связи показывают отношения между членами некоторой семьи. Отношения имеют для семантических сетей исключительно важное значение, поскольку предоставляют базовую структуру для организации знаний. Знания, заданные без учета отношений, превращаются просто в коллекцию несвязанных фактов. А если заданы отношения, то знания приобретают связную структуру, исследование которой позволяет выводить логическим путем другие знания. Например, на основании рис. 2.4, *б* можно сделать вывод, что Анна и Билл — бабушка и дедушка Джона, даже несмотря на то, что на этом рисунке нет явно заданной связи, обозначенной как “`grandfather-of`”.

Семантические сети иногда называют **ассоциативными сетями**, поскольку одни узлы в таких сетях ассоциированы, или связаны, с другими. В действительности в оригинальной работе Квиллиана человеческая память моделировалась как ассоциативная сеть, в которой понятия были представлены в виде узлов, а связи показывали, как соединены эти понятия друг с другом. Согласно указанной модели, если происходит стимуляция одного узла в результате чтения слов в предложении, то происходит активизация связей этого узла с другими узлами, и такая активность распространяется по сети. Если же достаточную активизацию получает другой узел, то в обладающем сознанием разуме всплывает концепция, представленная этим узлом. Например, очевидно, что человек знает тысячи слов,

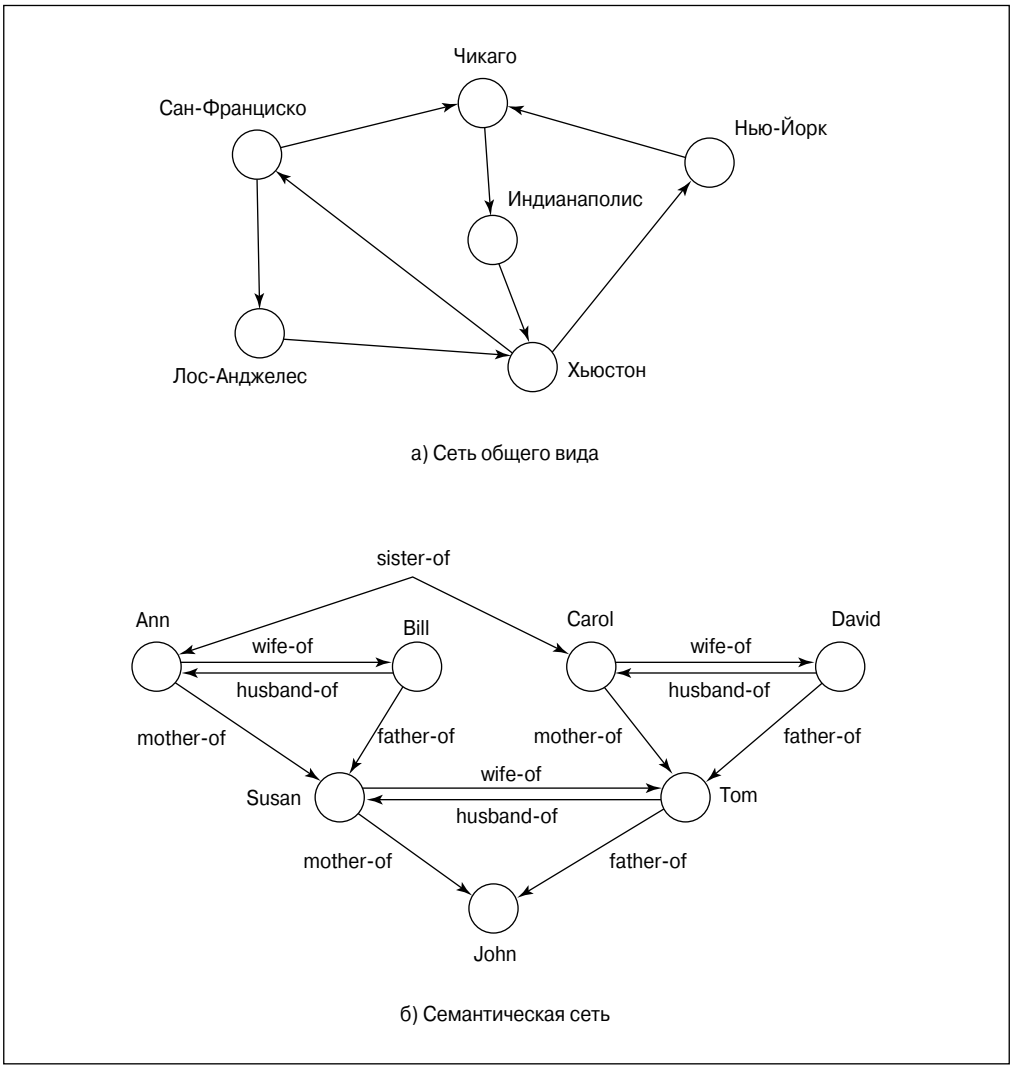


Рис. 2.4. Примеры двух типов сетей

но в его сознании отражаются только слова того конкретного предложения, которое он читает.

Как оказалось, во многих разных способах представления знаний особенно полезными являются отношения определенных типов. Поэтому при изучении различных задач принято использовать именно эти типы, а не определять каждый раз новые отношения. К тому же применение общепринятых типов позволяет другим людям проще понять, что описано в незнакомой для них сети.

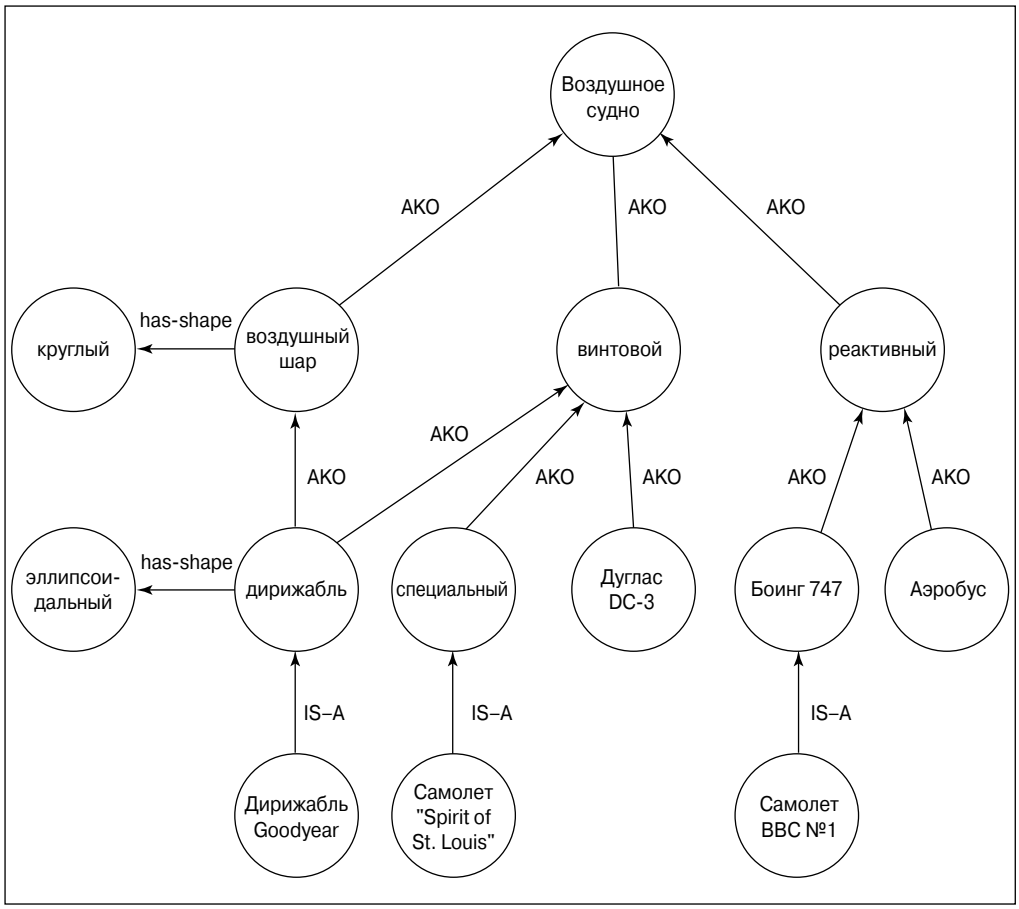


Рис. 2.5. Семантическая сеть со связями IS-A и A-KIND-OF (AKO)

К двум таким связям широко используемых типов относятся связи **IS-A** и **A-KIND-OF** (связь последнего типа иногда записывается как **AKO**). На рис. 2.5 показана семантическая сеть, в которой используются такие связи. На данном рисунке связь IS-A означает “является экземпляром” данного класса и указывает на конкретный экземпляр некоторого класса. В этом контексте понятие **класса** близко к математическому понятию множества, поскольку служит для обозначения группы объектов. Тем не менее множество может содержать элементы любого типа, а объекты в классе связаны определенными отношениями друг с другом, т.е. не могут быть взяты произвольно. Например, может быть определено множество, состоящее из следующих элементов:

$$\{3, \text{eggs}, \text{blue}, \text{tires}, \text{art}\}$$

Но элементы этого множества не связаны какими-либо обычными отношениями. С другой стороны, в семантической сети может быть задан класс, состоящий из самолетов, поездов и автомобилей. Объекты этого класса связаны друг с другом, поскольку все они представляют собой некоторые средства транспорта.

В данном случае связь AKO используется для обозначения отношения одного класса с другим. Связь AKO не может служить для обозначения отношений конкретных объектов; для этого предназначена связь IS-A. Связь AKO определяет отношение между отдельным классом и родительским классом (или классами), применительно к которому этот отдельный класс является дочерним.

Указанные отношения можно трактовать и так, что связь AKO определяет отношение между одними **родовыми** узлами и другими родовыми узлами, а связь IS-A определяет отношение между экземпляром, или **отдельным объектом**, и родовым классом. Следует отметить, что на рис. 2.5 более общие классы находятся в верхней части, а более конкретные классы — в нижней части. Более общий класс, на который указывает стрелка AKO, называется **суперклассом**. Если суперкласс имеет связь AKO, указывающую на другой узел, то он вместе с тем представляет собой класс суперкласса, на который указывает стрелка AKO. Иной способ представления таких отношений заключается в том, что связь AKO направлена от **подкласса** к классу. Вместо связи AKO иногда используется связь **ARE**, в которой ARE читается как обычное слово “are” (есть).

Все объекты класса должны иметь один или несколько общих **атрибутов**. Каждый атрибут имеет **значение**. Комбинация атрибута и значения называется **свойством**. Например, дирижабль имеет такие атрибуты, как размеры, вес, форма и цвет. Значением атрибута формы является эллипсоид. Иными словами, дирижабль имеет такое свойство, как эллипсоидальная форма. В семантических сетях можно также найти связи других типов. Связь **IS-A** определяет значение. Например, самолет, на котором летает Президент, приобретает атрибут “Самолета ВВС номер один”. А если Президент летит на вертолете, такой вертолет IS-A (является) “Вертолетом ВВС номер один”. Связь **CAUSE** выражает причинные знания. Например, горячий воздух CAUSE (становится причиной) того, что воздушный шар поднимается.

Рекламный дирижабль компании Goodyear является дирижаблем, а дирижабли имеют эллипсоидальную форму; из этого следует, что дирижабль Goodyear является эллипсоидальным. Повторение характеристик узла в его потомках называется **наследованием**. Если нет более определенного свидетельства, позволяющего утверждать обратное, то предполагается, что все элементы некоторого класса наследуют все свойства суперклассов этого класса. Например, воздушные шары имеют круглую форму. Но класс дирижаблей имеет связь, указывающую на узел, обозначающий эллипсоидальную форму, поэтому явно заданная связь рассматривается как имеющая более высокий приоритет. Наследование представляет собой полезное средство в представлении знаний, поскольку позволяет избавиться от

необходимости повторно указывать общие характеристики. Связи и наследование являются основой эффективных способов представления знаний, поскольку дают возможность показывать многие сложные отношения с помощью нескольких узлов и связей. Ниже в этой главе будут описаны объектно-ориентированные возможности языка CLIPS, которые реализуются с использованием встроенного в него языка COOL. Эти два языка позволяют создавать экспертные системы, используя правила, объекты или комбинации правил и объектов, связанных полноценными отношениями множественного наследования.

2.5 Тройки “объект–атрибут–значение”

Одна из проблем, связанных с применением семантических сетей, состоит в том, что не предусмотрены стандартные определения для имен связей. Например, в некоторых книгах связи IS-A используются для представления и общих, и индивидуальных отношений. Это означает, что связь IS-A применяется для представления такого же смысла, какой представляют обычные слова “is a” (является), а также может применяться вместо связи AKO.

Еще одной широко применяемой связью является HAS-A, которая устанавливает отношение между классом и подклассом. Связь HAS-A направлена в сторону, противоположную по отношению к связи AKO, и часто используется для обозначения отношения между одним объектом и частью другого объекта, например, как показано ниже.

```
автомобиль HAS-A двигатель
автомобиль HAS-A шины
автомобиль IS-A автомобиль производства компании Ford
```

Более определенно можно указать, что связь IS-A устанавливает отношение между значением и атрибутом, а связь HAS-A — между объектом и атрибутом.

Такие три понятия, как *объект*, *атрибут* и *значение*, встречаются вместе настолько часто, что иногда обнаруживается возможность создать упрощенную семантическую сеть с использованием только этих понятий. Чтобы охарактеризовать все знания, представленные в семантической сети, можно воспользоваться **тройкой “объект–атрибут–значение”** (**object-attribute-value — OAV**), или просто **триплетом**; такие триплеты использовались в экспертной системе MYCIN, предназначенной для диагностирования инфекционных заболеваний. Представление в виде тройки OAV является удобным способом оформления списка имеющихся знаний в форме таблиц, а также позволяет легко преобразовать полученную таблицу в компьютерный код по методу индукции правил. Пример таблицы с тройками OAV показан в табл. 2.1. *Индукция* — это мощный логический метод, который часто используется неправильно. В качестве классического примера можно указать такую ситуацию, когда нерадивого программиста спрашивают, по-

чему он не создает резервные копии данных, хранящихся на жестком диске своего компьютера. Обычно лентяи дают при этом ошибочный индуктивный ответ, состоящий в том, что жесткий диск на этом компьютере еще никогда не ломался, поэтому по индукции такая авария никогда не произойдет. (Такой способ "рассуждений" широкой публики очень нравится биржевым маклерам.) Дополнительные сведения об индуктивном логическом программировании приведены в [5].

Таблица 2.1. Таблица со списком троек "объект–атрибут–значение" (OAV)

Объект	Атрибут	Значение
apple	color	red
apple	type	mcintosh
apple	quantity	100
grapes	color	red
grapes	type	seedless
grapes	quantity	500

Тройки "объект–атрибут–значение" лежат в основе особенно полезного способа представления фактов и шаблонов, применяемых для согласования с фактами в антецеденте правила. Семантическая сеть для подобной системы состоит из узлов, представляющих объекты, атрибуты и значения, соединенных связями HAS-A и IS-A. Если должен быть представлен только один объект и применять отношения наследования не требуется, то может оказаться приемлемым еще более простое представление, называемое **парой "атрибут–значение"** (**attribute-value — AV**), или просто **парой**.

2.6 Язык PROLOG и семантические сети

Семантические сети могут быть легко преобразованы в программу на языке PROLOG. Например, ниже приведены операторы PROLOG, которые представляют некоторую часть отношений из семантической сети, приведенной на рис. 2.5.

```
is_a(goodyear_blimp,blimp).
is_a(spirit_of_st_louis,special).
has_shape(blimp,ellipsoidal).
has_shape(balloon,round).
```

Обратите внимание на то, что конец оператора PROLOG обозначается точкой (.).

Основные сведения о языке PROLOG

Каждый из приведенных выше операторов PROLOG представляет собой **предикативное выражение** (или просто предикат), поскольку эти операторы основаны на логике предикатов. Однако PROLOG нельзя назвать настоящим языком логики предикатов, поскольку он является компьютерным языком с исполняемыми операторами. В языке PROLOG предикативное выражение состоит из имени предиката, такого как `is_a`, за которым следуют от нуля или больше параметров, заключенных в круглые скобки и разделенных запятыми. Ниже приведены некоторые примеры предикатов PROLOG. Комментарии обозначаются точками с запятой и игнорируются машиной PROLOG.

```
color(red).                ; то, что цвет — красный, является фактом
mother(pat,ann).           ; Пэт — мать Энн
parents(jim,ann,tom)       ; Джим и Энн — родители Тома
surrogatemother(pat,tom).  ; Пэт — суррогатная мать Тома
```

Назначение предикатов с двумя параметрами проще понять, представив себе, что они состоят из имени предиката, за которым следует первый параметр. С другой стороны, смысл предикатов, имеющих больше двух параметров, необходимо определять явно, как показано в примере предиката `parents`. А при использовании семантических сетей возникает другая сложность, поскольку они позволяют представлять в основном только двоичные отношения в силу того, что линия, обозначающая связь, имеет только два конца. Таким образом, предикат `parents` нельзя представить графически с помощью одного ориентированного ребра, поскольку в определении предиката имеются три параметра. А попытка реализовать идею, согласно которой Том и Энн должны быть представлены в одном родительском узле, а Джим — в другом, приводит к новым осложнениям. Дело в том, что при этом исключается возможность использовать родительский узел для определения других бинарных отношений, таких как `mother-of`, поскольку Тома никак нельзя считать матерью.

Предикаты могут быть также представлены с помощью таких отношений, как IS-A и HAS-A, например, следующим образом:

```
is_a(red,color).
has_a(john,father).
has_a(john,mother).
has_a(john,parents).
```

Обратите внимание на то, что эти предикаты `has_a` не выражают тот же смысл, как и раньше, поскольку отец, мать и в целом родители Джона явно не

названы. Для того чтобы указать имена отца, матери и отдельно каждого из родителей, необходимо добавить несколько дополнительных предикатов, как показано ниже.

is_a(tom,father).
is_a(susan,mother).
is_a(tom,parent).
is_a(susan,parent).

Но даже эти дополнительные предикаты не выражают тот же смысл, что и первоначальные предикаты. Например, мы знаем, что Джон имеет отца, а Том является отцом, но из этого не следует, что Том — отец Джона.

Все приведенные выше операторы фактически представляют собой описания фактов в языке PROLOG. Программы на языке PROLOG состоят из фактов и правил, заданных в общей форме **целей**:

$$p\text{:-}\ p_1, p_2, \ldots, p_N.$$

В этом операторе терм p представляет собой голову правила, а термы p_k выполняют роль **подцелей**. Как правило, выражение, применяемое в форме такого оператора, является хорновским выражением, которое утверждает, что цель p, заданная в голове выражения, выполняется тогда и только тогда, когда выполнены все подцели. Исключение из этого правила возникает только в том случае, если используется специальный предикат, обозначающий отказ. Предикат с обозначением отказа является удобным, поскольку в классической логике нелегко доказать истинность отрицания, а язык PROLOG основан на классической логике. Отрицание рассматривается как невозможность найти доказательство, и поэтому процесс выполнения программы может оказаться очень длинным и сложным, если количество потенциальных согласований достаточно велико. Благодаря использованию **оператора отсечения** и специального предиката `failure` процесс доказательства истинности отрицания становится более эффективным, поскольку процедура поиска доказательства сокращается.

Запятые, разделяющие подцели, представляют логическую операцию `AND`, а символ `:-` интерпретируется как знак операции `IF`. Если в операторе задана только голова выражения, а правая часть отсутствует (как в приведенном ниже примере), то считается, что голова выражения имеет истинное значение.

$$p.$$

Именно поэтому предикаты, подобные следующим, рассматриваются как факты, следовательно, должны быть истинными:

color(red).
has_a(john,father).

Еще один способ трактовки понятия факта состоит в том, что факт рассматривается как безусловное заключение, которое не зависит от чего-либо иного, поэтому для его определения операция `IF` (или :-) не требуется. В отличие от этого, правила PROLOG требуют применения операции `IF`, поскольку представляют собой условные выражения, истинность которых зависит от одного или нескольких условий. В качестве примера можно привести следующие правила определения предиката `parent`:

$$\text{parent}(X, Y) \text{:- father}(X, Y).$$
$$\text{parent}(X, Y) \text{:- mother}(X, Y).$$

Эти правила означают, что X является родителем Y, если X — отец Y или если X — мать Y. Аналогичным образом, предикат `grandparent` (дедушка или бабушка) может быть определен таким образом:

$$\text{grandparent}(X, Y) \text{:- parent}(X, Z), \text{parent}(Z, Y).$$

С другой стороны, предикат `ancestor` (предок) можно определить, как показано ниже.

(1) $\text{ancestor}(X, Y) \text{:- parent}(X, Y).$
(2) $\text{ancestor}(X, Y) \text{:- ancestor}(X, Z), \text{ancestor}(Z, Y).$

Как принято в настоящей книге, обозначения (1) и (2) используются для идентификации отдельных компонентов примера.

Обеспечение поиска в языке PROLOG

Как правило, в системе, предназначенной для выполнения операторов PROLOG, применяется интерпретатор, хотя в некоторых системах может вырабатываться откомпилированный код. Общая структура системы PROLOG показана на рис. 2.6. Пользователь взаимодействует с системой PROLOG, вводя запросы в форме предикатов и получая ответы. **Предикативная база данных** содержит предикаты, представленные в виде правил и фактов, которые были в нее введены и в результате сформировали базу знаний. Интерпретатор предпринимает попытки определить, следует ли предикат запроса, введенный пользователем, из базы данных. Если запрос сформулирован как факт и этот факт находится в базе данных, интерпретатор возвращает ответ `yes`, а если этот факт в базе данных отсутствует — возвращает ответ `no`. С другой стороны, если запрос сформулирован в виде правила, то интерпретатор предпринимает попытки выполнить подцели этого правила, проводя **поиск в глубину**, как показано на рис. 2.7. Для сравнения на этом рисунке показан также ход **поиска в ширину**, хотя такой режим работы в системе PROLOG обычно не применяется.

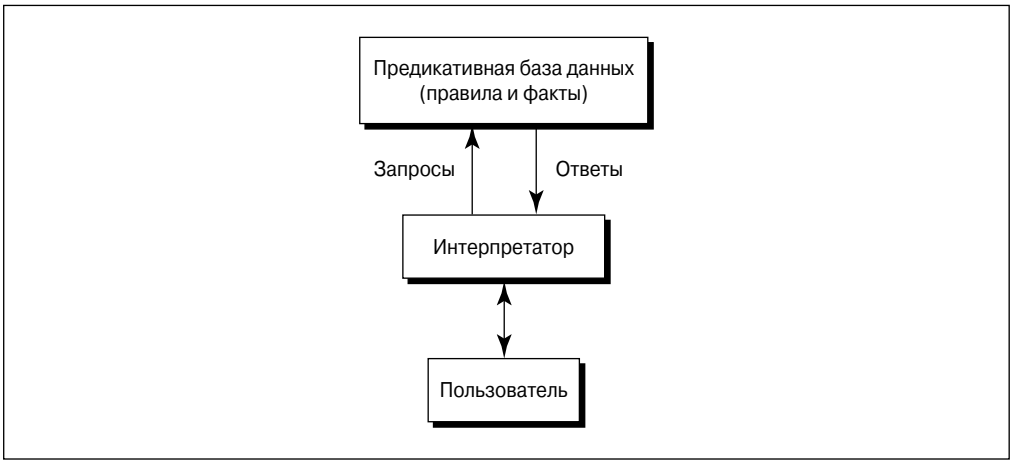


Рис. 2.6. Общая организация системы PROLOG

При поиске в глубину поиск в дереве начинается от корня, продолжается настолько далеко, насколько это возможно, после чего происходит возврат. Еще одна отличительная особенность поиска в системе PROLOG состоит в том, что он происходит слева направо. А поиск в ширину предусматривает полную обработку узлов на текущем уровне перед переходом на следующий, более низкий уровень.

В качестве примера поиска цели в системе PROLOG рассмотрим, как происходит обработка приведенных выше правил (1) и (2), относящихся к предикату `ancestor`. Предположим, что на данном этапе введены следующие факты:

(3) parent(ann,mary).
(4) parent(ann,susan).
(5) parent(mary,bob).
(6) parent(susan,john).

А теперь допустим, что системой PROLOG передан следующий запрос для определения того, является ли Энн предком Сьюзен:

:- ancestor(ann,susan).

Признаком того, что это — **запрос**, является отсутствие в правиле головы. Это означает, что система PROLOG должна доказать данное условие. Тремя типами хорновских выражений, применяемых в системе PROLOG, являются факты, правила и запросы. Условие может быть доказано, только если оно представляет собой заключение некоторого экземпляра выражения. Безусловно, для этого должно быть доказуемым само выражение, и такое доказательство осуществляется путем доказательства условий выражения.

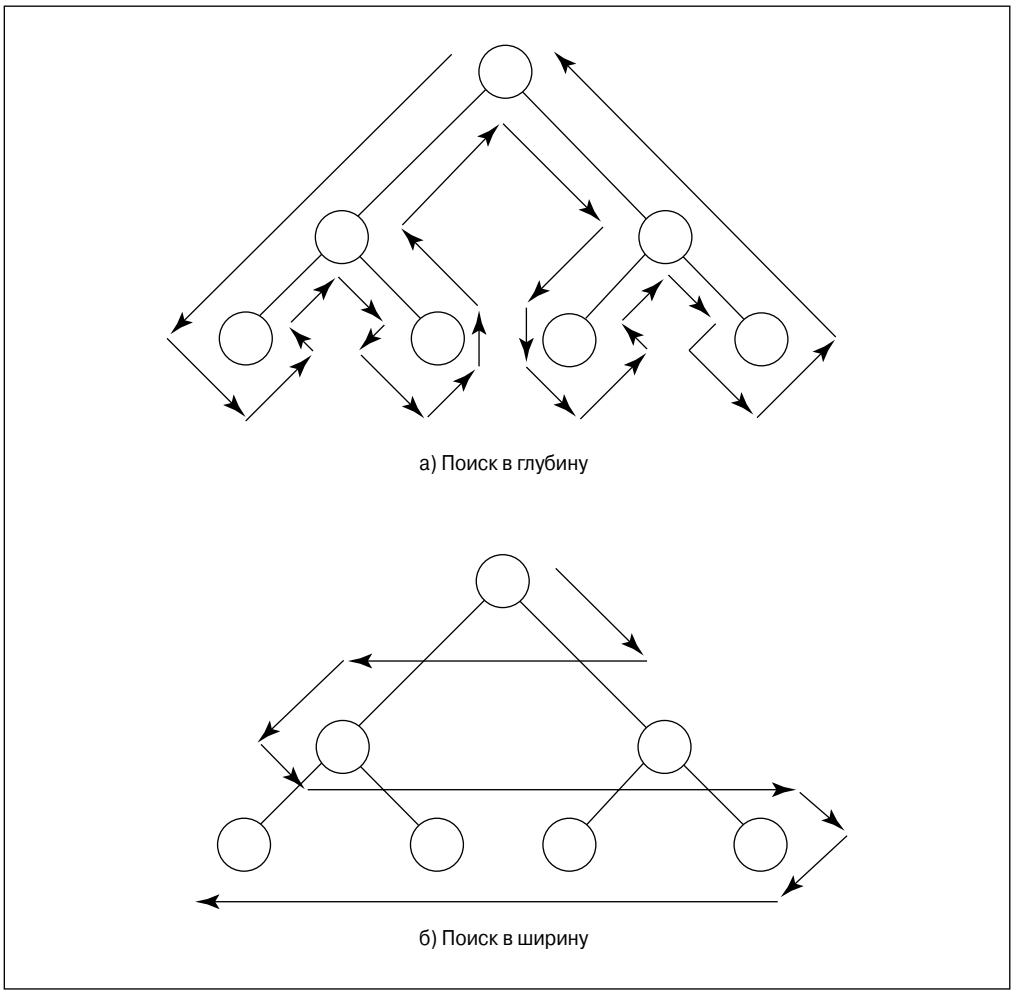


Рис. 2.7. Поиск в глубину и поиск в ширину применительно к дереву произвольной формы

После того как система PROLOG принимает этот входной запрос, начинается поиск выражения, голова которого согласуется с входным шаблоном ancestor(ann,susan). Такая процедура называется **сопоставлением с шаблоном** и полностью аналогична сопоставлению с шаблоном, в котором участвуют факты и антецеденты продукционного правила. Поиск начинается с первого введенного оператора, (1), который находится в **начале** списка операторов, и происходит в направлении к последнему оператору, (6), находящемуся в **конце** списка. При этом обнаруживается возможность согласования с первым правилом предиката `ancestor`, с правилом (1). Переменная X согласуется с `ann`, а переменная

Y — с `susan`. Поскольку теперь голова предиката согласована, система PROLOG предпринимает попытку согласовать тело правила (1), что приводит к созданию подцели `parent`(ann,susan). Затем система PROLOG осуществляет попытку согласовать эту подцель с другими выражениями и в конечном итоге согласует ее с фактом (4), parent(ann,susan). На этом все цели, подлежащие согласованию, исчерпываются, и система PROLOG отвечает "`yes`", поскольку первоначальный запрос является истинным.

В качестве еще одного примера предположим, что введен следующий запрос:

:- ancestor(ann,john).

Согласуется правило (1) предиката `ancestor`, в результате чего переменной X присваивается значение `ann`, а переменной Y — значение `john`. После этого система PROLOG предпринимает попытки согласовать тело правила (1), parent(ann,john), с каждым из операторов `parent`. Однако все попытки согласования оканчиваются неудачей, поэтому обнаруживается, что тело правила (1) не может быть истинным. Поскольку тело правила (1) не является истинным, голова этого правила также не может быть истинной.

Поскольку правило (1) не может быть истинным, система PROLOG затем предпринимает попытку проверить истинность правила (2) предиката `ancestor`. Переменной X присваивается значение `ann`, а переменной Y — значение `john`. Система PROLOG осуществляет попытки доказать, что голова правила (2) является истинной, доказывая то, что обе подцели правила (2) являются истинными. Это означает, что требуется доказать истинность обеих подцелей, ancestor(ann,Z) и ancestor(Z,john). Система PROLOG предпринимает попытки согласования подцелей в направлении слева направо, начиная с подцели ancestor(ann, Z). Проходя по списку выражений, начиная сверху, система PROLOG обнаруживает, что эта первая подцель согласуется с правилом (1), и поэтому пытается доказать истинность тела правила (1), parent(ann, Z). А после того как поиск снова начинается с верхней части списка, обнаруживается первое согласование этого тела с оператором (3), поэтому система PROLOG присваивает переменной Z значение `mary`. Теперь система PROLOG пытается согласовать вторую подцель правила (2), представленную в виде ancestor(mary,john). Эта подцель согласуется с правилом (1), и поэтому PROLOG пытается выполнить тело этого оператора, parent(mary,john). Однако в операторах от (3) до (6) факт parent(mary,john) отсутствует, поэтому данная попытка поиска оканчивается неудачей.

Затем система PROLOG пересматривает свое предположение, согласно которому переменной Z должен быть присвоено значение `mary`. Поскольку этот вариант оказался неприменимым, система пытается найти приемлемое значение. Еще одна возможность осуществляется с помощью правила (4), согласно которому переменной Z присваивается значение `susan`. Показанный здесь метод возврата в целях опробования другого пути поиска, после того как предыдущая попытка

воспользоваться некоторым путем окончилась неудачей, называется **перебором с возвратами** и часто используется для решения задач.

После выбора варианта присваивания переменной *Z* значения `susan` система PROLOG предпринимает попытку доказать, что выражение ancestor(susan,john) является истинным. Согласно правилу (1), для этого необходимо доказать истинность тела parent(susan,john). И действительно, факт (3) согласуется, поэтому указанный запрос обозначается как истинный, так как доказано, что его условия являются истинными:

:- ancestor(ann,john)

Обратите внимание на то, что управляющая структура PROLOG относится к типу марковского алгоритма, в котором последовательность поиска в ходе сопоставления с шаблонами обычно определяется тем, в каком порядке введены хорновские выражения. В этом состоит отличие системы PROLOG от экспертных систем, основанных на правилах, которые обычно следуют принципу Поста (согласно этому принципу, ход поиска не зависит от того, в каком порядке введены правила).

Система PROLOG обладает многими другими особенностями и возможностями, которые не упоминаются в настоящей книге. С точки зрения разработчиков экспертных систем, особенно важными возможностями PROLOG являются перебор с возвратами и сопоставление с шаблонами. К тому же важно то, что язык PROLOG имеет декларативный характер, поскольку это означает, что в качестве исполняемой программы непосредственно применяется спецификация программы.

2.7 Трудности, связанные с использованием семантических сетей

Безусловно, семантические сети могут оказаться весьма полезным средством представления знаний, но они имеют целый ряд ограничений, в частности, связанных с отсутствием стандартов именования связей, о чем было сказано выше. В результате возникают затруднения при попытке понять, для чего фактически была создана какая-то конкретная сеть, и действительно ли эта сеть была создана правильно. С указанной проблемой именования связей сопряжена проблема именования узлов. Например, если в сети имеется узел, обозначенный как “chair” (“кресло”), остается неизвестным, обозначает ли это слово одно из указанных ниже понятий, или что-то другое.

```
данное конкретное кресло
класс всех кресел
```

```
понятие кресла
кресло председателя собрания
```

Семантическая сеть может применяться для представления **категорических знаний** (т.е. знаний, которые определены однозначно) только при том условии, что строго определены сами имена связей и узлов. Безусловно, аналогичные проблемы обнаруживаются и при использовании языков программирования. Но, к счастью, указанная задача уже решена с введением в действие **расширяемого языка разметки** (**eXtensible Markup Language — XML**) и онтологий. Язык XML проявил себя как бесценное средство реализации стандартного способа условного представления формальной семантики в виде конструкций языков всех возможных типов. Кроме того, разработана версия XML, основанная на правилах, а также созданы языки разметки для математики, музыки, пищевой промышленности и многих других областей, в которых для обработки информации применяются компьютеры. Благодаря использованию языка XML и онтологий система World Wide Web (сокращенно WWW, или W3) постепенно превращается из хранилища данных в коллекцию информации, доступной для чтения с помощью компьютера, имеющую гораздо более широкую область применения. В связи с этим Web теперь все чаще именуют **семантической системой Web** (**Semantic Web**), а не просто системой Web, поскольку в этой системе в настоящее время реализуются новые способы представления смыслового значения информации.

Еще одна проблема, связанная с использованием семантических сетей, состоит в том, что при осуществлении поиска узлов возникает комбинаторный взрыв, особенно если ответ на запрос является отрицательным. Дело в том, что если запрос должен выработать отрицательный результат, то может оказаться, что требуется выполнить поиск по многим или даже по всем связям в сети. А как показано в описании задачи коммивояжера, приведенном в главе 1, количество связей равно факториалу от количества узлов в сети за вычетом единицы, если все узлы сети связаны друг с другом. Безусловно, такая степень связности встречается не во всех представлениях задач, но возможность комбинаторного взрыва нельзя исключить.

Семантические сети были первоначально предложены для использования в качестве модели ассоциативной памяти человека. В этой модели каждый узел имеет связи с другими узлами и выборка информации происходит в результате распространения активности по узлам сети. Но в мозгу человека должны также быть предусмотрены другие механизмы, поскольку для него не требуется много времени, чтобы ответить на вопрос: “Есть ли футбольная команда на Плутоне?” В мозгу человека имеется приблизительно 10^{11} нейронов и примерно 10^{15} связей. Если бы все его знания были представлены только с помощью семантической сети, то потребовалось бы чрезвычайно продолжительное время для формулировки отрицательного ответа на вопросы, подобные указанному выше вопросу о футбольной

команде, поскольку для поиска ответа пришлось бы ввести в действие все 10^{15} связей.

Кроме того, семантические сети не соответствуют требованиям логики, поскольку не позволяют применять для определения знаний такие способы, которые являются применимыми к логике. Как будет описано ниже в данной главе, логические способы представления позволяют определить какое-то конкретное кресло, некоторые кресла, все кресла, отсутствие кресел и т.д. Еще одна проблема состоит в том, что семантические сети остаются неприменимыми для эвристических методов, поскольку в сети невозможно представить эвристическую информацию, касающуюся того, как можно эффективно провести поиск в сети. **Эвристика** — это эмпирическое правило, которое может помочь найти решение, но, в отличие от алгоритма, не гарантирует нахождение решения. В искусственном интеллекте возможность применения эвристик очень важна, поскольку типичные задачи искусственного интеллекта настолько сложны, что алгоритмическое решение для них не существует или слишком неэффективно для практического использования. Единственной стандартной стратегией управления, встроенной в сеть, которая может помочь в решении задач, является наследование, но не все задачи могут быть представлены с использованием такой структуры.

Для устранения недостатков, присущих семантическим сетям, было опробовано много подходов. Некоторые введенные в них усовершенствования были основаны на логике, а другие усовершенствования, в которых предпринимались попытки закрепления процедур за узлами, были основаны на использовании эвристик. Такие процедуры должны были вызываться на выполнение после того, как узел становится активизированным. Но созданные в результате системы позволяли достичь лишь незначительного расширения возможностей за счет ухудшения естественной выразительности семантических сетей. По результатам анализа всех этих усилий был сделан вывод, что семантические сети, как и любые другие инструментальные средства, должны использоваться в тех областях, для которых они приспособлены в наибольшей степени, т.е. для представления бинарных отношений, поэтому не следует их загромождать, пытаясь превратить в универсальное инструментальное средство.

2.8 Схемы

Семантическая сеть может служить примером **поверхностной структуры знаний**. Поверхностность знаний в семантической сети обусловлена тем, что все знания в ней представлены в виде связей и узлов. **Структура знаний** аналогична структуре данных в том, что составляет упорядоченную коллекцию знаний, а не просто разрозненные данные. С другой стороны, **глубокую структуру знаний** имеют причинные знания, которые объясняют, почему происходит то или

иное событие. Например, может быть создана медицинская экспертная система на основе поверхностных знаний, содержащая примерно такие правила:

```
IF у пациента имеется высокая температура
THEN пациент должен принять аспирин
```

Но в подобной системе отсутствует фундаментальные знания о биохимии человеческого организма и о том, почему прием аспирина влечет за собой снижение температуры. Это правило вполне могло быть определено следующим образом:

```
IF у одного из присутствующих имеется розовая обезьяна
THEN он должен принять в подарок холодильник
```

Иными словами, знания в такой экспертной системе являются поверхностными, поскольку они основаны на синтаксисе, а не на семантике, а в этой синтаксической конструкции любые два словесных выражения могут быть заменены на `X` и `Y`, как в следующем правиле:

```
IF у какого-то человека имеется (X)
THEN он должен принять (Y)
```

Обратите внимание на то, что в этом правиле (`X`) и (`Y`) — не переменные, а подстановочные символы, заменяющие любые два словесных выражения. С другой стороны, врачи обладают причинными знаниями, поскольку учились много лет в медицинском институте и приобрели опыт лечения пациентов. Если же способ лечения, применяемый как обычно, оказывается неэффективным, врачи могут использовать свои способности к рассуждению, чтобы найти вариант, альтернативный по отношению к выбранному с помощью обычно применяемого ими метода, основанного на прецедентах. Иными словами, эксперт знает, когда можно нарушить правила.

Структура семантической сети слишком проста, поэтому с ее помощью невозможно представить знания многих типов, применяемые в реальном мире. Для лучшего представления сложных структур знаний требуются более сложные структуры представления. В искусственном интеллекте для описания более сложных структур знаний, чем семантические сети, используется термин **схема** (schema). Термин *схема* заимствован из психологии, где он обозначает непрерывную организацию знаний или реакций живого существа, вырабатываемых в ответ на стимулы. Это означает, что живые существа, изучая причинные отношения между стимулом и результатом, стремятся повторно испытать приятные стимулы и избежать неприятных.

Например, при осуществлении таких действий, как еда и питьё, складываются **сенсорно-двигательные схемы**, окрашенные ощущениями удовольствия, которые обеспечивают координацию информации, полученной от органов чувств, с необходимыми моторными (мускульными) движениями, с помощью которых осуществляется процесс еды и питья. Человек не обязан задумываться над этими

неявными знаниями, чтобы узнать, как выполнять эти действия, к тому же очень трудно точно объяснить, как они осуществляются, вплоть до уровня управления мускулами. А еще более значительные затруднения возникают при попытке объяснить схему, которой подсознательно руководствуется человек во время езды на велосипеде. Попробуйте объяснить словами чувство равновесия!

Еще одним важным типом схем являются **концептуальные схемы**, с помощью которых в мозгу человека складываются концепции. Например, допустим, что человек овладел концептуальным представлением о том, что́ такое животное. Но большинство людей, которых просят объяснить, что представляет собой животное, скорее всего, будут описывать его в таких терминах, что животное — существо с четырьмя ногами, заросшее мехом. К тому же, безусловно, сложившаяся у человека концепция животного будет зависеть от того, вырос ли он в сельской местности, в городе или на берегу реки. Тем не менее все люди имеют в своем сознании, складывающемся из концепций, определенные **стереотипы**. Безусловно, в обыденной речи термин *стереотип* может иметь отрицательную окраску, но в искусственном интеллекте он обозначает типичный пример. Поэтому для большинства людей стереотипом животного может быть собака.

Концептуальная схема — это абстракция, в которой конкретные объекты классифицируются по их общим свойствам. Например, если вы увидите рисунок с надписью “Искусственный фрукт”, на котором изображен небольшой круглый объект красного цвета с зеленым черенком, то, по-видимому, распозна́ете его как искусственное яблоко. Этот объект обладает свойствами принадлежности к яблокам, которые могут быть поставлены в соответствие с концептуальной схемой яблока.

Концептуальная схема настоящего яблока может охватывать такие общие свойства яблок, как размеры, цвет, вкус, назначение и т.д. Эта схема не будет включать подробных сведений о том, где именно было сорвано яблоко, на каком грузовике оно доставлено в супермаркет и как зовут человека, положившего его на полку. Эти подробности не имеют значения при анализе тех свойств, из которых складывается абстрактная концепция яблока. Кроме того, следует учитывать, что, например, слепой человек может иметь совсем другую концептуальную схему яблока, в которой наибольшую важность имеет текстура поверхности и запах.

Сосредоточение на общих свойствах объекта позволяет упростить проведение рассуждений об этих свойствах, поскольку при этом не приходится отвлекаться на незначащие подробности. Как правило, схемы имеют структуру, внутреннюю по отношению к применяемым в них узлам, а семантические сети не имеют такой структуры. Все знания о некотором узле, применяемом в семантической сети, заключены в надписи на этом узле. Семантическая сеть аналогична такой структуре данных, применяемой в компьютерных науках, в которой ключ поиска одновременно представляет собой данные, хранящиеся в узле. А схема аналогична структуре данных, в которой узлы содержат записи. Каждая запись может содержать данные, записи или указатели на другие узлы.

2.9 Фреймы

Одним из типов схем, который используется во многих приложениях искусственного интеллекта, является **фрейм**. Еще один тип схемы представляет собой **сценарий**, который по существу является упорядоченной во времени последовательностью фреймов. Фреймы предложены для использования в качестве метода понимания особенностей зрения, естественных языков и других областей и предоставляют удобную структуру для описания объектов, типичных для какой-то конкретной ситуации, в частности, стереотипов объектов. Фреймы являются особенно полезным средством моделирования знаний, основанных на здравом смысле, а эта область знаний с большим трудом поддается освоению с помощью компьютеров. Безусловно, семантические сети по существу можно рассматривать как двумерное представление знаний, а фреймы добавляют третье измерение, поскольку позволяют использовать узлы, имеющие внутреннюю структуру. В качестве таких структур могут применяться простые значения или другие фреймы.

Основная характерная особенность фрейма состоит в том, что он представляет взаимосвязанные знания по узкой теме, значительная часть которых — это знания, заданные по умолчанию. Система фреймов может оказаться весьма подходящей для описания механического устройства, например автомобиля. Такие компоненты автомобиля, как двигатель, корпус, тормозная система и т.д., должны рассматриваться в сочетании, чтобы можно было получить общее представление о связях между этими компонентами. Дополнительные сведения о компонентах могут быть получены с помощью изучения структуры фреймов. Безусловно, автомобили разных моделей отличаются друг от друга, но большинство автомобилей, как правило, имеют такие аналогичные компоненты, как колеса, двигатель, корпус и передаточный механизм. В этом состоит отличие фрейма от семантической сети, поскольку сеть, как правило, используется для общего представления знаний. Тем не менее, как и в случае семантических сетей, стандарты определения систем на основе фреймов отсутствуют. Для работы с фреймами создан целый ряд языков специального назначения, таких как FRL, SRL, KRL, KEE, HP-RL; кроме того, предусмотрены такие усовершенствованные средства работы с фреймами языка LISP, как LOOPS и FLAVORS.

Фрейм аналогичен структуре записи на языке высокого уровня, таком как C, или атому со списком свойств в языке LISP. В свою очередь, полям и значениям записи соответствуют такие компоненты фрейма, как **слоты** и **заполнители** слотов. Фрейм по существу представляет собой группу слотов и заполнителей, которые определяют объект, рассматриваемый в качестве стереотипа. Фрейм с описанием автомобиля показан на рис. 2.8. В терминах представления в виде тройки “объект–атрибут–значение” автомобиль — это объект, имя слота — атрибут, а заполнитель — значение.

Слоты	Заполнители
manufacturer	General Motors
model	Chevrolet Caprice
year	1979
transmission	automatic
engine	gasoline
tires	4
color	blue

Рис. 2.8. Фрейм с описанием автомобиля

Большинство фреймов не являются настолько простыми, как показано на рис. 2.8. Фреймы обнаруживают свою значимость в составе иерархических систем фреймов и в условиях применения наследования. Использование фреймов в виде заполнителей слотов и ввод в действие отношения наследования позволяет создавать очень мощные системы представления знаний. В частности, как оказалось, экспертные системы на основе фреймов являются очень полезным средством представления причинных знаний, поскольку хранимая в них информация организована с учетом причин и следствий. В отличие от этого, экспертные системы, основанные на правилах, обычно опираются на неорганизованные знания, которые не являются причинными.

Некоторые инструментальные средства, основанные на фреймах, такие как KEE, позволяют хранить в слотах самые разнообразные элементы. В частности, слоты фреймов могут хранить правила, графику, комментарии, отладочную информацию, вопросы для пользователей, гипотезы, касающиеся той или иной ситуации, или другие фреймы.

Фреймы обычно применяются для представления либо универсальных, либо специальных знаний. На рис. 2.9 показан универсальный фрейм, предназначенный для представления концепции собственности, которой может владеть человек.

Заполнителями могут быть значения, такие как название собственности в слоте `name`, или ряд значений, например, как в слоте `types`. Сами слоты могут также содержать процедуры, закрепленные за слотами и называемые **процедурными вложениями**. Сами процедуры, как правило, подразделяются на три типа. Процедура типа **if-needed** выполняется, если требуется значение заполнителя, но первоначально это значение не присутствует в слоте или является неприменимым значение **default**, предусмотренное по умолчанию. Во фреймах значения,

Слоты	Заполнители
name	property
specialization_of	a_kind_of object
types	(car, boat, house) if-added: Процедура ADD_PROPERTY
owner	default: government if-needed: Процедура FIND_OWNER
location	(home, work, mobile)
status	(missing, poor, good)
under_warranty	(yes, no)

Рис. 2.9. Универсальный фрейм, предназначенный для описания концепции собственности

предусмотренные по умолчанию, являются исключительно важными, поскольку позволяют моделировать некоторые аспекты функционирования мозга. Применяемые по умолчанию значения соответствуют ожиданиям человека в отношении некоторой ситуации, которые складываются на основании опыта. А после того как обнаруживается новая ситуация, осуществляется модификация наиболее подходящего фрейма, что позволяет проще приспособиться к ситуации. Люди не начинают с пустого места, сталкиваясь с каждой новой ситуацией. Вместо этого они модифицируют в существующих фреймах заданные по умолчанию значения или другие заполнители. Заданные по умолчанию значения часто используются для представления знаний, основанных на здравом смысле. **Знания, основанные на здравом смысле**, могут рассматриваться как общеизвестные знания. Мы используем здравый смысл, если недоступны знания, более соответствующие конкретной ситуации.

Процедура типа **if-added** вызывается на выполнение, если в слот должно быть введено какое-либо дополнительное значение. Например, в слоте `types` процедура `if-added` вызывает на выполнение в случае необходимости процедуру `ADD-PROPERTY` для добавления собственности нового типа. Например, эта процедура может быть вызвана после приобретения драгоценностей, телевизора, стереомагнитофона или собственности другого типа, поскольку указанные значения не содержатся в слоте `types`.

Процедура типа **if-removal** вызывается на выполнение каждый раз, когда возникает необходимость удалить из слота какое-либо значение. В частности, проце-

дура такого типа выполняется, если данные, представленные каким-то значением, устаревают.

Заполнители слотов могут также содержать отношения по такому же принципу, который применяется при специализации слотов. На рис. 2.9–2.11 показано, как используются отношения `a-kind-of` и `is-a` для создания иерархических связей между фреймами. Фреймы, представленные на рис. 2.9 и 2.10, являются универсальными, а фрейм, приведенный на рис. 2.11, является конкретным, поскольку представляет собой экземпляр фрейма с описанием определенного автомобиля. Разрабатывая эти фреймы, мы руководствовались соглашением, что отношения `a-kind-of` являются универсальными, а отношения `is-a` — конкретными.

Слоты	Заполнители
name	car
specialization_of	a-kind-of property
types	(sedan, sports, convertible)
manufacturer	(GM, Ford, Toyota)
location	mobile
wheels	4
transmission	(manual, automatic)
engine	(gasoline, hybrid gas/electric)

Рис. 2.10. Фрейм с описанием автомобиля — универсальный субфрейм с описанием типа собственности

Системы фреймов проектируются таким образом, чтобы более универсальные фреймы находились ближе к вершине иерархии. Предполагается, что специализация фреймов применительно к конкретным случаям может осуществляться путем модификации применяемых по умолчанию значений и создания более конкретных фреймов. С помощью фреймов может быть предпринята попытка моделировать объекты реального мира, используя универсальные знания для описания большинства атрибутов того или иного объекта и более конкретные знания — для описания частных случаев. Например, обычно принято рассматривать птиц как теплокровных, покрытых перьями живых существ, способных летать. Но некоторые птицы, такие как пингвины и страусы, не могут летать. Эти типы представляют более конкретные классы птиц, а их характеристики могут обнаруживаться на более низких уровнях иерархии фреймов по сравнению с такими птицами, как канарейки или

Слоты	Заполнители
name	John's car
specialization_of	is_a car
manufacturer	GM
owner	John Doe
transmission	automatic
engine	gasoline
status	good
under_warranty	yes

Рис. 2.11. Экземпляр фрейма с описанием определенного автомобиля

дрозды. Иными словами, верхние уровни в иерархии фреймов с описанием птиц представляют характеристики, в большей степени относящиеся ко всем птицам, а нижние уровни обозначают нечеткие границы между объектами реального мира. Объект, обладающий всеми типичными характеристиками, принято называть **прототипом**; этот термин буквально означает *первичный тип*.

Кроме того, классификация фреймов может проводиться на основании того, в какой области они применяются. **Ситуативные фреймы** содержат знания о том, чего можно ожидать в какой-то конкретной ситуации, например, на вечеринке по случаю дня рождения. С другой стороны, **фреймы действия** содержат слоты, которые определяют действия, подлежащие выполнению в какой-то конкретной ситуации. Это означает, что заполнителями слотов являются процедуры, предназначенные для выполнения определенных действий, таких как удаление дефектной детали с конвейера. Фрейм действия представляет процедурные знания. А для описания причинно-следственных отношений могут применяться **фреймы причинных знаний**, представляющие собой сочетания ситуативных фреймов и фреймов действия.

На практике были созданы очень сложные системы фреймов, предназначенные для решения самых различных задач. Одной из наиболее впечатляющих систем, продемонстрировавшей широкие возможности применения фреймов для творческого открытия математических понятий, стала программа AM (Automated Mathematician — автоматизированный математик) Дуга Лената (Doug Lenat). В классической системе AM Лената создавались, а затем исследовались сочетания интересных новых понятий. Эта система предложила некоторые совершенно новые математические доказательства для целого ряда теорем. Другие многочисленные

примеры программ для доказательства теорем и систем совершения открытий приведены в том разделе приложения Ж, который относится к данной главе.

2.10 Трудности, связанные с использованием фреймов

Первоначально фреймы предназначались для использования в качестве способа представления стереотипных знаний. Важной особенностью любого стереотипа является то, что он имеет вполне определенные характеристики, поэтому позволяет предоставить для многих своих слотов значения, заданные по умолчанию. Наглядным примером стереотипов, вполне подходящих для представления в виде фреймов, являются математические понятия. Подход на основе фреймов имеет интуитивную привлекательность, поскольку обеспечиваемое с его помощью упорядоченное представление знаний, вообще говоря, в большей степени доступно для понимания по сравнению с логическими или продукционными системами, в которых для той же цели применяется целый ряд правил [51].

Тем не менее при использовании систем фреймов обнаруживаются существенные недостатки, связанные с тем, что в этих системах допускается неограниченная модификация или уничтожение слотов. Классическим примером этой проблемы является фрейм `elephant` с описанием слонов, который приведен на рис. 2.12.

Слоты	Заполнители
name	elephant
specialization_of	a-kind-of mammal
color	grey
legs	4
trunk	a cylinder

Рис. 2.12. Фрейм `elephant`

На первый взгляд создается впечатление, что фрейм `elephant` представляет собой приемлемое общее описание слонов. Однако предположим, что существует конкретный трехногий слон по кличке Клайд. Возможно, Клайд потерял ногу из-за несчастного случая на охоте или просто притворяется, что у него нет одной ноги, чтобы попасть на страницы настоящей книги. Но важно то, что во фрейме `elephant` содержится утверждение, будто любой слон имеет четыре ноги, а не три. Поэтому мы не можем рассматривать фрейм `elephant` как действительное определение любого слона.

Безусловно, этот фрейм можно модифицировать таким образом, чтобы он допускал в качестве исключения наличие трехногих, двуногих, одноногих или даже безногих слонов. Но в результате этого определение становится не очень качественным. Дополнительные проблемы могут возникать и при использовании других слотов. Предположим, что Клайд переболел желтухой в тяжелой форме и его кожа стала желтой. Но разве из-за этого он перестал быть слоном?

Альтернативным по отношению к использованию фрейма в качестве определения является вариант, в котором фрейм рассматривается как описание, например, типичного слона. Но такой подход приводит к возникновению других проблем, связанных с наследованием. Обратите внимание на то, что во фрейме `elephant` указано, что слон относится к категории (`a-kind-of`) млекопитающих. А поскольку теперь фреймы интерпретируются как описания типичных представителей, то наша система фреймов указывает, что типичный слон представляет собой типичное млекопитающее. Тем не менее, хотя слон — млекопитающее, его, по-видимому, нельзя считать типичным млекопитающим. Судя по количеству особей, вероятно, на звание типичного млекопитающего с большим правом могут претендовать люди, коровы, крысы и овцы.

В большинстве систем фреймов не предусмотрены способы определения неизменных слотов. А поскольку в связи с этим может подвергнуться изменениям любой слот, то свойства, наследуемые одним фреймом от другого, могут быть изменены или исключены на любом уровне иерархии. Это означает, что любой фрейм по существу является примитивным, исходным фреймом, так как нет никаких гарантий того, что заданные в нем свойства будут общими. Каждый фрейм складывается из собственных правил и поэтому каждый фрейм является изначальным. В такой неограниченной системе ни в чем действительно нельзя быть уверенным и поэтому невозможно формулировать общезначимые утверждения, подобные тем, которые были сделаны при определении понятия *слон*. Кроме того, нет возможности достаточно надежно создавать сложные объекты на основании более простых определений, например, описывать слона с тремя ногами исходя из определения обычного слона. Проблемы подобного типа возникают и при использовании семантических сетей с наследованием. Если разрешено вносить изменения в свойства любого узла, то ни в чем нельзя быть уверенным.

Однако есть и другой способ трактовки фреймов, который является очень полезным. Если понятие фрейма будет расширено таким образом, чтобы оно охватывало свойства объектов, то появляется возможность рассматривать как фрейм любой объект. В язык CLIPS встроен полный объектно-ориентированный язык, называемый COOL, поэтому CLIPS может рассматриваться как расширение языка, основанного на фреймах. С тех пор как в язык CLIPS был включен язык COOL, стали доступными все преимущества объектов. Поэтому на языке CLIPS могут создаваться объектно-ориентированные экспертные системы, позволяющие использовать правила (рассматриваемые как небольшие фрагменты знаний) и вместе

с тем предоставляющие возможность организовывать более крупные фрагменты знаний в виде объектов в целях упрощения разработки и сопровождения. Правила могут применяться для работы с фактами или объектами, и поэтому CLIPS обладает всеми преимущества двух категорий инструментальных средств экспертных систем — основанных на правилах и основанных на объектах. Благодаря применению объектов в языке CLIPS становится проще создавать, эксплуатировать и сопровождать крупные базы знаний по сравнению с такими системами, в которых предпринимаются попытки представить все знания в тысячах отдельных правил и фактов.

2.11 Логика и теория множеств

Кроме правил, фреймов и семантических сетей, для представления знаний могут использоваться символы **логики** (логика изучает правила формирования неопровержимых рассуждений). Важной частью процесса формирования рассуждений является логический вывод заключений из посылок. Развитие способов применения компьютеров для осуществления рассуждений привело к созданию **логического программирования** и к разработке таких языков, основанных на логике, как PROLOG. Кроме того, логика играет чрезвычайно важную роль в экспертных системах, поскольку в таких системах применяются машины логического вывода, проводящие рассуждения от фактов к заключениям. В действительности для обозначения систем логического программирования и экспертных систем используется общий описательный термин — **автоматизированные системы формирования рассуждений**.

Самая ранняя система формальной логики была разработана древнегреческим философом Аристотелем в IV веке до н.э. Аристотелевская логика основана на понятии **силлогизма**. Аристотель изобрел четырнадцать типов силлогизмов, а еще пять типов были предложены во времена Средневековья. Силлогизмы имеют две **посылки** и одно **заключение**, которое вытекает из посылок. Классический пример силлогизма приведен ниже.

```
Посылка. Все люди смертны
Посылка. Сократ - человек
Заключение. Сократ смертен
```

В силлогизме **посылки** служат в качестве свидетельств, из которых должно обязательно следовать заключение. Силлогизм — это один из способов представления знаний. Еще одним таким способом являются **диаграммы Венна** (рис. 2.13).

Внешний круг представляет все смертные создания, а внутренний круг, обозначающий людей, изображается полностью внутри круга, представляющего смертных, для указания на то, что все люди смертны. Поскольку Сократ — человек, то представляющий его круг полностью находится внутри круга, представ-

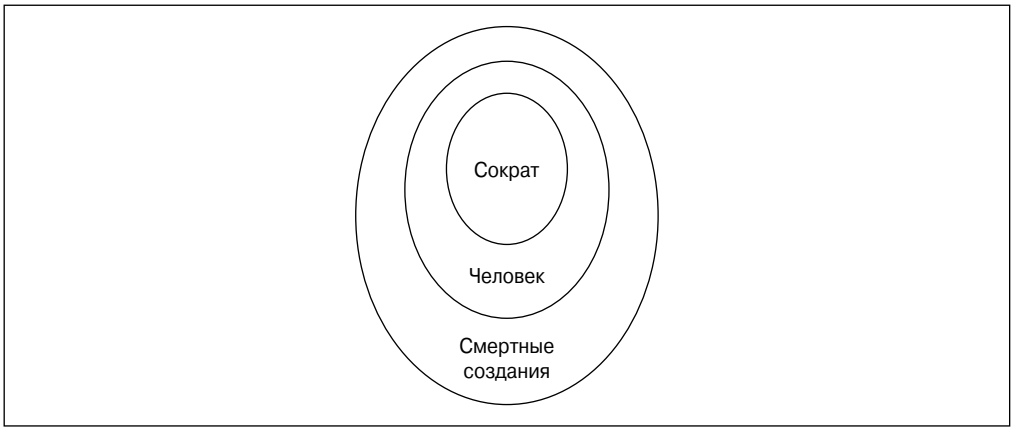


Рис. 2.13. Диаграмма Венна

ляющего людей. Строго говоря, круг, представляющий Сократа, должен выглядеть как точка, поскольку подразумевается, что круг обозначает класс. Но для удобства мы будем использовать круги при обозначении любых объектов. Заключение, что Сократ смертен, следует из того факта, что его круг находится внутри круга смертных созданий, поэтому он также должен быть смертным.

С точки зрения математики любой круг на диаграмме Венна представляет **множество**, т.е. коллекцию объектов. Объекты, принадлежащие множеству, называются **элементами** множества. Некоторые примеры множеств приведены ниже.

$$A = \{1, 3, 5\}$$
$$B = \{1, 2, 3, 4, 5\}$$
$$C = \{0, 2, 4, \dots\}$$
$$D = \{\dots, -4, -2, 0, 2, 4, \dots\}$$
$$E = \{\text{airplanes, balloons, blimps, jets}\}$$
$$F = \{\text{airplanes, balloons}\}$$

В этих примерах три точки, ..., называемые **троеточием**, указывают, что перечень термов продолжается до бесконечности.

Символ в виде греческой буквы эпсилон ($\in$) указывает, что некоторый элемент принадлежит к множеству. Например, выражение $1 \in A$ означает, что число 1 — элемент ранее определенного множества A. Если элемент не принадлежит к множеству, то для обозначения этого используется символ $\notin$, как в примере $2 \notin A$.

Если два произвольных множества, допустим, X и Y, определены так, что каждый элемент X является элементом Y, то X является **подмножеством** Y; это утверждение записывается в математической форме, как $X \subset Y$, или $Y \supset X$.

Из указанного определения подмножества следует, что каждое множество является подмножеством самого себя. А подмножество, не совпадающее с самим множеством, называется **строгим подмножеством**. Например, множество X, которое определено выше, является строгим подмножеством множества Y. В ходе рассуждений о множествах удобно считать рассматриваемые множества подмножествами **универсального множества**. При смене темы обсуждения изменяется и само универсальное множество (универсум). На рис. 2.14 показано подмножество, сформированное в результате **пересечения** двух множеств в универсуме всех автомобилей. А при обсуждении целых чисел универсумом являются все целые числа. Универсум отображается как прямоугольник, окружающий свои подмножества. Универсальные множества используются не только для удобства. Неразборчивое использование условий для определения множеств может приводить к логическим парадоксам, а универсальные множества позволяют избежать таких парадоксов.

Допустим, что A — множество всех синих автомобилей, B — множество всех автомобилей с ручной передачей, а C — множество всех синих автомобилей с ручной передачей. В таком случае можно записать следующее:

$$C = A \cap B$$



Рис. 2.14. Пересечение множеств

Здесь символ $\cap$ представляет операцию пересечения множеств. Еще один способ записи этого утверждения может быть сформулирован в терминах элементов x следующим образом:

$$C = \{x \in U \mid (x \in A) \wedge (x \in B)\}$$

В этом выражении U обозначает универсальное множество.

Символ | читается “такой, что”. Вместо символа | иногда используется двоеточие (:). Символ $\wedge$ обозначает логическую операцию AND.

Приведенное выше выражение, определяющее множество C, читается так: C состоит из элементов x универсума U, таких, что x является элементом A и x является элементом B. Логическая связка AND впервые была определена в булевой алгебре. Выражение, состоящее из двух операндов, связанных знаком логической операции AND, является истинным тогда и только тогда, когда оба операнда являются истинными. Если множества A и B не имеют общих элементов, то $A \cap B = \varnothing$, где греческая буква фи ($\varnothing$) представляет **пустое множество**, или **множество меры нуль**, $\{\}$, которое не содержит элементов. Иногда для обозначения пустого множества используется греческая буква лямбда (Λ). В качестве других обозначений для универсального множества применяется цифра 1, а пустого множества — цифра 0. Хотя пустое множество не имеет элементов, оно все еще остается множеством. В качестве примера можно привести ресторан с множеством клиентов. Если в ресторан никто не пришел, то множество клиентов пусто, но ресторан из-за этого не перестает существовать.

К другим операциям с множествами относится **объединение**, которое представляет собой множество всех элементов, принадлежащих либо множеству A, либо множеству B, либо обоим множествам одновременно:

$$A \cup B = \{x \in U \mid (x \in A) \vee (x \in B)\}$$

В этом выражении символ $\cup$ представляет операцию объединения множеств.

Символ $\vee$ обозначает логическую операцию OR.

Дополнением множества A является множество всех элементов, не принадлежащих к A:

$$A' = \{x \in U \mid \sim(x \in A)\}$$

В этом выражении штрих ($'$) обозначает дополнение множества.

Символ тильды ($\sim$) представляет логический оператор NOT.

Диаграммы Венна, представляющие основные операции над множествами, показаны на рис. 2.15.

2.12 Пропозициональная логика

Старейшим и одним из простейших типов **формальной логики** является силлогизм. Термин *формальный* означает, что логика, к которой он относится, распространяется только на форму логических утверждений, но не учитывает их значения. Иными словами, в формальной логике рассматривается только синтаксис утверждений, а не их семантика. Такое свойство логики становится чрезвычайно важным при создании экспертных системах, поскольку позволяет отделить

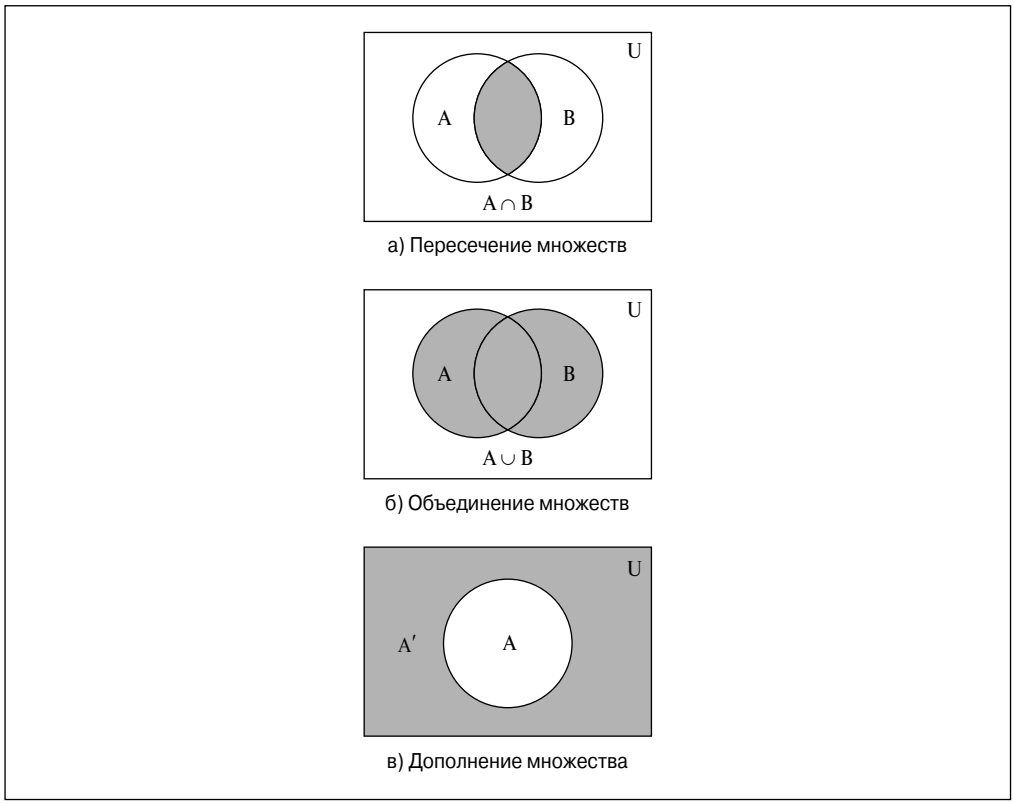


Рис. 2.15. Диаграммы Венна, представляющие основные операции с множествами

знания от рассуждений. Дело в том, что иногда высказывания, которые внешне выглядят как рассуждения, могут представлять собой знания. Например, утверждение "Президент всегда прав, поскольку он никогда не бывает неправ" внешне напоминает рассуждение в связи с наличием в нем слова "поскольку", которое связывает две части предложения. Но фактически утверждение подобного типа представляет собой **тавтологию**, поскольку, в отличие от фактов, которые могут быть в реальном мире истинными или не истинными, тавтология всегда в чисто логическом смысле истинна, так как ссылается для доказательства на саму себя. В действительности в этой тавтологии утверждается "X есть X". Но если вы не принадлежите к той же политической партии, что и Президент, то можете не согласиться с этим утверждением исходя из его семантики, а не формы, и предпочесть другую тавтологию: "Президент всегда неправ, поскольку он никогда не бывает прав".

Безусловно, для многих термин *формальная логика* звучит устрашающе, но логика не сложнее, чем алгебра. В действительности алгебра фактически пред-

ставляет собой формальную логику чисел. Например, предположим, что вас попросили решить такую задачу: в школе имеются 25 компьютеров с общим количеством микросхем памяти, равным 60. В некоторых компьютерах имеется две микросхемы памяти, в других — четыре. Каково количество компьютеров того и другого типа?

Решение этой задачи может быть записано алгебраически следующим образом:

$$25 = X + Y$$
$$60 = 2X + 4Y$$

После чего эту систему уравнения можно легко решить и получить $X = 20$ и $Y = 5$.

А теперь рассмотрим следующую задачу. На скотном дворе имеется 25 животных с общим количеством ног, равным 60. Некоторые из этих животных имеют две ноги, а другие — четыре. (*Примечание.* Наш трехногий слон Клайд находится в Африке, а не на этом скотном дворе.) Сколько имеется животных каждого типа? Вполне очевидно, что могут применяться одни и те же алгебраические уравнения, идет ли речь о компьютерах, животных или о чем-либо другом. По такому же принципу, с помощью которого алгебраические уравнения позволяют нам сосредоточиться на математических манипуляциях с символами без учета того, что они собой представляют, формальная логика позволяет сосредоточиться на рассуждениях, не погружаясь в путаницу, связанную с определением нюансов того, о каких объектах мы рассуждаем.

В качестве примера применения формальной логики рассмотрим силлогизм с бессмысленными словами сквиг и муф.

```
Посылка. Все сквиги являются муфами
Посылка. Джон - сквиг
Заключение. Джон - муф
```

Безусловно, слова сквиг и муф не имеют смысла и значения, но форма приведенного выше доказательства из-за этого не перестает быть правильной. Это означает, что рассматриваемое доказательство остается **действительным**, независимо от того, какие слова в нем используются, поскольку применяемый в нем силлогизм имеет действительную форму. Фактически действительным является любой силлогизм в следующей форме, независимо от того, какие слова будут подставлены вместо X, Y и Z:

```
Посылка. Все X суть Y
Посылка. Z есть X
Заключение. Z есть Y
```

Этот пример показывает, что в формальной логике смысл не учитывается и имеет значение только форма или внешнее представление. Логика становится

таким мощным инструментом именно благодаря используемой в ней концепции отделения формы от смысла, или семантики. В результате отделения формы от семантики появляется возможность объективно оценивать, является ли доказательство действительным, не испытывая воздействия предубеждений, вызванных семантикой. В качестве аналогии для формальной логики может рассматриваться алгебра, в которой правильность таких выражений, как $X + X = 2X$, остается неоспоримой, независимо от того, обозначает ли X целое число, количество яблок или аэропланов.

Аристотелевы силлогизмы оставались фундаментом логики до 1847 года, пока английский математик Джордж Буль (George Boole) опубликовал первую книгу с описанием **символической логики**. Безусловно, Лейбниц (Leibnitz) разработал свою собственную версию символической логики в XVII веке, но эта версия так и не получила широкого распространения. Одним из новых понятий, предложенных Булем, явилась модификация аристотелева представления, называемого **экзистенциальным значением**, согласно которому субъект рассуждений должен существовать. В соответствии с классическими аристотелевыми взглядами такое высказывание, как “все русалки хорошо плавают”, не может использоваться как посылка или следствие, поскольку русалки не существуют. А булевы представления, получившие теперь название *современных представлений*, ослабляют указанное ограничение. Важность современных представлений состоит в том, что они позволяют рассуждать о пустых классах объектов. Например, такое высказывание, как “все жесткие диски, которые отказывают, являются дешевыми”, не может использоваться в аристотелевом силлогизме, если мы не можем вести речь хотя бы об одном фактически отказавшем диске.

Еще один вклад, сделанный Булем, состоял в том, что этот ученый дал определение понятия множества **аксиом**. Формулировки аксиом состоят из символов, применяемых для представления объектов и классов, а также алгебраических операций, применяемых для манипулирования этими символами. Аксиомы представляют собой фундаментальные определения, на основании которых создаются сами логические системы, такие как математика и логика. А используя лишь одни аксиомы, можно создавать теоремы. *Теорема* — это утверждение, которое может быть доказано путем демонстрации того, что оно следует из аксиом. В период между 1910 и 1913 годами Уайтхед (Whitehead) и Расселл (Russell) опубликовали свою монументальную трехтомную работу *Principia Mathematica*, состоящую из 2000 страниц, в которой было показано, что формальная логика может служить основанием математики. Публикация указанной работы была признана важной вехой в развитии математики, поскольку Уайтхед и Расселл показали в ней, как перенести всю математику на твердое основание (а не прибегать к интуиции), полностью отказавшись от смыслового толкования арифметики и вместо этого сосредоточившись на внешних формах и манипуляциях с ними. По этой причине

математику иногда в шутку называют “способом нанесения на бумагу бессмысленных знаков”.

В 1931 году знаменитый математик Гёдель (Gödel) доказал, что не всегда возможно обосновать внутреннюю согласованность и отсутствие противоречий в формальных системах, основанных на аксиомах. Доказательство Гёделя вызвало значительное замешательство среди математиков, которые надеялись на то, что когда-либо удастся раз и навсегда доказать истинность арифметики. Но это не означает, что (как могло бы показаться неосведомленному человеку) мы фактически не можем быть уверены в правильности арифметических расчетов (следует отметить, что любой профессиональный математик, читая эти разглагольствования, найдет их весьма забавными).

Пропозициональная логика, иногда называемая **пропозициональным исчислением**, — это символическая логика, применяемая для манипулирования высказываниями. В частности, пропозициональная логика может использоваться для манипулирования **логическими переменными**, обозначающими высказывания. Безусловно, большинство людей считают, что исчисление есть нечто подобное математическим системам, которые изобретены Ньютоном и Лейбницем, но этот термин имеет более общий смысл. Аналог термина *исчисление* в английском языке (calculus) произошел от латинского слова, которое обозначало маленький камешек, используемый при выполнении расчетов. В своем наиболее общем смысле термин *исчисление* обозначает специальную систему манипулирования символами. С другой стороны, для обозначения пропозициональной логики применяются также такие термины, как **исчисление утверждений** и **исчисление высказываний**. Вообще говоря, в пропозициональной логике рассматривается определенный тип предложений естественного языка, а сами эти предложения, как показано в табл. 2.2, подразделяются на четыре основных типа.

Таблица 2.2. Типы предложений

Тип	Пример
Повелительные	Сделайте то, что я вам говорю!
Вопросительные	Что это?
Восклицательные	Это великолепно!
Повествовательные	Квадрат имеет четыре равные стороны

В пропозициональной логике рассматриваются подмножества предложений, представляющих собой повествовательные предложения, которые могут подразделяться на истинные или ложные. Очевидно, что предложение “Квадрат имеет четыре равные стороны” обладает истинностным значением *истина*, а предложение “Джордж Вашингтон был вторым по счету президентом США” обладает истинностным значением *ложь*. Предложение, истинностное значение которого

может быть определено, называется **утверждением**, или **высказыванием**. Высказывание принято называть также **закрытым предложением**, поскольку его истинностное значение не подлежит обсуждению.

Если бы авторы снабдили настоящую книгу предисловием, в котором было бы сказано “все, что содержится на этих страницах — ложно”, такое предложение нельзя было бы рассматривать ни как истинное, ни как ложное. Если бы оно было истинным, то сказанное в предисловии было бы истинным, а это невозможно. Если же такое предложение не является истинным, то все, что содержится в книге, должно быть истинным; и это означает, что высказанное предложение ложно, а это также не может быть истинным. Предложения подобного типа известны как формулировки **парадокса лжеца**. Предложения, на которые невозможно дать однозначный ответ, называются **открытыми предложениями**. Например, открытым является предложение “Шпинат имеет превосходный вкус”, поскольку оно является истинным для одних людей и ложным для других. Предложение “Он имеет высокий рост” также является открытым, поскольку в нем содержится местоимение “он”, а не указан конкретный человек. Истинностное значение не может быть присвоено открытому предложению до тех пор, пока не станет известно конкретное лицо (или персонаж), на которое указывает это местоимение. Еще одно затруднение, возникающее при анализе рассматриваемого предложения, состоит в определении смысла выражения “высокий рост”. Человек, который для одних кажется высоким, другим может не показаться таковым. С подобной неоднозначностью понятия “высокий рост” невозможно справиться с помощью пропозициональной логики или логики предикатов, но такие проблемы легко решаются с использованием нечеткой логики, которая рассматривается в главе 5.

Путем применения логических связок к отдельным высказываниям могут быть сформированы **составные высказывания**. Наиболее широко применяемые логические связки показаны в табл. 2.3.

Таблица 2.3. Широко применяемые логические связки

Связка	Значение
$\wedge$	AND; конъюнкция
$\vee$	OR; дизъюнкция
$\sim$	NOT; отрицание
$\rightarrow$	если. . . то. . . ; условная операция
$\leftrightarrow$	если и только если; двусторонняя условная операция

Строго говоря, знак операции отрицания — это не связка, поскольку отрицание представляет собой унарную операцию, применяемую к одному операнду, следующему за знаком этой операции, т.е. фактически знак операции отрицания ничего не связывает. Операция отрицания имеет более высокий приоритет по сравнению

с другими операциями, поэтому нет необходимости задавать круглые скобки вокруг выражения, в котором она применяется. Таким образом, выражение $\sim p \wedge q$ означает то же, что и выражение $(\sim p) \wedge q$.

Условная операция аналогична стрелке продукционных правил в том, что также может быть представлена в форма `IF-THEN`. Например, следующее выражение:

```
IF идет дождь THEN захвати с собой зонтик
```

может быть представлено в такой форме:

$$p \rightarrow q$$

в которой применяются следующие обозначения:

p — идет дождь

q — захвати с собой зонтик

Иногда вместо символа $\rightarrow$ используется символ $\supset$. Для обозначения условной операции применяется также другой термин — **материальная импликация**.

Двусторонняя условная операция, $p \leftrightarrow q$, эквивалентна приведенному ниже выражению и является истинной только тогда, когда p и q имеют одинаковые истинностные значения.

$$(p \rightarrow q) \wedge (q \rightarrow p)$$

Таким образом, выражение $p \leftrightarrow q$ является истинным, только если p и q одновременно имеют истинное значение или ложное значение. Двусторонняя условная операция имеет такие толкования:

p если и только если q

q если и только если p

если p то q, и если q то p

Как уже было сказано выше, тавтология — это составное высказывание, которое всегда является истинным, независимо от того, истинны или ложны отдельные высказывания, входящие в его состав. С другой стороны, **противоречие** — это составное высказывание, которое всегда является ложным. А **контингентными** называются высказывания, которые не представляют собой ни тавтологию, ни противоречие. Тавтологии и противоречия называются соответственно аналитически истинными и аналитически ложными высказываниями, поскольку их истинностное значение может быть определено исключительно на основании анализа формы. Например, истинностная таблица для выражения $p \vee \sim p$ показывает, что это — тавтология, а для выражения $p \wedge \sim p$, что это — противоречие.

Если условная операция одновременно представляет собой тавтологию, то эта операция именуется **импликацией** и содержит символ $\Rightarrow$ вместо символа $\rightarrow$.

Двусторонняя условная операция, которая одновременно представляет собой тавтологию, называется **логической эквивалентностью**, или **материальной эквивалентностью**, и символически обозначается с помощью символа $\Leftrightarrow$ или $\equiv$. Два логически эквивалентных высказывания всегда имеют одно и то же истинностное значения. Например, $p \equiv \sim \sim p$.

К сожалению, не существует единственного возможного определения для импликации, поскольку количество истинностных таблиц для двух переменных, которые могут принимать истинные и ложные значения, равно 16. Поэтому в ранних экспертных системах, которые разрабатывались в 1980-х годах, применялись одиннадцать разных определений для операции импликации.

Условная операция не имеет точно такой же смысл, как и выражение `IF-THEN` в процедурном языке или в экспертной системе, основанной на правилах. В процедурных языках и в экспертных системах конструкция `IF-THEN` указывает, что если истинны условия в определении `IF`, то должны быть выполнены действия, описания которых следует за ключевым словом `THEN`. А в логике значение условной операции определяется по ее истинностной таблице. Смысл такой операции может быть переведен на естественный язык многими способами. Например, если дано следующее выражение:

$$p \rightarrow q$$

в котором p и q представляют собой любые высказывания, то данное выражение может быть переведено на естественный язык таким образом:

p влечет за собой q

если p то q

q только если p

p является достаточным для q

q если p

q является необходимым для p

Например, допустим, что через p обозначено высказывание “гражданин в возрасте 18 лет или старше”, а q обозначает высказывание “имеет право голосовать”. Условное выражение $p \rightarrow q$ может обозначать любое из следующих предложений:

```
гражданин находится в возрасте 18 лет или старше, и это
влечет за собой, что данный гражданин имеет право
голосовать если гражданин находится в возрасте 18 лет или
старше, то данный гражданин имеет право голосовать
гражданин имеет право голосовать, только если этот
гражданин находится в возрасте 18 лет или старше
гражданин находится в возрасте 18 лет или старше, и этого
достаточно, чтобы данный гражданин имел право голосовать
```

```
гражданин имеет право голосовать, если этот гражданин
находится в возрасте 18 лет или старше
гражданин должен иметь возраст 18 лет или старше,
и это условие является необходимым для того, чтобы он имел
право голосовать
```

В некоторых случаях потребовалось изменить формулировки, чтобы полученные предложения стали грамматически правильными предложениями естественного языка. В последнем предложении говорится о том, что если не выполняется требование, обозначенное как q, то не выполняется и p. Это предложение можно представить на правильном естественном языке так: “Чтобы получить возможность голосовать, необходимо иметь возраст 18 лет или старше”.

Значения бинарных логических связок показаны в табл. 2.4. Эти связки называются бинарными, так как требуют двух операндов. Связка отрицания ($\sim$) представляет собой знак унарной операции, применяемой к одному операнду, который следует за этим знаком (табл. 2.5).

Таблица 2.4. Истинностная таблица для бинарных логических связок

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

Таблица 2.5. Истинностная таблица для связки отрицания

p	$\sim p$
T	F
F	T

Множество логических связок называется **адекватным**, если с помощью связок, взятых только из этого множества, можно представить любую истинностную функцию. К примерам адекватных множеств относятся $\{\sim, \wedge, \vee\}$, $\{\sim, \wedge\}$, $\{\sim, \vee\}$ и $\{\sim, \rightarrow\}$.

Множество, содержащее единственный элемент, называется **одноэлементным множеством**. Одноэлементными являются два из адекватных множеств. Эти множества включают связки NOT-OR (**NOR**) и NOT-AND (**NAND**). Множество со связкой NOR обозначается как $\{\downarrow\}$, а множество со связкой NAND — как $\{|\}$. Знак операции (|) называется **штрихом**, или **дизъюнкцией отрицаний**. Этот знак используется для отрицания того, что p и q одновременно являются истинными.

Таким образом, в выражении $p \mid q$ утверждается, что по меньшей мере одно из высказываний p или q является истинным. А в **операции конъюнкции отрицаний** ($\downarrow$) отрицается то, что является истинным либо p, либо q. Это означает, что в выражении $p \downarrow q$ утверждается, что p и q одновременно являются ложными.

2.13 Логика предикатов первого порядка

Безусловно, пропозициональная логика является очень полезной, но имеет целый ряд ограничений. Основная проблема состоит в том, что пропозициональная логика может применяться только с полным высказыванием. Это означает, что с ее помощью нельзя исследовать внутреннюю структуру высказывания. Пропозициональная логика не позволяет даже доказать обоснованность примерно такого силлогизма:

```
Все люди смертны
Все женщины - люди
Следовательно, все женщины смертны
```

В целях обеспечения возможности анализировать более общие случаи была разработана **логика предикатов**. В своей простейшей форме она принимает вид логики предикатов **первого порядка** и является основой таких языков логического программирования, как PROLOG. В настоящем разделе для обозначения логики предикатов первого порядка будет применяться просто термин *логика предикатов*. Пропозициональная логика — это подмножество логики предикатов.

Логика предикатов позволяет рассматривать внутреннюю структуру предложений. В частности, в ней допускается использование таких специальных слов, как “все”, “некоторые” и “ни один”, называемых **кванторами**. Эти слова очень важны, поскольку позволяют присваивать явные количественные оценки другим словам и более точно формулировать предложения. Все кванторы отвечают на вопрос “сколько” и поэтому позволяют применять более широкий круг выражений по сравнению с пропозициональной логикой.

2.14 Квантор всеобщности

Предложение с квантором всеобщности, или универсально квантифицированное предложение, принимает одно и то же истинностное значение при подстановке всех объектов из одной и той же области определения. **Квантор всеобщности** представляется с помощью символа $\forall$, за которым следует один или несколько параметров, соответствующих **переменным области определения**. Символ $\forall$ интерпретируется как “для каждого” или “для всех”. Например, в области определения чисел следующее выражение гласит о том, что для каждого x (где x —

число) выражение $x + x = 2x$ является истинным:

$$(\forall x)(x + x = 2x)$$

А если указанное выражение будет обозначено символом p, то приведенное выше утверждение может быть представлено еще более кратко следующим образом:

$$(\forall x)(p)$$

В качестве еще одного примера предположим, что p обозначает предложение "все собаки — животные", как показано ниже.

$$(\forall x)(p) \equiv (\forall x)(\text{если } x - \text{собака} \rightarrow x - \text{животное})$$

Противоположным по отношению к этому предложению является предложение "ни одна собака не является животным", которое записывается следующим образом:

$$(\forall x)(\text{если } x - \text{собака} \rightarrow \sim x - \text{животное})$$

Это высказывание можно также записать таким образом:

```
Каждая собака не является животным
Все собаки не являются животными
```

В качестве еще одного примера укажем, что предложение "все треугольники являются многоугольниками" записывается следующим образом:

$$(\forall x)(x \text{ является треугольником} \rightarrow x \text{ является многоугольником})$$

Это высказывание можно прочитать так: "Для всех x, если x — треугольник, то x — многоугольник". Более короткий способ записи логических высказываний, в которых участвуют предикаты, состоит в использовании **предикативных функций** для описания свойств рассматриваемого предмета. Поэтому приведенное выше логическое высказывание можно также записать следующим образом:

$$(\forall x)(\text{triangle}(x) \rightarrow \text{polygon}(x))$$

Предикативные функции обычно записываются с применением более краткой системы обозначений, в которой предикаты обозначаются прописными буквами. Например, предположим, что T обозначает треугольник, а P — многоугольник. В таком случае утверждение, касающееся треугольников, может быть записано более кратко, как показано ниже.

$$(\forall x)(T(x) \rightarrow P(x)) \text{ или } (\forall y)(T(y) \rightarrow P(y))$$

Следует отметить, что вместо фиктивных переменных x и y можно использовать любые другие переменные. В качестве еще одного примера предположим,

что H — предикативная функция, обозначающая людей, а M — функция, обозначающая смертных. В таком случае утверждение, согласно которому все люди смертны, можно записать таким образом:

$$(\forall x)(H(x) \rightarrow M(x))$$

Это утверждение читается так: для всех x, если x — человек, то x смертен. Как показано на рис. 2.16, это предложение логики предикатов может быть также представлено с помощью семантической сети. Кроме того, оно может быть выражено в терминах правил:

```
IF x - человек
THEN x смертен
```

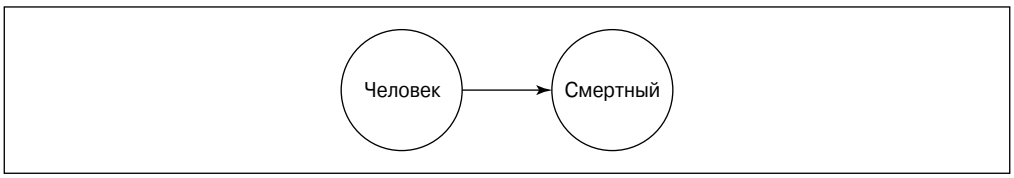


Рис. 2.16. Представление утверждения логики предикатов в виде семантической сети

Квантор всеобщности может также интерпретироваться как конъюнкция предикатов, относящихся к отдельным экземплярам. Как было указано выше, экземпляром называется конкретный случай. Например, допустим, что собака по имени Спарклер представляет собой конкретный экземпляр класса собак. Эту мысль можно выразить следующим образом:

```
Dog(Sparkler)
```

В данном примере `Dog` — предикативная функция, а `Sparkler` — экземпляр.

Такое предложение логики предикатов:

$$(\forall x)P(x)$$

может интерпретироваться в терминах экземпляров a_i, как показано ниже.

$$P(a_1) \wedge P(a_2) \wedge P(a_3) \wedge \cdots \wedge P(a_N)$$

В этом случае многоточие указывает, что действие предиката распространяется на все элементы данного класса. Таким образом, в приведенном выше выражении сказано, что предикат применяется ко всем экземплярам класса.

В выражениях может использоваться несколько кванторов. Например, как показано ниже, для формулировки закона коммутативности сложения для чисел требуются два квантора.

$$(\forall x)(\forall y)(x + y = y + x)$$

В этом выражении утверждается, что “для каждого x и для каждого y сумма x и y равна сумме y и x”.

2.15 Квантор существования

Квантором еще одного типа является **квантор существования**. Квантор существования определяет утверждение как истинное применительно по крайней мере к одному элементу области определения. Он представляет собой ограниченную форму квантора всеобщности (в котором утверждается, что некоторое выражение является истинным для всех элементов области определения). Квантор существования записывается как символ $\exists$, за которым следует одна или несколько переменных, например, как показано ниже.

$$(\exists x)(x \cdot x = 1)$$
$$(\exists x)(\text{elephant}(x) \wedge \text{name}(Clyde))$$

В первом из приведенных выше предложений указано, что имеется некоторое число x, результат умножения которого на самого себя равен 1. Во втором выражении сказано, что существует некоторый слон по кличке Клайд.

Квантор существования можно прочитать несколькими способами, в частности, таким образом:

```
существует
по меньшей мере один
для некоторых
имеется некоторый
некоторые
```

В качестве еще одного примера можно привести выражение, в котором указано, что все слоны имеют четыре ноги:

$$(\forall x)(\text{elephant}(x) \rightarrow \text{four-legged}(x))$$

А утверждение, что некоторые слоны имеют три ноги, записывается со знаком логической операции AND и квантором существования следующим образом:

$$(\exists x)(\text{elephant}(x) \wedge \text{three-legged}(x))$$

Так же как квантор всеобщности может быть выражен с помощью конъюнкции, квантор существования может быть выражен с помощью дизъюнкции экземпляров, a_i:

$$P(a_1) \vee P(a_2) \vee P(a_3) \vee \cdots \vee P(a_N)$$

В табл. 2.6 в качестве примеров приведены утверждения с кванторами, в которых P обозначает предложение “слоны — млекопитающие”, и их отрицания.

Числа в круглых скобках обозначают примеры, которые будут рассматриваться в последующем описании.

Таблица 2.6. Примеры отрицаемых кванторов

Пример	Значение
(1a) $(\forall x)(P)$	Все слоны — млекопитающие
(1b) $(\exists x)(\sim P)$	Некоторые слоны не млекопитающие
(2a) $(\exists x)(P)$	Некоторые слоны — млекопитающие
(2b) $(\forall x)(\sim P)$	Никакие слоны не млекопитающие

Примеры (1a) и (1b) являются отрицаниями по отношению друг к другу; таковыми являются также примеры (2a) и (2b). Обратите внимание на то, что отрицанием выражения с квантором всеобщности из примера (1a) становится выражение с отрицанием предложения P с квантором существования, как показано в примере (1b). Аналогичным образом отрицание выражения с квантором существования из примера (2a) представляет выражение с отрицанием предложения P и квантором всеобщности, как показано в примере (2b).

2.16 Кванторы и множества

Кванторы и множества могут использоваться для определения подмножеств универсального множества U, как показано в табл. 2.7.

Таблица 2.7. Некоторые выражения с множествами и их логические эквиваленты

Выражение с множествами	Логический эквивалент
$A = B$	$\forall x(x \in A \leftrightarrow x \in B)$
$A \subseteq B$	$\forall x(x \in A \rightarrow x \in B)$
$A \cap B$	$\forall x(x \in A \wedge x \in B)$
$A \cup B$	$\forall x(x \in A \vee x \in B)$
A'	$\forall x(x \in U \mid \sim(x \in A))$
U (универсальное множество)	T (истинное значение)
$\varnothing$ (пустое множество)	F (ложное значение)

Отношение, согласно которому A является строгим подмножеством B (имеющее вид $A \subset B$), означает, что все элементы множества A принадлежат к множеству B, но, в свою очередь, в множестве B имеется хотя бы один элемент, не принадлежащий к множеству A. Предположим, что E обозначает всех слонов, а M — всех млекопитающих. В таком случае следующее отношение между множествами представляет собой утверждение, что все слоны — млекопитающие, но не все

млекопитающие являются слонами:

$$E \subset M$$

Обозначив множество животных серого цвета как G, а множество животных с четырьмя ногами — как F, можно записать утверждение, что все серые и четырехногие слоны являются млекопитающими, следующим образом:

$$(E \cap G \cap F) \subset M$$

Ниже приведены некоторые примеры предложений с кванторами, в которых используются указанные обозначения.

E — слоны
R — рептилии
G — серые
F — четырехногие
D — собаки
M — млекопитающие

Никакие слоны не являются рептилиями

$$E \cap R = \varnothing$$

Некоторые слоны - серые

$$E \cap G \neq \varnothing$$

Никакие слоны не являются серыми

$$E \cap G = \varnothing$$

Некоторые слоны не являются серыми

$$E \cap G' \neq \varnothing$$

Все слоны являются серыми и четырехногими

$$E \subset (G \cap F)$$

Все слоны и собаки являются млекопитающими

$$(E \cup D) \subset M$$

Некоторые слоны являются четырехногими и серыми

$$(E \cap F \cap G) \neq \varnothing$$

Между формой представления информации с помощью множеств и логической формой имеется еще одна аналогия, которая выражается в виде законов де Моргана, как показано в табл. 2.8. Символ эквивалентности (≡), т.е. знак двусторонней условной операции, означает, что выражение, находящееся слева от него, имеет такое же истинностное значение, как и выражение, находящееся справа. Это означает, что указанные выражения эквивалентны.

Таблица 2.8. Законы де Моргана, представленные в форме с использованием множеств и логической форме

Форма с использованием множеств	Логическая форма
$(A \cap B)' \equiv A' \cup B'$	$\sim(p \wedge q) \equiv \sim p \vee \sim q$
$(A \cup B)' \equiv A' \cap B'$	$\sim(p \vee q) \equiv \sim p \wedge \sim q$

2.17 Ограничения логики предикатов

Безусловно, логика предикатов применима в ситуациях многих типов, но некоторые типы утверждений невозможно представить на основе логики предикатов с использованием кванторов всеобщности и кванторов существования. Например, в логике предикатов невозможно выразить следующее утверждение:

`Большинство учащихся в классе получили оценки отлично`

В этом утверждении квантор “большинство” означает “больше половины”.

Квантор “большинство” не может быть выражен в терминах квантора всеобщности и квантора существования. Для реализации квантора “большинство” в логике должны быть предусмотрены некоторые предикаты, обеспечивающие подсчет количества элементов, что возможно при использовании нечеткой логики, описанной в главе 5. Еще одно ограничение логики предикатов состоит в том, что она не позволяет выражать зависимости, которые могут быть истинными только иногда, но не всегда. Указанная проблема также может быть решена с помощью нечеткой логики. Однако внедрение в логическую систему средств проведения вычислений влечет за собой также появление дополнительных усложнений; к тому же в результате логика начинает в большей степени напоминать математику.

2.18 Резюме

В настоящей главе приведены основные сведения о логике, представлении знаний и некоторых методах представления знаний. Выбор правильного метода представления знаний в экспертных системах имеет очень важное значение.

Рассматривались также логические ошибки, поскольку для инженера по знаниям важно понимать правила предметной области и не путать форму представления знаний с семантикой. Если не заданы формальные правила, то экспертная система может не позволить достичь правильных заключений, что повлечет за собой губительные последствия при эксплуатации таких ответственных систем, от которых зависит жизнь и собственность людей.

С точки зрения логики знания могут классифицироваться многими способами и рассматриваться как априорные, апостериорные, процедурные, декларативные и неявные. К числу методов, широко применяемых в экспертных системах для представления знаний, относятся продукционные правила, объекты, семантические сети, схемы, фреймы и логика. Каждый из указанных подходов имеет свои преимущества и недостатки. Прежде чем приступать к проектированию экспертной системы, необходимо принять решение о том, какой из способов представления знаний может стать основой выбора подхода к решению рассматриваемой задачи. Следует не пытаться применять одно и то же инструментальное средство для решения всех задач, а выбирать каждый раз наилучшее средство. Кроме того, в данной главе речь шла о том преимуществе языка CLIPS, что он позволяет представлять знания не только с помощью объектов, но и с помощью правил. Многочисленные ссылки, с помощью которых можно ознакомиться с дополнительной информацией о логике, знаниях и логических ошибках приведены в приложении Ж. В приложениях А–В содержатся полезные справочные сведения об эквивалентностях, кванторах и свойствах множеств.

Задачи

2.1. Нарисуйте семантическую сеть для классификации компьютеров с использованием связей AKO и IS-A. Рассмотрите такие классы, как микрокомпьютеры; мэйнфреймы; суперкомпьютеры; вычислительные системы; выделенные компьютеры; компьютеры общего назначения; одноплатные компьютеры; компьютеры, реализованные в виде одной микросхемы; однопроцессорные и многопроцессорные компьютеры. Включите данные о конкретных экземплярах.

2.2. Нарисуйте семантическую сеть для классификации средств обеспечения межкомпьютерного взаимодействия с использованием связей AKO и IS-A. Рассмотрите такие классы, как локальные сети, глобальные сети, сети с маркерным кольцом, звездообразные сети, централизованные сети, децентрализованные сети, распределенные сети, сети с модемными линиями связи, телекоммуникационные сети, группы новостей и электронная почта. Включите данные о конкретных экземплярах.

2.3. Нарисуйте систему фреймов для описания здания, в котором вы проходите обучение. Предусмотрите возможность представить данные об офисах, аудиториях, лабораториях и т.д. Включите данные о конкретных экземплярах с заполненными слотами для одного фрейма каждого типа, такого как офис и аудитория.

2.4. Нарисуйте систему фреймов действия, позволяющую узнать, какие действия следует предпринять в случае аппаратного отказа конкретной компьютерной системы. Рассмотрите возможность аварийного отказа диска, источника электропитания, процессора и памяти.

2.5. Нарисуйте диаграммы Венна и обозначьте отдельные элементы каждой диаграммы как термы выражений, в которых используются описанные ниже операции с множествами. В приведенных примерах $\{1, 2\}$ — первое множество, а $\{2, 3\}$ — второе.

а) Результат операции “исключительное ИЛИ” с двумя множествами, A и B, состоит из всех элементов, которые находятся или в одном, или в другом, но не обоих множествах одновременно. Операцию “исключительное ИЛИ” называют также **симметрической разностью множеств** и обозначают знаком операции $(/)$. Например:

$$\{1, 2\}/\{2, 3\} = \{1, 3\}$$

б) Результат выполнения операции **разности множеств** с двумя множествами, которая обозначается знаком операции $(-)$, состоит из всех элементов первого множества, которые не принадлежат также ко второму множеству. Например:

$$\{1, 2\} - \{2, 3\} = \{1\}$$

2.6. Составьте истинностные таблицы и определите, какие из следующих выражений представляют собой тавтологии, противоречия или **контингентные утверждения**, и какие не относятся ни к одному из этих типов. При выполнении упражнений а) и б) вначале представьте рассматриваемые утверждения с помощью логических символов и связок.

а) Если я пройду этот курс и получу оценку “отлично”, то я пройду этот курс или получу оценку “отлично”.

б) Если я пройду этот курс, то получу оценку “отлично”

и

я пройду этот курс и не получу оценку “отлично”.

в) $((A \wedge \sim B \rightarrow (C \wedge \sim C)) \rightarrow (A \rightarrow \sim B)$.

г) $(A \rightarrow B) \wedge (\sim B \vee C) \wedge (A \wedge \sim C)$.

д) $A \rightarrow \sim B$ (контингентное утверждение).

2.7. Два предложения логически эквивалентны тогда и только тогда, когда они имеют одно и то же истинностное значение. Таким образом, если A и B — любые утверждения, то следующее выражение с двусторонним условным оператором становится истинным при любых значениях утверждений, входящих в его состав, т.е. превращается в тавтологию:

$$A \leftrightarrow B \text{ или эквивалентность } A \equiv B$$

Определите, являются ли два приведенных ниже предложения логически эквивалентными, записав их с использованием логических символов, а также определите, показывает ли истинностная таблица двусторонней условной операции с этими предложениями, что полученное выражение представляет собой тавтологию.

```
Если вы едите банановый сплит, то не можете есть торт.
Если вы едите торт, то не можете есть банановый сплит.
```

2.8. Запишите логическое выражение, эквивалентное выражениям с соответствующими операциями разности множеств и симметрической разности множеств.

2.9. Покажите, что следующие эквивалентности соблюдаются для любых множеств A, B, C и пустого множества $\varnothing$.

а) $(A \cup B) \equiv (B \cup A)$.

б) $(A \cup B) \cup C \equiv A \cup (B \cup C)$.

в) $A \cup \varnothing \equiv A$.

г) $A \cap B \equiv B \cap A$.

д) $A \cap A' \equiv \varnothing$.

2.10. Запишите приведенные ниже утверждения в квантифицированной форме.

а) Все собаки — млекопитающие.

б) Никакая собака не является слоном.

в) Некоторые программы содержат ошибки.

г) Ни одна из моих программ не содержит ошибок.

д) Все ваши программы содержат ошибки.

2.11. **Степенны́м множеством** $P(S)$ множества S называется множество, все элементы которого представляют собой подмножества множества S. Множество $P(S)$ всегда содержит по меньшей мере такие элементы, как пустое множество $\varnothing$ и множество S.

а) Найдите степенное множество множества $A = \{2, 4, 6\}$.

б) Сколько элементов содержит степенное множество множества с N элементами?

2.12. Выполните следующие задания.

а) Запишите истинностную таблицу для выражений, представленных в табл. 2.9.

Таблица 2.9. Логические выражения, которым могут быть даны осмысленные определения

Осмысленное определение	Выражение
либо p либо q	$(p \vee q) \wedge \sim(p \wedge q)$
ни p ни q	$\sim(p \vee q)$
p если не q	$\sim q \rightarrow p$
p потому что q	$(p \wedge q) \wedge (q \rightarrow p)$
никакие p не есть q	$p \rightarrow \sim q$

б) Покажите, что $(p \vee q) \wedge \sim(p \wedge q) \equiv p/q$, где знак / обозначает операцию "исключительное ИЛИ".

2.13. Выполните следующие упражнения.

а) Запишите истинностные таблицы для операций NAND и NOR.

б) Докажите, что $\{\downarrow\}$ и $\{|\}$ — адекватные множества, представив связки $\sim$, $\wedge$ и $\vee$ в терминах связки $\downarrow$, а затем — связки $|$. Для этого составьте истинностные таблицы, чтобы показать, что следующие выражения представляют собой логические эквивалентности:

$$\begin{aligned} \sim \sim p &\equiv p \\ (p \wedge q) &\equiv (p \downarrow p) \downarrow (q \downarrow q) \\ \sim p &\equiv p \mid p \\ (p \vee q) &\equiv (p \mid p) \mid (q \mid q) \end{aligned}$$

в) Поскольку $p \rightarrow q \equiv \sim(p \wedge \sim q)$, представьте выражение $p \rightarrow q$ с помощью знаков операции $\downarrow$.

г) Каковы преимущества и недостатки использования адекватных одноэлементных множеств, во-первых, с точки зрения удобства использования системы обозначений, и, во-вторых, конструирования электронных схем для микросхем?

2.14. Каковы преимущества и недостатки проектирования экспертной системы со знаниями, относящимися к нескольким предметным областям?

2.15. Объясните, почему приходится по-разному кипятить воду в высокогорной местности (например, в Денвере) и на равнине (например, в Хьюстоне), если в процессе кипячения воды готовится яйцо, которое должно быть сварено вкрутую. Относится ли эта задача к проблематике логики или физики?

2.16. Даны приведенные ниже операторы PROLOG; докажите, что Том — дедушка самого себя.

```
mother(pat,ann).              ; Пэт — мать Энн
parents(jim,ann,tom).         ; Джим и Энн — родители Тома
surrogatemother(pat,tom).     ; Пэт — суррогатная мать Тома
```

Глава 3

Методы логического вывода

3.1 Введение

Настоящая глава представляет собой вводное описание методов формирования рассуждений о знаниях. При изучении этих методов необходимо отделить смысл слов, используемых в рассуждениях, от самих рассуждений, чтобы иметь возможность рассматривать данную тему на основе формального подхода, обсуждая различные методы логического вывода [83]. В частности, в данной главе будет рассматриваться важный вид рассуждений, применяемый в экспертных системах, в котором заключения логически выводятся из фактов с использованием правил. Такой вид рассуждений имеет особое значение в экспертных системах, поскольку формирование логических выводов — это фундаментальный метод решения задач в экспертных системах [60].

Как было сказано в главе 1, экспертные системы обычно используются в том случае, если отсутствует подходящий алгоритм или алгоритмическое решение. Экспертная система должна построить цепь логических выводов для выработки решения по такому же принципу, как это делают люди, особенно если сталкиваются с неполной или недостающей информацией. Простая компьютерная программа, в которой для решения задачи применяется некоторый алгоритм, просто неспособна выдать ответ, если не заданы все необходимые параметры.

С другой стороны, экспертная система вырабатывает наилучшее предположение так же, как это делает человек, если ему приходится решать некоторую задачу, а оптимальное решение не может быть определено. Идея состоит в том, что экспертная система должна предложить хотя бы какое-то решение так же, как мог бы сделать человек, в надежде на то, что это решение (к счастью) окажется удачным, а не оставлять задачу вообще нерешенной. Безусловно, полученное решение может лишь на 95% приближаться к оптимальному решению, но это все равно

лучше, чем полное отсутствие решения. В таком подходе фактически нет ничего нового. Например, в математике для решения задач, не имеющих аналитического решения, уже в течение многих столетий используются числовые расчеты, такие как метод решения дифференциальных уравнений Рунге–Кутта.

Примечание. Полезный справочный материал к этой главе содержится в приложениях А–В.

3.2 Деревья, решетки и графы

Дерево — это иерархическая структура данных, состоящая из **узлов**, хранящих информацию (или знания), и **ребер**, соединяющих узлы. Ребра иногда называют **связями**, или **дугами**, а узлы — **вершинами**. На рис. 3.1 показано бинарное дерево общего вида, в котором имеются нуль, одно или два ребра в расчете на каждый узел. В **ориентированном дереве** на самом верхнем уровне **иерархии** находится **корневой узел**, а на самых нижних уровнях — **листовые узлы**. Дерево может рассматриваться как семантическая сеть специального типа, в которой каждый узел, кроме корневого, имеет точно один **родительский узел** и нуль или больше **дочерних узлов**. Бинарное дерево обычного типа может содержать не больше двух дочерних узлов в расчете на каждый узел, причем левый и правый дочерние узлы различаются.

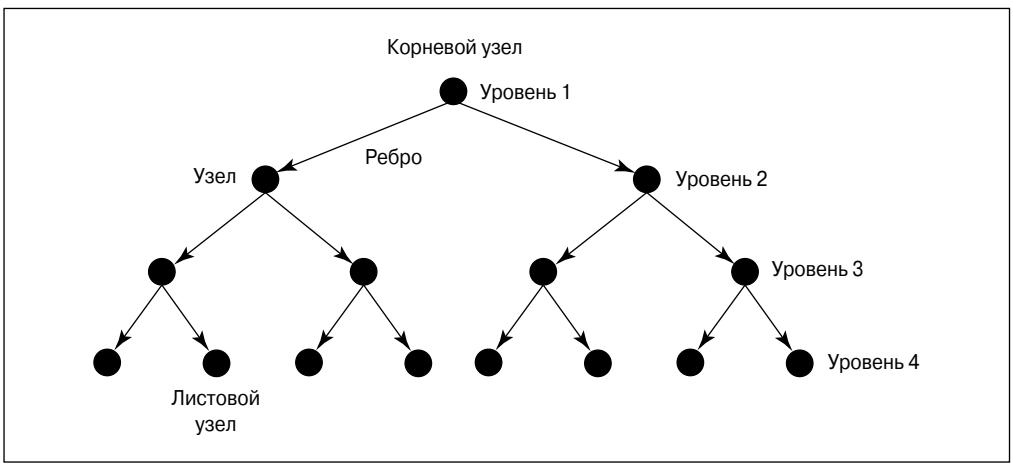


Рис. 3.1. Бинарное дерево

Если узел имеет больше одного родительского узла, то он находится в сети. На рис. 3.1 заслуживает внимания то, что имеется одна и только одна последовательность ребер (или **путь**) от корневого каталога до любого другого узла,

поскольку прохождение в направлении, противоположном указанному стрелками, не допускается. В ориентированных деревьях все стрелки указывают вниз.

Дерево — это частный случай общей математической структуры, называемой **графом**. При описании конкретного примера графа, такого как сеть Ethernet, термины *сеть* и *граф* часто используются как синонимы. В графе допускается наличие от нуля и больше связей между узлами, а родительские и дочерние узлы не различаются. Простым примером графа является карта автомобильных дорог, на которой города представляют собой узлы, а дороги — связи. Со связями могут быть ассоциированы направления, обозначенные стрелками, и **весовые коэффициенты**, характеризующие некоторые аспекты анализа связей. В качестве аналогии можно указать улицы с односторонним движением, для которых установлены предельные значения веса грузов, перевозимых на грузовиках. В качестве весовых коэффициентов в графах может быть задана информация любого типа. Например, если граф представляет авиамаршруты, то весовыми коэффициентами могут служить расстояния в милях между городами, стоимость перелета, расход топлива и т.д. Еще одним примером графа является контролируемая искусственная нейронная сеть. В этом графе присутствуют циклы, поскольку во время обучения, сопровождаемого передачей информации от одного слоя сети к другому, в результате чего происходит модификация весовых коэффициентов, возникает обратная связь. Как показано на рис. 3.2, *а*, простой граф не имеет связей, которые возвращаются непосредственно в сам узел, т.е. петель. **Контуром**, или **циклом**, называется путь через граф, который начинается и оканчивается в одном и том же узле; в качестве примера можно указать путь ABCA на рис. 3.2, *а*. **Ациклическим графом** называется граф, не имеющий циклов. **Связный граф** — это граф, в котором имеются связи, протянувшиеся ко всем узлам (см. рис. 3.2, *б*). А на рис. 3.2, *в* показан граф с ориентированными связями, называемый **ориентированным графом**, в котором имеется также **петля**. Ориентированный ациклический граф называется **решеткой**, пример решетки см. на рис. 3.2, *г*. Дерево с единственным путем от корня к его единственному листовому узлу называется **вырожденным деревом**. На рис. 3.2, *д* показаны вырожденные бинарные деревья с тремя узлами. Как правило, в дереве стрелки явно не показаны, поскольку подразумевается, что стрелки всегда направлены вниз.

Деревья и решетки имеют иерархическую структуру (в которой родительские узлы находятся на более высоких уровнях по сравнению с дочерними), поэтому являются удобным средством классификации объектов. Примером может служит генеалогическое дерево, которое показывает родственные связи и родословную близких людей. Еще одной областью применения деревьев и решеток является принятие решений; используемые при этом структуры называются **деревьями решений**, или **решетками решений**, и очень широко применяются для формирования простых рассуждений. В данной главе для обозначения и деревьев, и решеток будет использоваться термин **структура**. Структура, применяемая в процессе

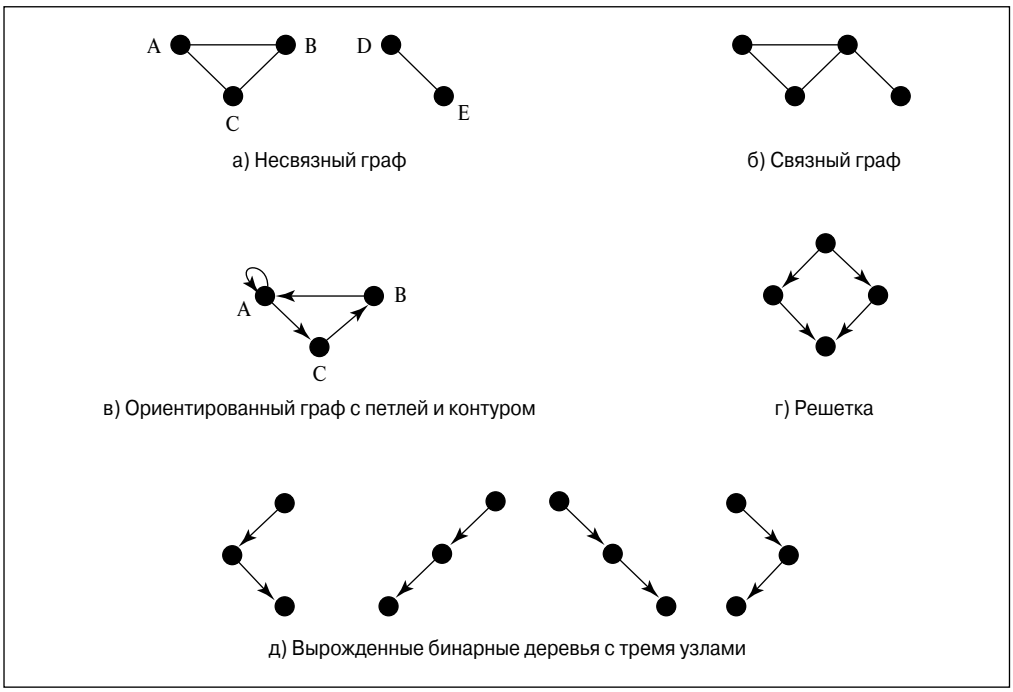


Рис. 3.2. Примеры простых графов

принятия решений, или структура решений, является одновременно и схемой представления знаний, и методом формирования рассуждений о содержащихся в ней знаниях. На рис. 3.3 показан пример дерева решений, предназначенного для классификации животных. Этот пример относится к классической игре, состоящей из двадцати вопросов. В узлах содержатся вопросы, ребра "да" и "нет" показывают возможные ответы на вопросы, а в листовых узлах приведены предположения, касающиеся того, как называется искомое животное.

С другой стороны, на рис. 3.4 представлена небольшая часть дерева решений, предназначенного для классификации кустарников малины различных видов. В отличие от деревьев, применяемых в компьютерных науках, допускается изображать деревья классификации с корневым узлом, находящимся в нижней части. В нижней части рисунка показан корень, от которого отходит ребро к узлу "Листья — простые" и еще одно ребро к узлу "Листья — сложные". Процесс принятия решения начинается с нижнего узла, с помощью которого выявляются наиболее важные характерные особенности, например, касающиеся того, являются ли листья простыми или сложными. А по мере продвижения вверх по дереву приходится выяснять более конкретные подробности, требующие более внимательного наблюдения. Это означает, что вначале исследуются более крупные множества

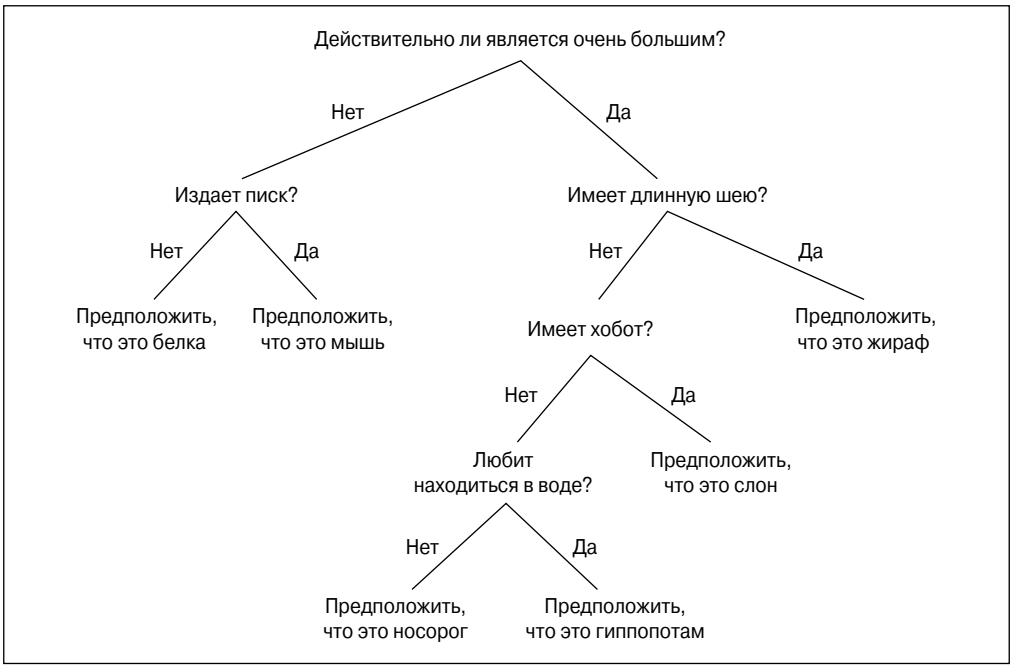


Рис. 3.3. Дерево решений, на котором представлены знания о животных

альтернатив, а затем процесс принятия решений начинает сходиться к тем возможностям, которые представлены с помощью все меньших и меньших множеств. Это — удобный способ организации процесса принятия решений с точки зрения затрат времени и усилий для проведения все более детальных наблюдений.

Если все решения являются бинарными, то появляется возможность легко построить бинарное дерево решений, которое становится очень эффективным. Для ответа на каждый вопрос приходится переходить вниз (или вверх) на один уровень дерева. Один вопрос позволяет принять решение о выборе одного из двух возможных ответов, с помощью двух вопросов вырабатывается решение о выборе одного из четырех возможных ответов, три вопроса позволяют выбрать один из восьми возможных ответов и т.д. Если бинарное дерево сконструировано таким образом, что во всех листовых узлах представлены ответы, а на всех ребрах, ведущих вниз, заданы вопросы, то дерево решений позволяет представить максимальное количество ответов на N вопросов, равное 2^N. Например, десять вопросов позволяют классифицировать одно из 1024 животных, а двадцать — находить один из 1 048 576 возможных ответов.

Еще одной полезной особенностью деревьев решений является то, что они могут быть **самообучающимися**. Если предположение оказывается ошибочным, то вызывается процедура, с помощью которой пользователю передается просьба,

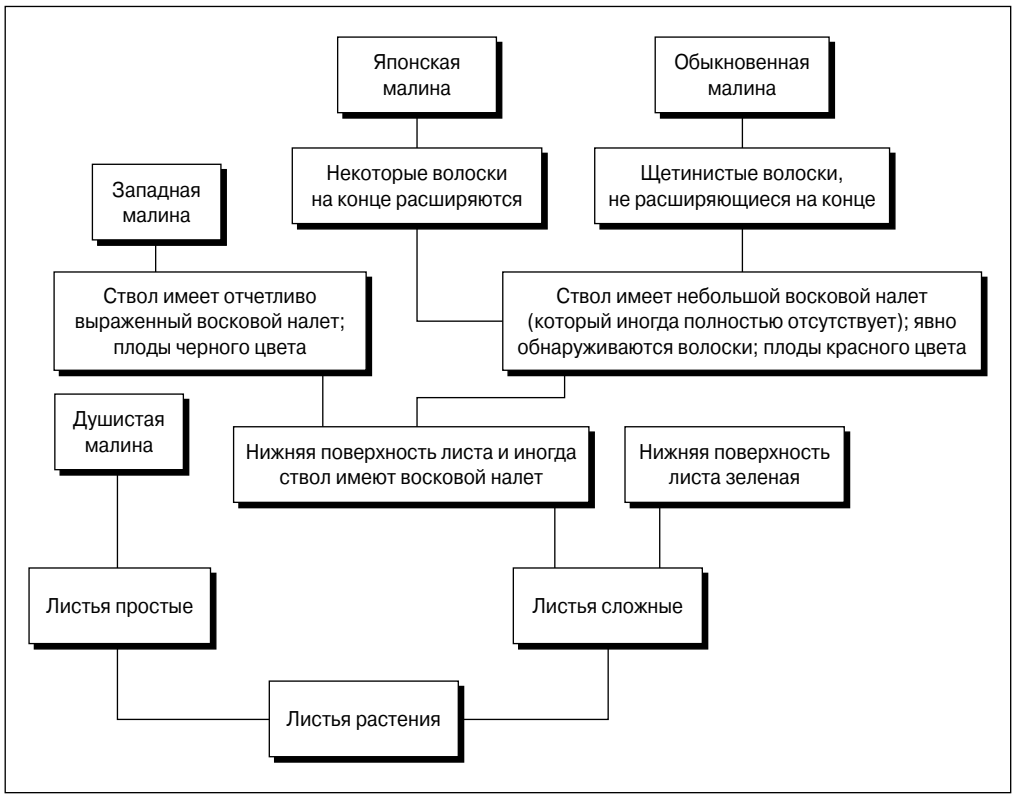


Рис. 3.4. Часть дерева решений, предназначенного для классификации кустарников малины

чтобы он указал новый, правильный классификационный вопрос и дал ответы, связанные с вариантами выбора “да” и “нет”. После этого динамически создаются новые родительский узел, ребра и листовые узлы, которые добавляются к дереву. В первоначальной программе Animals, написанной на языке BASIC, знания хранились в операторах `DATA`. После того как пользователь обучал программу распознаванию нового животного, происходила автоматическая регистрация результатов обучения, в ходе которой программа формировала новые операторы `DATA`, содержащие информацию о новом животном. Для обеспечения максимальной эффективности знания о животных хранились в виде деревьев. А язык CLIPS позволяет создавать автоматически новые правила (в ходе того как программа усваивает новые знания) с помощью ключевого слова **build** [33]. Такая процедура **автоматизированного приобретения знаний** является очень полезной, поскольку в простых случаях позволяет преодолеть узкое место в приобретении знаний, которое будет рассматриваться в главе 6.

Структуры решений могут быть механически преобразованы в продукционные правила. Такая процедура легко осуществляется с помощью поиска в ширину в дереве и выработки в каждом узле правил IF–THEN. Например, дерево решений, приведенное на рис. 3.3, может быть частично преобразовано в правила следующим образом:

```
IF QUESTION = "IS IT VERY BIG?" AND RESPONSE = "NO"
   THEN QUESTION := "DOES IT SQUEAK?"

IF QUESTION = "IS IT VERY BIG?" AND RESPONSE = "YES"
   THEN QUESTION := "DOES IT HAVE A LONG NECK?"
```

Аналогичная процедура выполняется и для других узлов. При этом в листовом узле вырабатывается не вопрос, а ответ ANSWER. Кроме того, могут быть предусмотрены соответствующие процедуры, передающие пользователю запросы на ввод данных и позволяющие создавать новые узлы при обнаружении случаев неправильной классификации.

Безусловно, структуры решений являются очень мощными инструментальными средствами классификации, но их возможности ограничены, поскольку структуры решений, в отличие от экспертных систем, не позволяют использовать переменные. Экспертные системы — это инструментальные средства общего назначения, а не простые классификаторы.

3.3 Пространства состояний и пространства задач

Для решения многих практических задач могут применяться графы. Один из удобных методов описания поведения некоторого объекта состоит в определении графа, называемого **пространством состояний**. Состояние — это коллекция характеристик, которые могут использоваться для определения **состояния**, или статуса, объекта. Пространство состояний — это множество состояний, с помощью которого представляются **переходы** между состояниями, которым подвергается объект. В результате перехода объект попадает из одного состояния в другое.

Примеры пространств состояний

В качестве простого примера использования пространства состояний рассмотрим покупку безалкогольного напитка в автомате. По мере того как в автомат вкладывают монеты, происходит переход из одного состояния в другое. На рис. 3.5 показана диаграмма пространства состояний, подготовленная на основе того предположения, что в распоряжении покупателя имеются только 25-центовые монеты (Q) и 5-центовые монеты (N), а для приобретения напитка необходимо опустить

в автомат 55 центов. Если бы мы предусмотрели возможность использовать монеты другого достоинства, например 10- и 50-центовые, то диаграмма стала бы сложнее, поэтому они здесь не показаны.

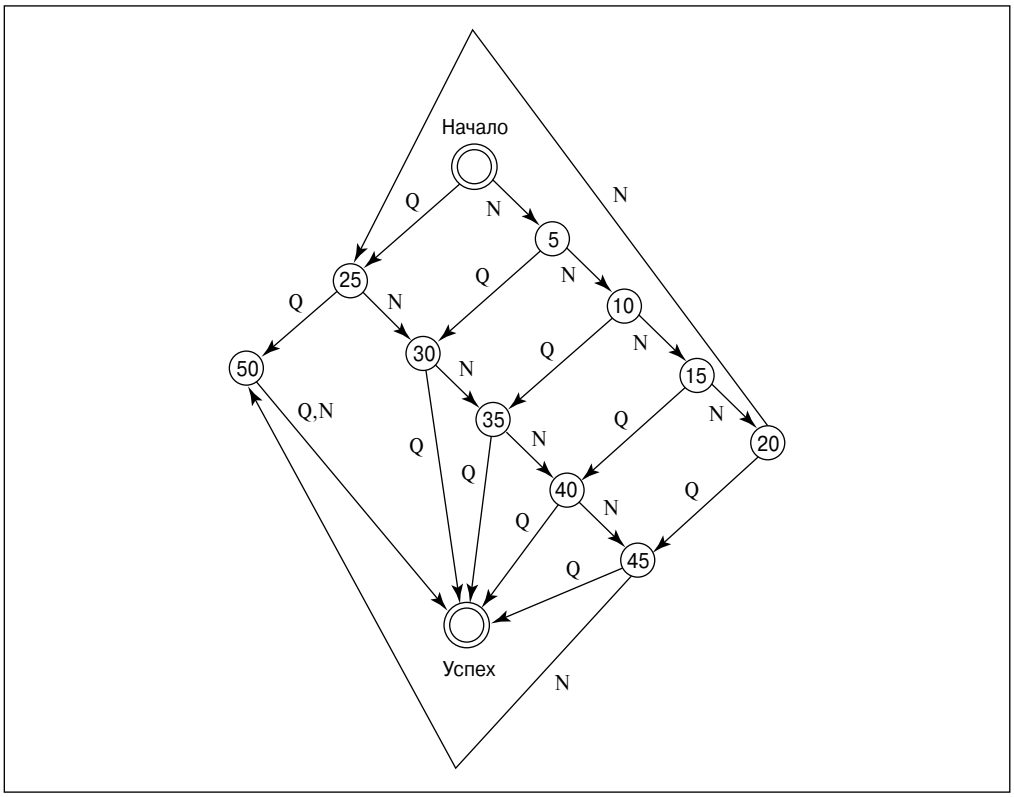


Рис. 3.5. Диаграмма состояний автомата по продаже безалкогольных напитков, который принимает 25-центовые монеты (Q) и 5-центовые монеты (N)

Состояния обозначены кружками, а возможные переходы в другие состояния — стрелками. Начальное состояние и состояние успеха обозначены двойными кружками, чтобы их было проще отличать от других состояний. Обратите внимание на то, что данная диаграмма представляет собой взвешенный ориентированный граф, в котором весовыми коэффициентами являются монеты разного достоинства, которые могут быть введены в автомат в каждом состоянии.

Эта диаграмма называется также **диаграммой конечного автомата**, поскольку описывает конечное количество состояний автомата. Термин *автомат* используется в очень общем смысле. Автоматом может быть реальный объект, алгоритм, концепция и т.д. С каждым состоянием связаны действия, позволяющие перевести автомат в другое состояние. Но в каждый конкретный момент времени данный автомат может находиться только в одном состоянии. После того как автомат при-

нимает входные данные в каком-либо состоянии, он переходит из этого состояния в другое. Если даны только правильные входные данные, то автомат переходит из начального состояния в состояние успеха, или в конечное состояние. А если некоторое состояние не рассчитано на получение определенных входных данных, то автомат зависает в этом состоянии. Например, данный конечный автомат по продаже безалкогольных напитков не рассчитан на получение монет достоинством в 10 центов. Поэтому если кто-то вложит в автомат 10-центовую монету, то ответ будет неопределенным. Это означает, что качественный проект должен предусматривать возможность неправильного ввода данных в любом состоянии и в связи с этим переход в состояние ошибки. А состояние ошибки проектируется таким образом, чтобы в нем выдавались соответствующие сообщения об ошибках и предпринимались все необходимые корректирующие действия.

Конечные автоматы часто используются в компиляторах и других программах для определения того, являются ли действительными входные данные. Например, на рис. 3.6 показана часть конечного автомата, позволяющего проверить, являются ли действительными входные строки. Символы входных строк проверяются один за другим. Автомат принимает только символьные строки `WHILE`, `WRITE` и `BEGIN`. Показаны стрелки, отходящие от состояния `BEGIN` и соответствующие успешно принятым входным данным, а также стрелки, которые обеспечивают переход в состояние ошибки под действием неправильных входных данных. В целях повышения эффективности некоторые состояния (например, то, на которое указывают стрелки "*L*" и "*T*") используются для проверки и строки `WHILE`, и строки `WRITE`.

Примечание. Показаны только некоторые из переходов в состояние ошибки.

Диаграммы состояний являются также полезным средством описания решений задач. В приложениях такого рода пространство состояний может рассматриваться как **пространство задач**, в котором одни состояния соответствуют промежуточным этапам решения задач, а другие соответствуют ответам. В пространстве задач может быть несколько состояний успеха, соответствующих возможным решениям. Для поиска решения задачи в пространстве задач необходимо найти действительный путь от начального узла (формулировки задачи) до узла успеха (ответа). Дерево решений, применяемое для классификации животных, может рассматриваться как пространство задач, в котором ответы "да" или "нет" соответствуют вопросам, определяющим переходы между состояниями.

Еще один пример пространства задач обнаруживается в решении классической задачи с обезьяной и бананами, которая показана на рис. 3.7. Задача состоит в том, чтобы дать обезьяне инструкции с указаниями, как получить бананы, свисающие с потолка. Бананы находятся вне досягаемости обезьяны. В комнате имеются кушетка и лестница. Исходная начальная конфигурация обычно предусматривает

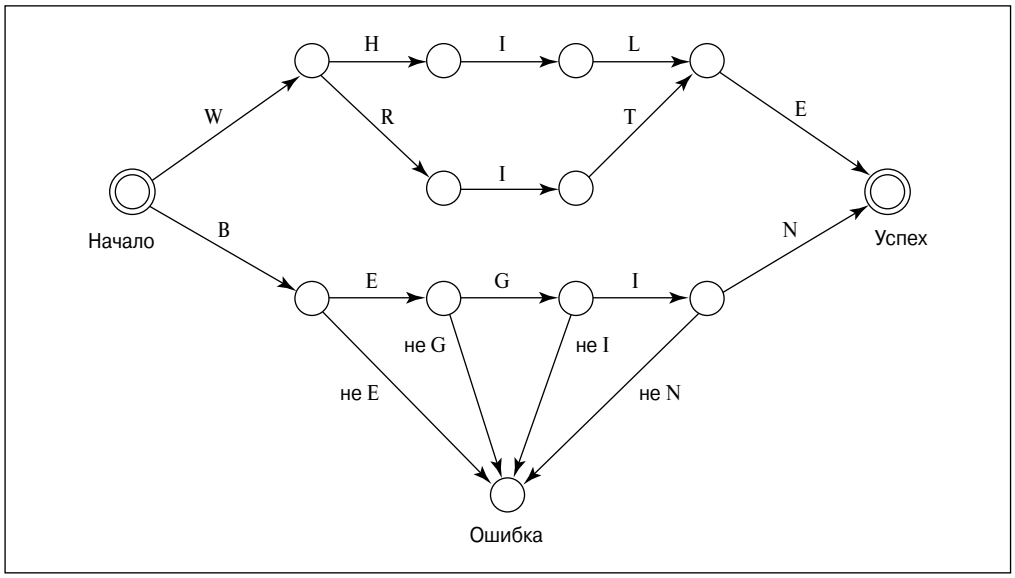


Рис. 3.6. Часть конечного автомата для определения допустимых строк `WHILE`, `WRITE` и `BEGIN`

такое положение, в котором обезьяна находится на кушетке. Инструкции могут состоять в следующем:

```
спрыгнуть с кушетки
подойти к лестнице
передвинуть лестницу туда, где находятся бананы
вскарабкаться на лестницу
схватить бананы
```

Инструкции могут изменяться в зависимости от начальной конфигурации, в которой находятся обезьяна, кушетка и лестница. Количество начальных состояний велико, поэтому на диаграмме не показаны специальные двойные кружки, обозначающие начальное состояние. Например, еще одним возможным начальным состоянием является то, в котором обезьяна находится на кушетке, стоящей под бананами. В таком случае обезьяна должна убрать кушетку с этого места, прежде чем пододвинуть лестницу под бананы. А в простейшем начальном состоянии обезьяна уже находится на лестнице, стоящей под бананами.

Безусловно, решение указанной задачи для человека кажется очевидным, но оно требует значительного объема рассуждений. Практическим приложением системы формирования рассуждений, подобных этой, является выдача инструкций для робота, касающихся решения задачи. Общим решением должна стать система формирования рассуждений, которая не исходит из того предположения, что все объекты в рассматриваемой среде постоянно находятся на одном и том же ме-

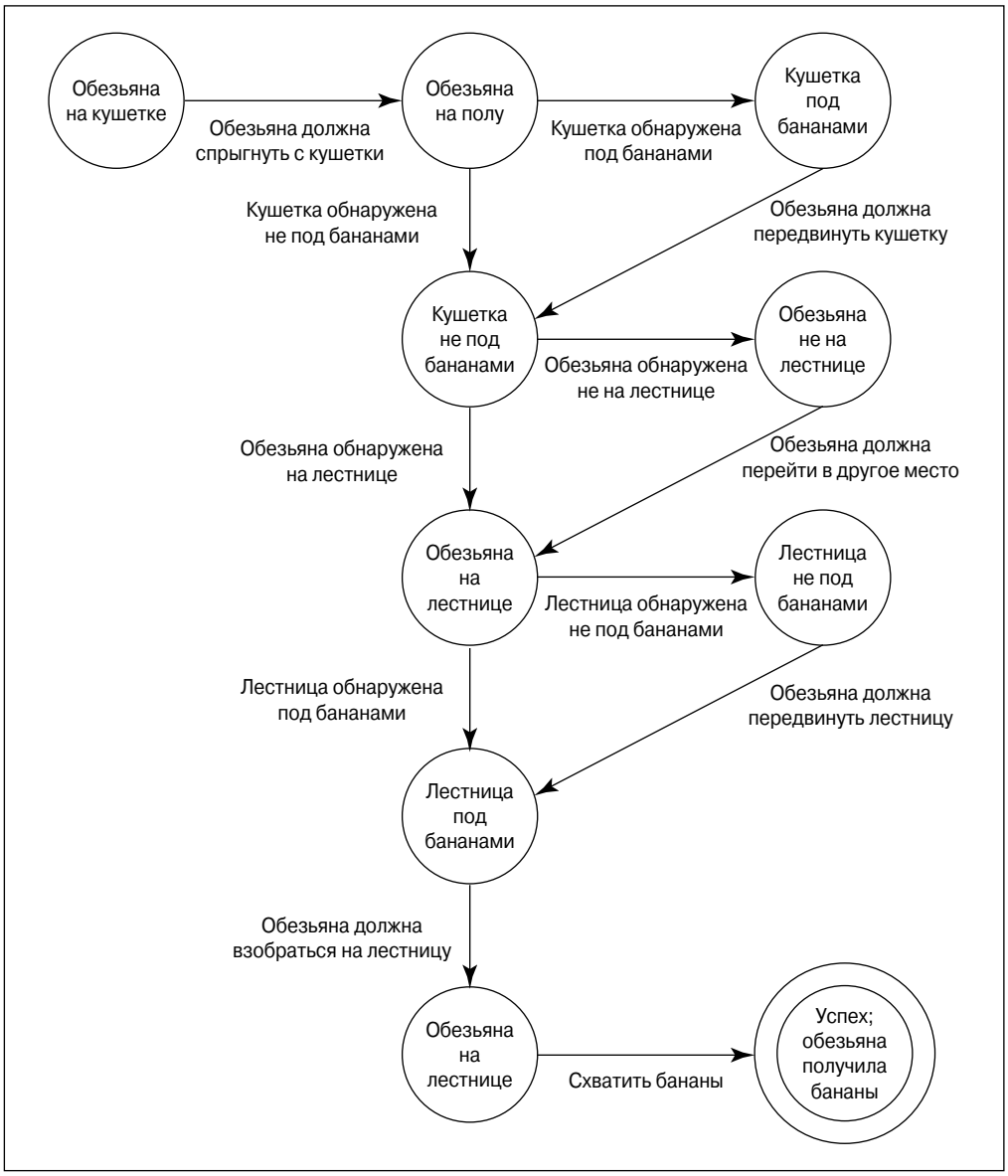


Рис. 3.7. Пространство состояний для задачи с обезьяной и бананами

сте, а обеспечивает возможность справляться с самыми различными ситуациями. Решение задачи с обезьяной и бананами, основанное на использовании правил, распространяется вместе с компакт-диском с программным обеспечением CLIPS, прилагаемым к данной книге.

Еще одним полезным приложением графов является исследование путей для поиска решения некоторой задачи. На рис. 3.8, *а* приведена простая сеть для задачи коммивояжера. Предположим, что в этом примере задача состоит в том, чтобы найти полный путь от узла A, в котором посещаются все прочие узлы. Как обычно предусмотрено в задаче коммивояжера, ни один узел не должен посещаться дважды. На рис. 3.8, *б* в форме дерева приведены все возможные пути, начинающиеся от узла A. На этом графе жирными линиями показаны правильные пути $ABDCA$ и $ACDBA$.

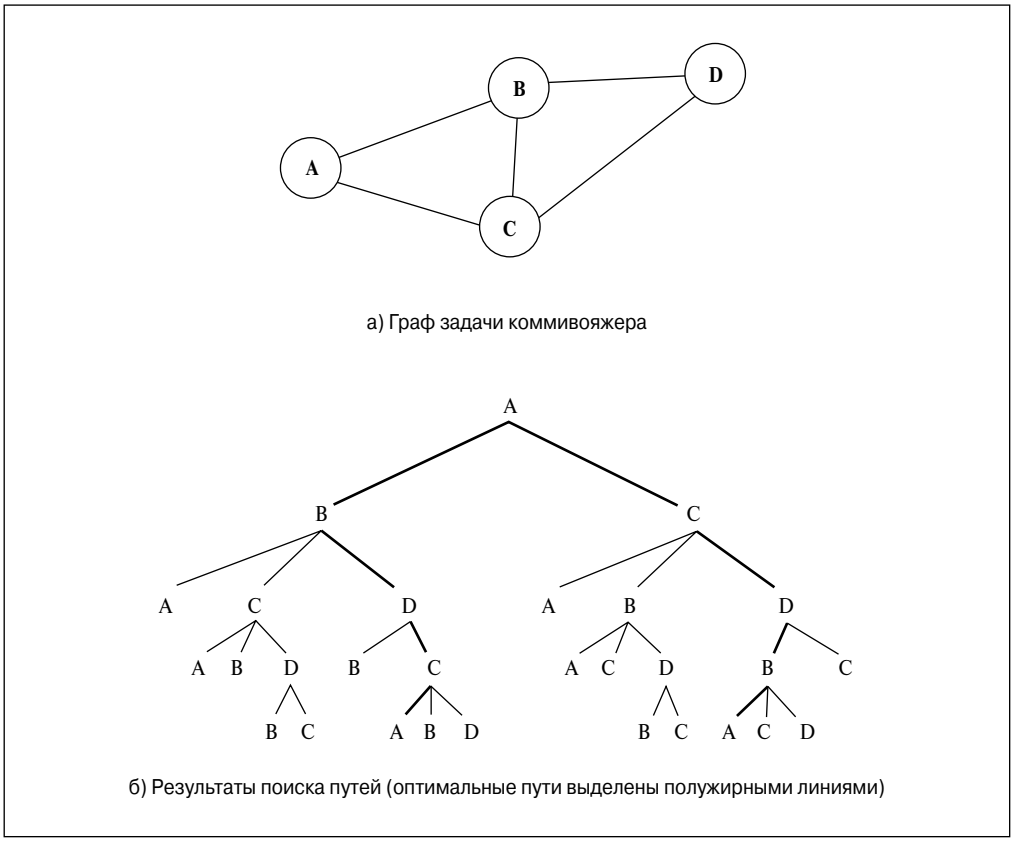


Рис. 3.8. Задача коммивояжера

В зависимости от алгоритма поиска процесс исследования путей, выполняемого в целях поиска правильного пути, может потребовать выполнения значительного объема перебора с возвратами. Например, предположим, что вначале был выбран неудачный путь ABA, затем выполнен возврат к узлу B, а от узла B безуспешно проведен поиск с использованием путей CA, CB, CDB и CDC.

После этого безуспешно проводится поиск с использованием пути BDB, и так до тех пор, пока не обнаруживается первый правильный путь $ABDCA$.

Пространства слабо структурированных задач

Полезным приложением пространств состояний является представление слабо структурированных задач. В главе 1 слабо структурированная задача определена как задача, с которой ассоциируются какие-либо неопределенности. Эти неопределенности могут быть описаны более точно с помощью пространства задач.

В качестве примера слабо структурированной задачи еще раз рассмотрим случай, в котором некоторое лицо собирается отправиться в путешествие и, не сумев найти ничего подходящего с помощью поиска в оперативном режиме, обращается в бюро путешествий (см. главу 1). В табл. 3.1 перечислены некоторые характеристики этой слабо структурированной задачи как пространства задач с указанием ответов рассматриваемого лица на вопросы агента бюро путешествий.

Таблица 3.1. Пример слабо структурированной задачи выбора путешествия

Характерная особенность	Ответ
Цель не ясна	Я собираюсь куда-нибудь поехать
Неограниченное пространство состояний	Я еще не решил, в какое место отправиться
Не дискретное пространство задач	Я просто люблю путешествовать; место назначения не имеет значения
Промежуточные состояния трудно достижимы	Я не имею достаточного количества денег, чтобы отправиться куда угодно
Набор возможных операций в пространстве состояний неизвестен	Я не знаю, как получить деньги
Временно́е ограничение	Я должен поскорее уехать в путешествие

Сравнение табл. 3.1 с табл. 1.10, приведенной в главе 1, показывает, что концепция пространства задач позволяет более точно задавать характеристики слабо структурированной задачи. Очень важно точно охарактеризовать такие параметры, чтобы определить, осуществимо ли решение, а в случае положительного ответа узнать, что требуется для получения решения. Задачу не следует обязательно считать слабо структурированной лишь потому, что в ней имеются одна, несколько или даже все подобные характеристики, поскольку многое зависит от того, насколько строгим является подход к постановке задачи. Например, все задачи доказательства теорем имеют бесконечное количество потенциальных решений, но из-за этого задача доказательства теорем не становится слабо структурированной.

Как показывает табл. 3.1, при решении задачи выбора путешествия возникает много неопределенностей, тем не менее, агенты бюро путешествий повседневно справляются с подобными задачами. Безусловно, не все ситуации могут быть такими же безвыходными, как указанная, но сам характер задачи показывает, почему поиск алгоритмического решения был бы очень трудным.

Правильно сформулированной задачей является такая задача, в отношении которой известна явная постановка задачи, цель решения и операции, которые могут применяться для перехода из одного состояния (этапа) решения в другое. Правильно сформулированная задача является **детерминированной**, поскольку после применения некоторой операции к некоторому состоянию можно с уверенностью сказать, каким будет следующее состояние. Пространство задач является ограниченным, а состояния — дискретными. Это означает, что имеется конечное количество состояний и все состояния определены полностью.

В описанной выше задаче выбора путешествия пространство состояний является неограниченным, поскольку имеется бесконечно большое количество возможных пунктов назначения, в которые может отправиться путешественник. Аналогичная ситуация возникает при использовании аналогового измерительного прибора, который может показывать бесконечное количество возможных результатов измерения. Если каждый результат измерения, полученный от такого измерительного прибора, рассматривается как состояние, то количество возможных состояний бесконечно велико, а сами состояния не могут быть полностью определены, поскольку положения стрелки измерительного прибора соответствуют вещественным числам. Между любыми двумя вещественными числами имеется бесконечное количество вещественных чисел, поэтому состояния не являются дискретными, и одно состояние может отличаться от другого лишь на бесконечно малую величину. В отличие от этого показания цифрового измерительного прибора являются дискретными и относятся к ограниченному пространству состояний.

3.4 Деревья AND–OR и цели

Применение обратного логического вывода для поиска решения задач предусмотрено в экспертных системах многих типов. Наглядным примером системы обратного логического вывода является система PROLOG, в которой предпринимаются попытки решать любые задачи путем разбиения их на меньшие подзадачи и последующего решения подзадач. В 1990-х годах система PROLOG получила широкое распространение при создании коммерческих приложений в бизнесе и промышленности (`http://www.ddj.com/documents/s=9064/ddj0212ai004/0212aie004.htm`). Оптимисты рассматривают процесс решения задачи как процесс достижения цели, но для достижения достаточно значимой цели иногда приходится рассматривать много подцелей.

К одному из типов деревьев (или решеток), которые являются удобным средством представления задач обратного логического вывода, относятся деревья (или решетки) AND–OR. На рис. 3.9 показан простой пример решетки AND–OR, предназначенной для достижения цели получения документа об окончании колледжа. Для достижения этой цели необходимо либо лично посещать колледж, либо поступить на заочные курсы. При прохождении заочных курсов работу можно выполнять либо по заданиям, полученным по почте, либо в оперативном режиме с использованием домашнего компьютера и модема.

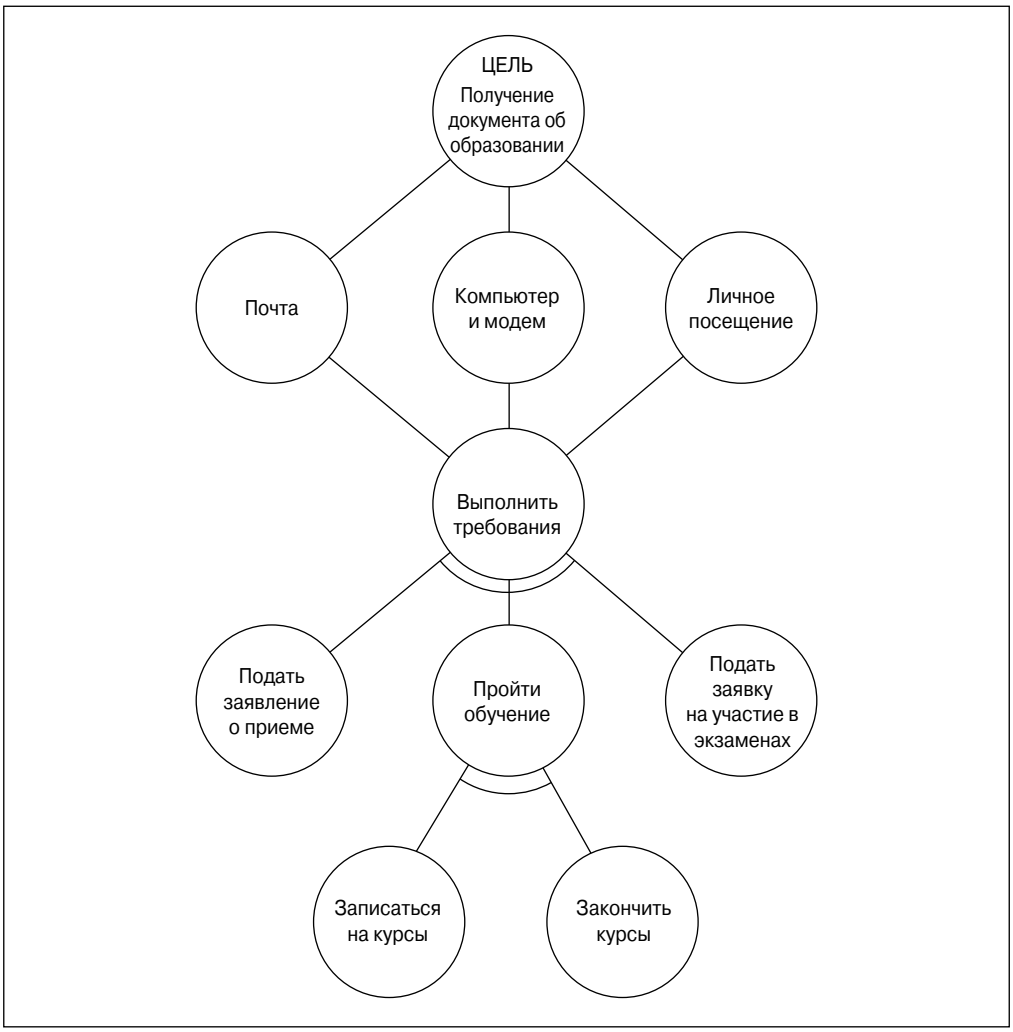


Рис. 3.9. Решетка AND–OR, показывающая, как получить документ об окончании колледжа

Чтобы выполнить требования, необходимые для получения документа об образовании, необходимо достичь трех подцелей: во-первых, подать заявление, во-вторых, пройти обучение и, в-третьих, сдать экзамены. Обратите внимание на то, что ребра, ведущие от цели "Выполнить требования" к этим трем подцелям, соединены отрезком кривой. Этот отрезок кривой, проведенный через ребра, показывает, что узел "Выполнить требования" — это узел AND, поэтому заданная в нем цель может быть выполнена, только если будут выполнены все три его подцели. А такие цели, не обозначенные отрезками кривых, как "Почта", "Компьютер и модем" и "Личное посещение", заданы узлами OR, в которых выполнение любой из указанных подцелей позволяет достичь родительской цели — "Получение документа об образовании".

Структура, показанная на этой диаграмме, представляет собой решетку, поскольку подцель "Выполнить требования" имеет три родительских узла: "Почта", "Компьютер и модем" и "Личное посещение". Обратите внимание на то, что было бы возможно изобразить эту диаграмму в виде дерева, просто продублировав подцель "Выполнить требования" и его поддерево целей, относящееся к целям "Почта", "Компьютер и модем" и "Личное посещение". Тем не менее цель "Выполнить требования" остается одинаковой для каждой из ее родительских целей, поэтому построение такого дерева не дает никаких реальных преимуществ и приводит к созданию более громоздкой диаграммы.

В качестве еще одного простого примера на рис. 3.10 показано дерево AND–OR для решения задачи прибытия на работу с помощью различных возможных способов. Для полноты изложения следует отметить, что это дерево может быть также преобразовано в решетку. Для этого, например, можно добавить ребро от узла "Приехать на вокзал" к узлу "На автомобиле" и от узла "Пройти на вокзал" к узлу "Пешком". А на рис. 3.11 показана решетка типа AND-exclusive OR (И–исключительное ИЛИ).

Еще один способ описания решений задач состоит в использовании решетки AND–OR–NOT, в которой применяются символические обозначения логических вентилей, а не такая система обозначений, как в деревьях AND–OR. Обозначения логических вентилей, используемых для выполнения операций AND, OR и NOT, показаны на рис. 3.12. Эти логические элементы реализуют истинностные таблицы для операций AND, OR и NOT, которые рассматривались в главе 2. А на рис. 3.13 показан вариант диаграммы, приведенной на рис. 3.9, в котором применяются логические элементы AND и OR.

Деревья AND–OR и деревья решений имеют одни и те же основные преимущества и недостатки. Основным преимуществом решеток AND–OR–NOT является то, что они допускают возможность реализации в виде аппаратных средств, что позволяет добиваться высоких скоростей обработки. Такие решетки могут быть специально спроектированы в целях дальнейшего изготовления в виде интегральных микросхем. На практике для экономии производственных затрат используется

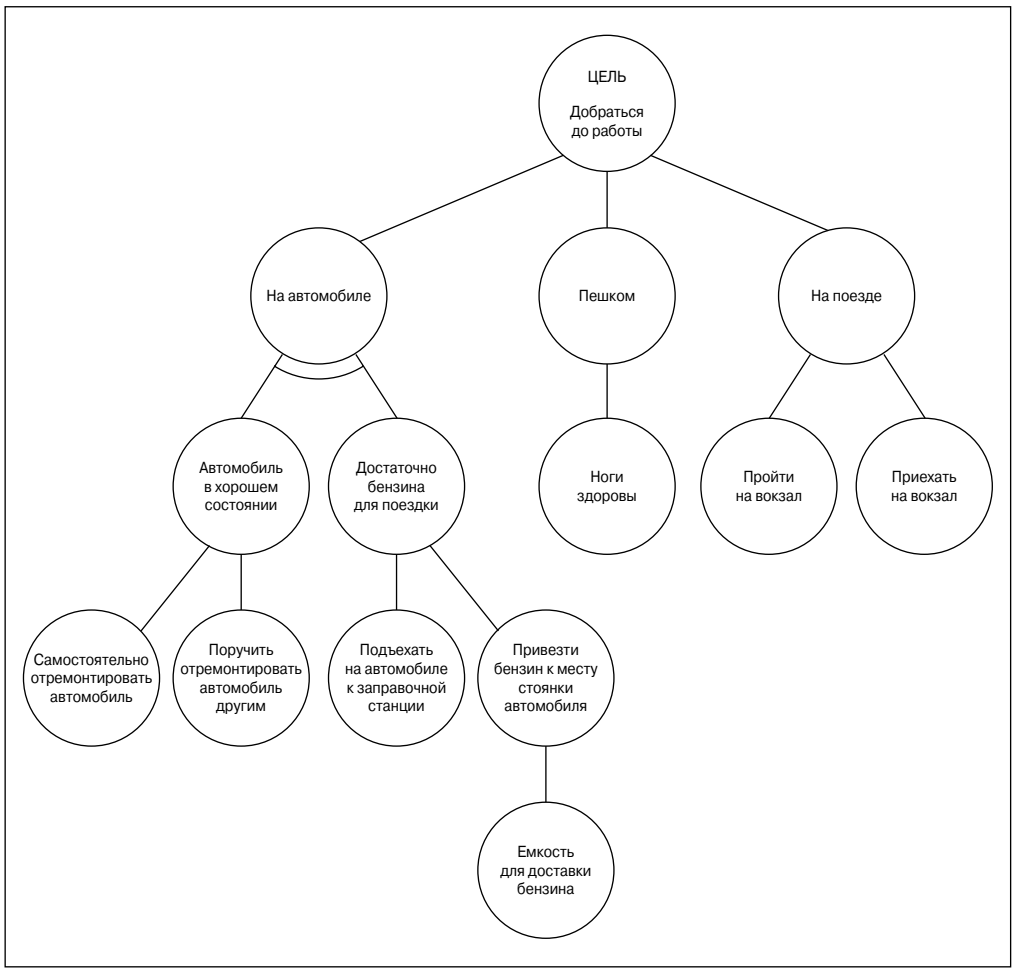


Рис. 3.10. Простое дерево AND–OR, показывающие, как добраться до места работы с помощью различных способов

только один тип логических вентилей, такой как NOT AND (или NAND), а не отдельно взятые логические вентили AND, OR и NOT. На основании принципов логики можно доказать, что с помощью логического элемента NAND может быть реализована любая логическая функция. А производство интегральных микросхем, состоящих из устройств одного и того же типа, дешевле, чем производство микросхем, состоящих из логических вентилей нескольких типов.

Микросхема, в которой используется прямой логический вывод, способна вычислять ответ как функцию от полученных входных данных очень быстро, поскольку обработка данных происходит в ней параллельно. Микросхемы подобного типа могут использоваться для текущего контроля данных, полученных от датчи-

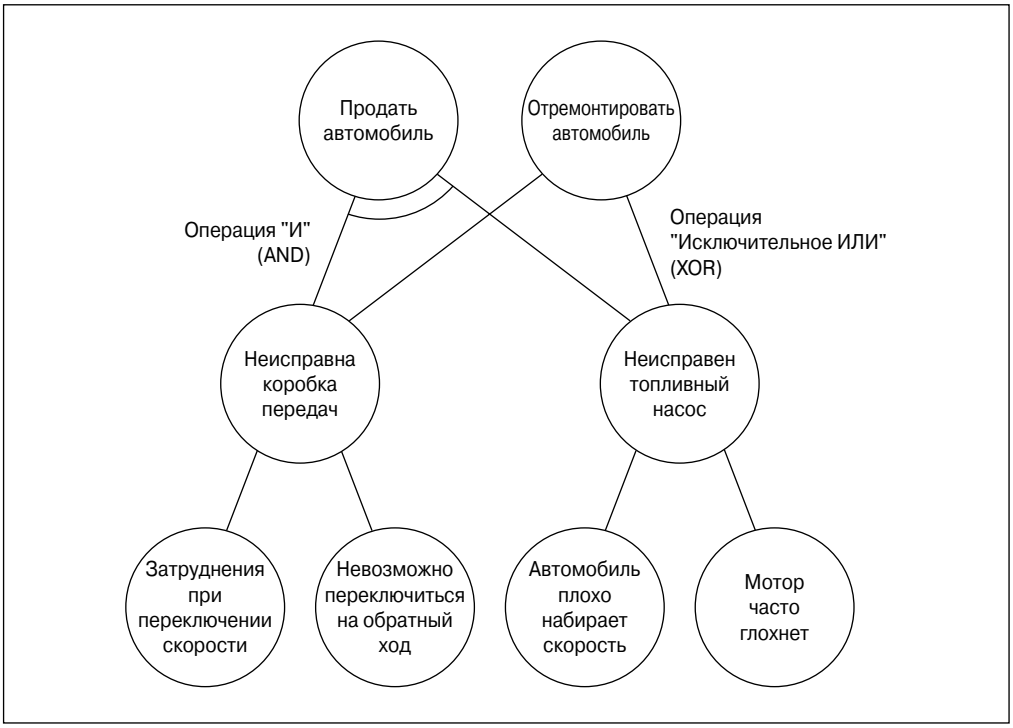


Рис. 3.11. Решетка AND–OR для принятия решения о том, следует ли продать или отремонтировать свой автомобиль

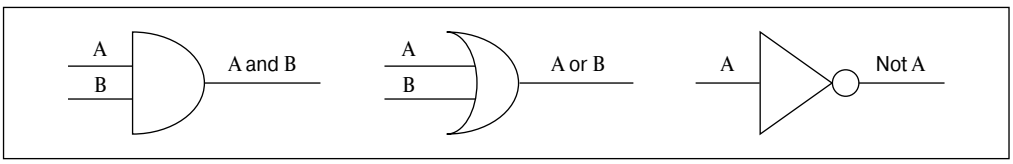


Рис. 3.12. Символические обозначения логических вентилей AND, OR и NOT

ка, в реальном времени и выработки соответствующих результатов в зависимости от входных данных. Основным недостатком рассматриваемого подхода является то, что микросхемы, предназначенные для выполнения логических операций, как и другие структуры принятия решений, не могут справляться с ситуациями, на которые они не рассчитаны. С другой стороны, искусственная нейронная система, реализованная в виде микросхемы, способна успешно обрабатывать непредвиденные входные данные.

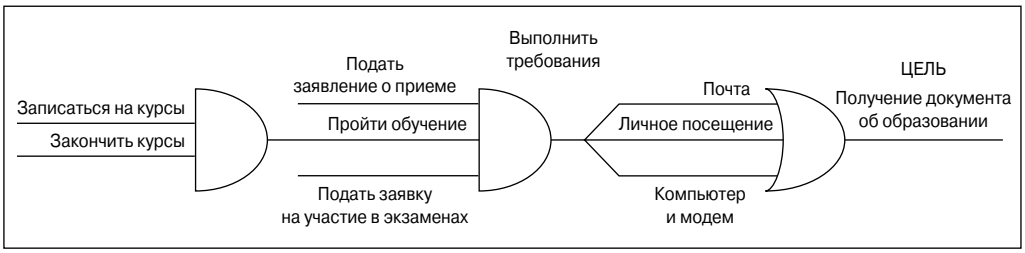


Рис. 3.13. Вариант диаграммы, представленной на рис. 3.9, в котором используются логические вентили AND и OR

3.5 Дедуктивная логика и силлогизмы

В главе 2 рассматривались способы представления знаний, основанные на логике, а в данном разделе речь идет о том, как осуществляются логические выводы для получения новых знаний или информации. В оставшейся части этой главы рассматриваются различные методы логического вывода. Общий обзор методов логического вывода показан на рис. 3.14. Краткие сведения по этой теме приведены ниже.

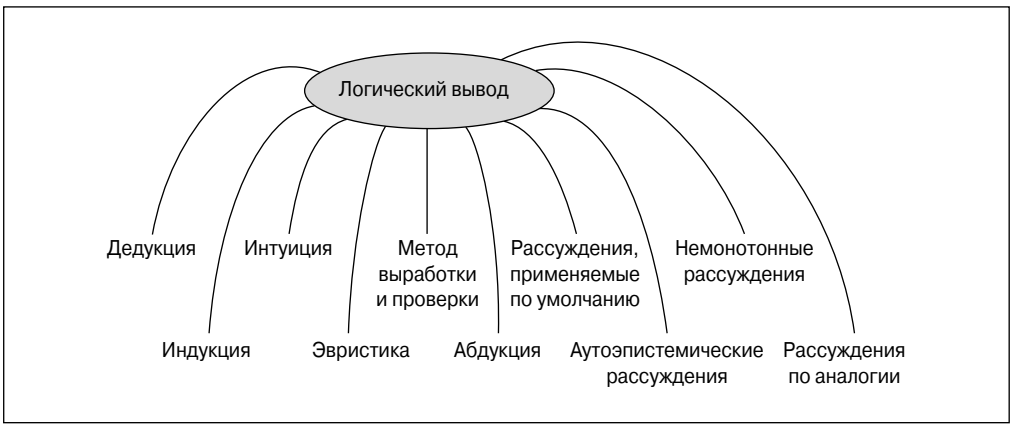


Рис. 3.14. Типы логического вывода

- Дедукция. Логическое рассуждение, в котором заключения должны следовать из соответствующих им посылок.
- Индукция. Логический вывод от частного случая к общему. Индукция — это один из основных методов машинного обучения, в котором компьютеры обучаются без вмешательства человека. Три широко применяемых метода машинного обучения основаны на использовании коннекционистской сети Хопфилда, символических алгоритмов ID3/ID5R и эволюционных генетиче-

ских алгоритмов (`http://ai.bpa.arizona.edu/papers/mlir93/mlir93.html`).

- Интуиция. Метод, не основанный на проверенной теории. Ответ представляет собой всего лишь предположение, возможно, сформулированное путем подсознательно распознавания какого-то основополагающего образа. Логический вывод такого типа еще не реализован в экспертных системах. Но искусственные нейронные системы когда-либо смогут оправдать прогнозы, касающиеся их использования для осуществления логического вывода такого типа, поскольку эти системы способны выходить за рамки того, что было усвоено ими в результате обучения, а не просто предоставлять обусловленный ситуацией ответ или выполнять интерполяцию. Это означает, что нейронная сеть всегда вырабатывает свое наилучшее возможное предположение, касающееся искомого решения.
- Эвристика. Эмпирические правила, основанные на опыте.
- Метод выработки и проверки — метод проб и ошибок. Часто используется в сочетании с планированием для достижения максимальной эффективности.
- Абдукция. Метод формирования рассуждений в обратном направлении, от истинного заключения к посылкам, которые могли привести к получению этого заключения.
- Рассуждения, применяемые по умолчанию. Рассуждения, которые допускают возможность в отсутствие конкретных знаний принимать по умолчанию общепринятые или общеизвестные знания.
- Аутоэпистемические рассуждения. Самопознание, или рассуждения человека о том, каким ему, например, кажется цвет неба.
- Немонотонные рассуждения. Рассуждения, применяемые в тех условиях, когда ранее полученные знания могут оказаться неправильными после получения нового свидетельства.
- Рассуждения по аналогии. Логический вывод заключения исходя из наличия признаков, подобных признакам другой ситуации. В нейронных сетях рассуждения по аналогии выполняются путем распознавания образов в данных и последующего распространения полученных результатов на новую ситуацию.

На рис. 3.14 явно не показан еще один метод рассуждений, в котором используются **знания, полученные на основе здравого смысла**; этот метод представляет собой сочетание всех прочих методов логического вывода и основан на собственном опыте человека. Люди используют рассуждения на основе здравого смысла во многих обычных ситуациях, но задача реализации такого метода формирования рассуждений в компьютерах является очень сложной. Весьма важная попытка

создать огромную базу данных, содержащую знания, полученные на основе здравого смысла и пригодные для применения в компьютерах, осуществляется Дугом Ленатом (Doug Lenat), начиная с 1980-х годов, а к настоящему времени уже созданы практические приложения (`http://www.OpenCyc.org`). Эта база данных состоит из многих утверждений и правил, которые могут использоваться для более успешного создания самых разных приложений, начиная с лучшего понимания речи и заканчивая более эффективным построением онтологий. Любопытная особенность этой базы данных состоит в том, что не всегда оправдывается старая поговорка "Чем больше, тем лучше". В следующем сообщении (полученном в ходе личной переписки по электронной почте) Дуг Ленат объяснил, с чем это связано.

> Вручную были введены 3 миллиона правил; благодаря обобщению и факторизации к настоящему времени мы смогли уменьшить это количество до 1,5 миллиона. Мы пытаемся добиться не максимального увеличения, а максимального уменьшения количества правил. В ином случае мы могли бы легко увеличить это количество так, чтобы правил было, например, больше миллиарда. Рассмотрим 2 типа животных — крысу и верблюда. К крысе относится также правило, что она не является верблюдом. А поскольку нам известно 10 тысяч животных, то мы могли бы написать 100 миллионов правил такого рода. Вместо этого в системе Cyc имеется 10 тысяч и одно правило: линнеевские таксономические отношения плюс одно правило, в котором сказано, что два таксона, применительно к которым неизвестно, что они состоят в отношении genl/spec (род/вид), являются непересекающимися. Говоря о "десятках тысяч дополнительных правил", вы фактически ведете речь об одном правиле весьма специального типа, а именно о правиле с определенным уровнем сложности, согласно которому НИ ОДНО из упомянутых выше утверждений (например, что крыса — не верблюд) вообще не может рассматриваться как правило.
>
> Количество утверждений в нашей базе знаний составляет всего лишь пару миллионов благодаря тому, что мы пользуемся возможностью избавиться от остальных правил с помощью самого́ информационного наполнения. Мы НЕ СТРЕМИМСЯ увеличивать количество правил, поскольку из-за этого уменьшилась бы производительность, но если бы мы ХОТЕЛИ, чтобы база знаний стала больше, то могли бы взять за основу 10 тысяч и одно из этих (двух миллионов) утверждений в текущей базе знаний и эквивалентно заменить их 100 миллионами правил, которые только внешне кажутся немного менее общими. Это было бы неправильно.

Способы применения нечеткой логики для формирования рассуждений на основе здравого смысла рассматриваются в главе 5.

Одним из наиболее часто применяемых методов формирования логических выводов является **дедуктивная логика**, которая с древних времен использовалась для определения обоснованности **доказательств**. В английском языке аналог слова доказательство (argument) применяется также для описания ожесточенного "обмена мнениями", но в логике это слово имеет совершенно другой смысл. Логическое доказательство — это группа утверждений, в которой последнее рассматривается как обоснованное с использованием предыдущих в **цепи рассуждений**. Одним из типов логического доказательства является силлогизм, который рассматривался в главе 2. Пример силлогизма приведен ниже.

```
Посылка. Любой, кто умеет составлять программы,
интеллектуально развит
Посылка. Джон умеет составлять программы
Заключение. Следовательно, Джон интеллектуально развит
```

В доказательстве посылки используются в качестве свидетельств, позволяющих доказать истинность заключений. Посылки называют также **антецедентами**, а заключение — **консеквентом**. Наиболее важной характерной особенностью дедуктивной логики является то, что из истинных посылок должно следовать истинное заключение. Как показано выше, посылки принято отделять от заключения чертой, поэтому нет необходимости явно указывать, что является посылками и что заключением.

Приведенное выше доказательство можно записать более кратко следующим образом:

Любой, кто умеет составлять программы, интеллектуально развит
Джон умеет составлять программы
∴ Джон интеллектуально развит

В этом доказательстве троеточие (∴) означает "следовательно".

Перейдем к более подробному описанию силлогистической логики. Основное преимущество изучения силлогизмов состоит в том, что это — простая, хорошо изученная область логики, которая может быть полностью доказана. Кроме того, силлогизмы часто бывают полезными потому, что их можно выразить в терминах правил `IF-THEN`. Например, предыдущий силлогизм может быть перефразирован таким образом:

```
IF Любой, кто умеет составлять программы, интеллектуально
   развит и Джон умеет составлять программы
THEN Джон интеллектуально развит
```

Вообще говоря, силлогизмом считается любое действительное дедуктивное доказательство, имеющее две посылки и одно заключение. Классическим счита-

ется силлогизм специального типа, называемый **категорическим силлогизмом**. В таком силлогизме посылки и заключения определяются как категорические утверждения, имеющие одну из четырех форм, которые показаны в табл. 3.2.

Таблица 3.2. Категорические утверждения

Форма	Схема	Значение
A	Все S есть P	общее утвердительное
E	Никакие S не есть P	общее отрицательное
I	Некоторые S есть P	частное утвердительное
O	Некоторые S не есть P	частное отрицательное

Следует отметить, что в логике термин *схема* определяет логическую форму утверждения. Такое определение является также примером еще одного способа применения слова "схема", который отличается от трактовки, используемой в искусственном интеллекте (см. главу 2). В логике слово "схема" служит для обозначения наиболее существенной формы доказательства. Кроме того, схемы могут задавать логическую форму всего силлогизма, как показано ниже.

Все M есть P
Все S есть M
$\therefore$ Все S есть P

Субъект этого заключения, S, называется **младшим членом**, а предикат заключения, P, — **старшим членом**. Посылка, содержащая старший член, называется **большей посылкой**, а посылка, содержащая младший член, — **меньшей посылкой**. Если в силлогизме, например, как показано ниже, выделены бóльшая и меньшая посылки, то силлогизм рассматривается как представленный в **стандартной форме**.

Бóльшая посылка. Все M есть P
Меньшая посылка. Все S есть M
Заключение. Все S есть P

Субъект представляет собой то, о чем говорится в силлогизме, а **предикат** служит для описания некоторого свойства субъекта. Например, в следующем высказывании субъект обозначается словом "микрокомпьютеры", а предикатом является слово "компьютеры":

Все микрокомпьютеры являются компьютерами

В приведенном ниже высказывании субъектом является выражение "компьютеры с объемом оперативной памяти 1 Гбайт", а предикатом — выражение "компьютеры с большим объемом памяти".

```
Все компьютеры с объемом оперативной памяти 1 Гбайт
являются компьютерами с большим объемом памяти
```

С древних времен формы категорических утверждений обозначались буквами A, E, I и O. Буквами A и I обозначаются утвердительные высказывания; принято считать, что эти буквы происходят от первых двух гласных латинского слова affirmo (утверждаю); с другой стороны, буквы E и O взяты из слова nego (отрицаю). Формы A и I называются **утвердительными по качеству**, поскольку в них утверждается, что субъекты включены в класс, описываемый предикатом. Формы E и O являются **отрицательными по качеству**, поскольку указывают, что субъекты исключены из класса предиката.

Глагол "являться" в латинском языке обозначается словом copula (**связка**), которое обозначает связь между посылками. Связка соединяет две части высказывания. В стандартном категорическом силлогизме связка представлена глаголом "быть" в настоящем времени. Таким образом, для представления посылок может использоваться еще одна версия:

Все S являются P

Третий член силлогизма, M, называется **средним членом** и становится общим для обеих посылок. Средний член очень важен, поскольку силлогизм определен таким образом, что заключение нельзя вывести из любой отдельно взятой посылки. Поэтому следующее доказательство не представляет собой действительный силлогизм, поскольку следует только из первой посылки:

Все A есть B
Все B есть C
$\therefore$ Все A есть B

Количественный показатель, или **квантор**, указывает, какая часть класса рассматривается в посылке. Кванторы "все" и "ни один" являются **общими**, поскольку распространяются на целые классы. А квантор "некоторые" называется **частным**, поскольку он распространяется только на часть класса.

Модус силлогизма определяется тремя буквами, которые указывают соответственно форму большей посылки, меньшей посылки и заключения. Например, следующий силлогизм имеет модус AAA:

Все M есть P
Все S есть M
$\therefore$ Все S есть P

Как показано в табл. 3.3, для размещения термов S, P и M могут использоваться четыре возможных шаблона. Каждый шаблон называется фигурой, а номер **фигуры** определяет ее тип.

Таблица 3.3. Шаблоны категорических утверждений

	Фигура 1	**Фигура 2**	**Фигура 3**	**Фигура 4**
Бóльшая посылка	$M\ P$	$P\ M$	$M\ P$	$P\ M$
Меньшая посылка	$S\ M$	$S\ M$	$M\ S$	$M\ S$

Итак, приведенный выше пример можно полностью описать как силлогизм типа AAA-1. Но лишь из того, что доказательство имеет силлогистическую форму, отнюдь не следует, что оно представляет собой действительный силлогизм. Рассмотрим следующий силлогизм, представленный в форме AEE-1:

Все M есть P
Ни один S не есть M
$\therefore$ Ни один S не есть P

Как показывает следующий пример, этот силлогизм нельзя считать действительным:

Все микрокомпьютеры являются компьютерами
Ни один мэйнфрейм не является микрокомпьютером
$\therefore$ Ни один мэйнфрейм не является компьютером

Безусловно, для того чтобы проверить обоснованность силлогистических доказательств, можно попытаться воспользоваться примерами, но для этой цели проще применить **процедуру принятия решений**. *Процедурой принятия решений* называется метод доказательства обоснованности. Любая процедура принятия решений представляет собой некоторый общий механический метод или алгоритм, позволяющий автоматизировать процесс определения обоснованности. Такие процедуры принятия решений существуют для силлогистической логики и пропозициональной логики, но, как показал Чёрч (Church) в 1936 году, для логики предикатов такая процедура не может быть предусмотрена. Вместо этого для выработки доказательств люди или компьютеры должны применять определенную изобретательность. В 1970-х годах с помощью таких программ, как Automated Mathematician и Eurisko Дуга Лената, удалось повторно найти математические доказательства для гипотезы Гольдбаха и теоремы уникального разложения на множители. Но редакторы математических журналов не очень охотно публиковали творческие работы, подготовленные с помощью компьютеров, так же как редакторов литературных газет и журналов мало привлекала перспектива публикации поэм или романов, которые сочинялись с помощью компьютеров (однако от стремления заплатить за эти сочинения немного меньше они не отказывались).

Процедура принятия решений для высказываний состоит в построении истинностной таблицы и проверке ее на наличие тавтологии. А процедура принятия

решений в отношении силлогизмов может быть осуществлена с использованием диаграмм Венна с тремя перекрывающимися кружками, представляющими члены S, P и M (рис. 3.15, *а*). Бóльшая посылка для следующего силлогизма в форме AEE-1 показана на рис. 3.15, *б*:

Все M есть P
Ни один S не есть M
$\therefore$ Ни один S не есть P

Заштрихованная часть кружка M показывает, что в этой части нет элементов. На рис. 3.15, *в* меньшая посылка учтена путем штриховки ее части, не имеющей элементов. На основании рис. 3.15, *в* можно определить, что заключение силлогизма в форме AEE-1 является ложным, поскольку в P имеются некоторые элементы S.

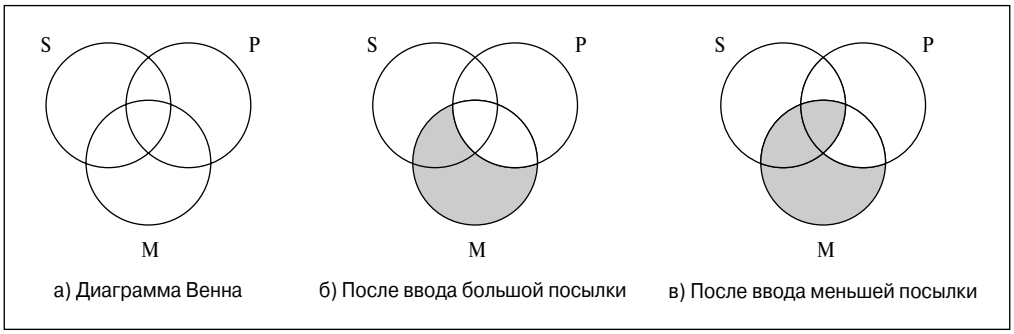


Рис. 3.15. Процедура принятия решений для силлогизма в форме AEE-1

В качестве еще одного примера укажем, что следующий силлогизм в форме EAE-1 является действительным, как показано на рис. 3.16, *в*:

Ни один M не есть P
Все S есть M
$\therefore$ Ни один S не есть P

Задача построения диаграмм Венна для силлогизмов, включающих кванторы “некоторые”, становится немного более сложной. Ниже приведены общие правила построения диаграмм для категорических силлогизмов в соответствии с булевым представлением, согласно которому в множествах, соответствующих высказываниям A и E, может не быть элементов.

1. Если класс пуст, соответствующий ему кружок заштриховывается.
2. Общие утверждения, A и E, всегда представляются на диаграмме перед частными.

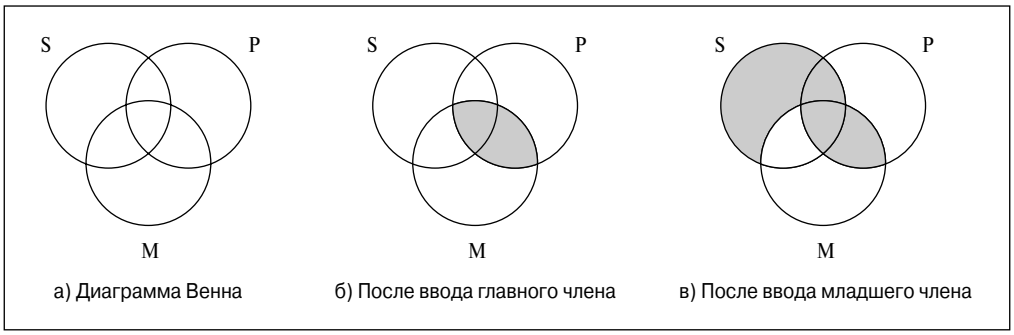


Рис. 3.16. Процедура принятия решений для силлогизма в форме EAE-1

3. Если класс включает по меньшей мере один элемент, он обозначается звездочкой (*).
4. Если некоторое высказывание не позволяет определить, в каком из двух смежных классов существует объект, то символ * изображается на линии, разделяющей классы.
5. Если некоторая область заштрихована, то в нее нельзя помещать символ *.

В качестве примера рассмотрим следующий силлогизм:

Некоторые компьютеры являются лэптопами
Все лэптопы являются портативными устройствами
$\therefore$ Некоторые портативные устройства являются компьютерами

Этот силлогизм может быть преобразован в форму IAI-4 таким образом:

Некоторые P есть M
Все M есть S
$\therefore$ Некоторые S есть P

В соответствии с правилами 2 и 1 построения диаграмм Венна начнем с общего высказывания, относящегося к меньшей посылке, и заштрихуем соответствующий кружок, как показано на рис. 3.17, *а*. Затем применим правило 3 к частной большой посылке и поставим символ *, как показано на рис. 3.17, *б*. Поскольку на диаграмме показано заключение “Некоторые портативные устройства являются компьютерами”, из этого следует, что доказательство в форме IAI-4 представляет собой действительный силлогизм.

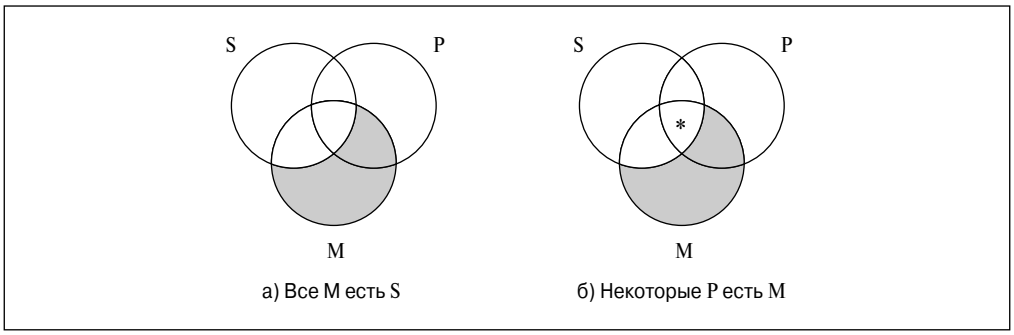


Рис. 3.17. Силлогизм в форме IAI-4

3.6 Правила вывода

Безусловно, диаграммы Венна могут использоваться в качестве процедуры принятия решений для силлогизмов, но при наличии более сложных доказательств диаграммы становятся неудобными, поскольку их чтение затрудняется. Но с силлогизмами связана более фундаментальная проблема, поскольку с их помощью может быть представлена лишь небольшая часть возможных логических высказываний. В частности, категорические силлогизмы позволяют рассматривать только категорические высказывания в форме A, E, I и O.

Другие способы описания доказательств предоставляет пропозициональная логика. В действительности люди, не осознавая этого, часто используют пропозициональную логику. Например, рассмотрим следующее пропозициональное доказательство:

Если на компьютер подается питание, то компьютер будет работать
На компьютер подается питание
$\therefore$ Компьютер будет работать

Это доказательство может быть выражено с помощью формального способа, если для обозначения высказываний будут использоваться буквы, следующим образом:

A — На компьютер подается питание
B — Компьютер будет работать

Поэтому приведенное выше доказательство может быть записано таким образом:

$$\begin{array}{l} A \rightarrow B \\ \underline{A} \\ \therefore B \end{array}$$

Доказательства, подобные этому, встречаются очень часто. Общая схема представления доказательств указанного типа является следующей:

$$\begin{array}{l} p \rightarrow q \\ \underline{p} \\ \therefore q \end{array}$$

В этой схеме p и q — логические переменные, которые могут представлять любые высказывания. В пропозициональной логике предусмотрена возможность использовать логические переменные, а это позволяет применять больше сложные типы высказываний по сравнению с четырьмя силлогистическими формами A, E, I и O. Схема логического вывода для этой пропозициональной формы известна под разными названиями: **непосредственное формирование рассуждений**, **правило модус поненс**, **закон отделения** и **принятие антецедента** [65].

Обратите внимание на, что данный пример можно также выразить в следующей силлогистической форме:

Все компьютеры, на которые подается питание, будут работать
На этот компьютер подается питание
∴ Этот компьютер будет работать

Этот пример показывает, что правило модус поненс в действительности представляет собой частный случай силлогистической логики. Но важность правила модус поненс обусловлена тем, что этот способ логического вывода образует фундамент экспертных систем, основанных на правилах.

Но системы, основанные на правилах, не полагаются исключительно на логику, поскольку и люди при решении задач используют более широкие средства по сравнению с логикой. В реальном мире может потребоваться применить несколько конкурирующих правил, а не какое-либо единственное правило силлогизма. Машина применения правил экспертной системы может решить, является ли данное конкретное правило пригодным для выполнения, так же, как может решительно поступить и человек, отвечая на свой собственный вопрос: “Должен ли я съесть эту последнюю конфету и удовлетворить свое страстное желание или не есть ее и оставаться стройным?”

Недостаточно просто написать на языке C целый ряд правил и получить в свое распоряжение всю мощь какого-то инструментального средства экспертной системы, хотя в случае очень простых приложений может оказаться, что в этом как раз и состоит все, что требуется (`http://www.ddj.com/documents/s=9064/ddj0301aie001/0301aie001.htm`). Истинная мощь инструментального средства экспертной системы проявляется после того, как в ней начинают присутствовать сотни или тысячи правил, и эти правила вступают в конфликт.

Rete-алгоритм сопоставления с шаблонами является одним из наиболее мощных и вместе с тем эффективных методов разрешения конфликтов между правилами и представляет собой основу методов сопоставления с шаблонами языка CLIPS (`http://www.ddj.com/documents/s=9064/ddj0212ai002/0212aie002.htm`).

Составное высказывание $p \rightarrow q$ соответствует правилу, а высказывание p соответствует шаблону, который должен быть согласован с антецедентом правила, для того чтобы это правило было выполнено. Но, как описано в главе 2, условное выражение $p \rightarrow q$ не является точно эквивалентным правилу, поскольку условное выражение — это логическое определение, задаваемое с помощью истинностной таблицы, поэтому может быть предусмотрено много разных возможных определений условного выражения.

Вообще говоря, в данной книге используются предусмотренное в математической логике соглашение, в соответствии с которым для обозначения постоянных высказываний, таких как "подается питание", служат прописные буквы, например A, B, C, ... Строчные буквы, такие как p, q, r, ..., представляют собой логические переменные, которые могут обозначать различные постоянные высказывания. Следует отметить, что это соглашение противоположно применяемому в языке PROLOG, поскольку в этом языке для обозначения переменных используются прописные буквы.

Приведенная выше схема правила модус поненс может быть записана с применением логических переменных, имеющих другие имена, как в следующем примере, но будет по-прежнему означать то же самое:

$$\begin{array}{l} r \rightarrow s \\ r \\ \hline \therefore s \end{array}$$

Для этой схемы применяется также другая система обозначений:

$$r, r \rightarrow s; \therefore s$$

В этой системе для отделения одной посылки от другой используется запятая, а точка с запятой обозначает конец посылок. До сих пор в данной главе рассматривались доказательства только с двумя посылками, но может применяться более общая форма доказательства, которая показана ниже.

$$P_1, P_2, \ldots, P_N; \therefore C$$

В этой форме прописные буквы P_i обозначают посылки, такие как r и $r \rightarrow s$, а C обозначает заключение. Обратите внимание на то, что такое доказательство

напоминает по своей форме оператор выполнения цели языка PROLOG, который рассматривался в главе 2:

$$p :- p_1, p_2, \ldots, p_N.$$

Цель p выполняется, если выполнены все подцели, p_1, p_2, ..., p_N. Аналогичное доказательство для продукционных правил может быть записано в следующей общей форме:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_N \rightarrow A$$

Эта форма означает, что если выполнено каждое условие правила, C_i, то осуществляется действие правила, A. Как было описано выше, логическое утверждение, представленное в указанной форме, является строго не эквивалентным правилу, поскольку логическое определение условного выражения не совпадает с определением продукционного правила. Однако эта логическая форма является полезным интуитивным вспомогательным средством, способствующим пониманию сути правил.

В языке PROLOG для обозначения логических операций AND и OR применяются другие символы, отличные от обычных символов $\wedge$ и $\vee$. В операторах PROLOG запятая между подцелями обозначает конъюнкцию, $\wedge$, а с помощью точки с запятой обозначается дизъюнкция, $\vee$. Например, следующий оператор указывает, что цель p выполняется, если выполнена одна из целей, p_1 или p_2:

$$p\text{:-}\ p_1; p_2.$$

Знаки операций конъюнкции и дизъюнкции могут чередоваться. Например, следующий оператор PROLOG:

$$p\text{:-}\ p_1, p_2; p_3, p_4.$$

равносилен двум приведенным ниже операторам PROLOG.

$$p\text{:-}\ p_1, p_2.$$
$$p\text{:-}\ p_3, p_4.$$

Безусловно, в языке PROLOG реализована мощная стратегия формирования рассуждений, но правила, созданные указанным выше способом, не всегда подходят для применения по произвольному назначению. Тем не менее при использовании языка PROLOG существует возможность более точно определить способ выполнения правил, чтобы можно было лучше приспособить форму представления знаний и стратегию логического вывода с учетом потребностей приложения. Примером такой организации работы может служить приложение, предназначенное для автоматизации офисов врачей-педиатров (`www.visualdatallc.com`), которое более подробно описано в [73].

Вообще говоря, если и посылки, и заключение полностью представлены в виде схем, то доказательство

$$P_1, P_2, \ldots P_N; \therefore C$$

является формально правильным дедуктивным доказательством тогда и только тогда, когда следующее выражение представляет собой тавтологию:

$$P_1 \wedge P_2 \wedge \ldots P_N \rightarrow C$$

В качестве примера можно указать, что приведенное ниже выражение представляет собой тавтологию, поскольку является истинным при любых значениях p и q, T (истина) или F (ложь).

$$(p \wedge q) \rightarrow p$$

Для проверки сказанного читатель может составить истинностную таблицу.

Следующее доказательство, формируемое по правилу модус поненс:

$$\begin{array}{l} p \rightarrow q \\ \underline{p \qquad\quad} \\ \therefore q \end{array}$$

является действительным, поскольку может быть представлено в виде тавтологии:

$$(p \rightarrow q) \wedge p \rightarrow q$$

Следует отметить, что в настоящем изложении предполагается наличие у знака операции “стрелка” более низкого приоритета, чем у знаков операций конъюнкции и дизъюнкции. Такое соглашение позволяет обойтись без применения дополнительных круглых скобок, подобных показанным ниже.

$$((p \rightarrow q) \wedge p) \rightarrow q$$

Истинностная таблица для правила модус поненс показана в табл. 3.4. Это правило представляет собой тавтологию, поскольку все значения выражения, соответствующего доказательству, показанные в крайнем правом столбце, являются истинными, независимо от того, какие значения имеют посылки доказательства. Обратите внимание на то, что в третьем, четвертом и пятом столбцах истинностные значения записаны под обозначениями определенных знаков операций, таких как $\rightarrow$ и $\wedge$. Эти знаки операций называются основными связками, поскольку соединяют две основные части составного высказывания.

Безусловно, описанный выше метод определения действительных доказательств является вполне применимым, но требует проверки каждой строки истинностной таблицы. Количество таких строк равно 2^N, где N — количество посылок.

Таблица 3.4. Истинностная таблица для правила модус поненс

p	q	$p \to q$	$(p \to q) \wedge p$	$(p \to q) \wedge p \to q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Поэтому с увеличением количества посылок количество строк возрастает очень быстро. Например, при использовании пяти посылок потребовалось бы проверить 32 строки, а при десяти посылках — 1024 строки. Более краткий метод определения того, является ли доказательство действительным, состоит в рассмотрении только тех строк истинностной таблицы, в которых все посылки являются истинными. В эквивалентном определении действительного доказательства указывается, что доказательство является действительным тогда и только тогда, когда для каждой из таких строк заключение является истинным. Это означает, что заключение тавтологически следует из посылок. В правиле модус поненс посылка $p \to q$ и посылка p одновременно являются истинными только в первой строке, таким же является и заключение. Поэтому правило модус поненс представляет собой действительное доказательство. А если бы в истинностной таблице находилась любая другая строка, в которой все посылки были бы истинными, а заключение — ложным, то такое доказательство было бы недействительным.

Более краткий способ выражения истинностной таблицы для правила модус поненс показан в табл. 3.5, где явно представлены все строки. Но на практике необходимо учитывать только те строки, в которых имеются истинные посылки, такие как первая строка.

Таблица 3.5. Вариант представления истинностной таблицы для правила модус поненс в более краткой форме

		Посылки		**Заключение**
p	q	$p \to q$	p	q
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	F	F

Истинностная таблица для правила модус поненс показывает, что это правило может служить действительным доказательством, поскольку в первой строке имеются истинные посылки и истинное заключение, а другие строки, в которых были бы истинные посылки и ложное заключение, отсутствуют.

Тем не менее некоторые доказательства могут вводить в заблуждение. Чтобы убедиться в этом, вначале рассмотрим следующий пример действительного доказательства по правилу модус поненс:

Если в программе нет ошибок, то программа успешно проходит этап компиляции
В программе нет ошибок

$\therefore$ Программа успешно проходит этап компиляции

Сравните его со следующим доказательством, которое немного напоминает доказательство по правилу модус поненс:

Если в программе нет ошибок, то программа успешно проходит этап компиляции
Программа успешно проходит этап компиляции

$\therefore$ В программе нет ошибок

Является ли это доказательство действительным? Схема доказательства рассматриваемого типа является таковой:

$$\begin{array}{l} p \rightarrow q \\ q \\ \hline \therefore p \end{array}$$

а его истинностная таблица в краткой форме показана в табл. 3.6.

Таблица 3.6. Истинностная таблица доказательства $p \rightarrow q, q; \therefore p$ в краткой форме

		Посылки		**Заключение**
p	q	$p \rightarrow q$	q	p
T	T	T	T	T
T	F	F	F	T
F	T	T	T	F
F	F	T	F	F

Обратите внимание на то, что это доказательство не является действительным. Безусловно, первая строка показывает, что заключение истинно, если все посылки являются истинными, но третья строка указывает на то, что при всех истинных посылках заключение становится ложным. Таким образом, данное доказательство не соответствует приведенному выше обязательному критерию определения действительного доказательства. Хотя многие программисты хотели бы, чтобы подобные доказательства были истинными, логика (и опыт) показывают, что это

доказательство — ошибочно, или недействительно. Данное конкретное ошибочное доказательство называется **ошибочным обращением посылки и заключения** (fallacy of the converse). Понятие ошибочного обращения определено в табл. 3.9.

В качестве еще одного примера укажем, что следующая схема доказательства является действительной, поскольку, как видно в табл. 3.7, заключение является истинным тогда и только тогда, когда все посылки истинны:

$$\begin{array}{l} p \rightarrow q \\ \sim q \\ \hline \therefore \sim p \end{array}$$

Таблица 3.7. Истинностная таблица для доказательства $p \rightarrow q, \sim q; \therefore \sim p$ в краткой форме

		Посылки		Заключение
p	q	$p \rightarrow q$	$\sim q$	$\sim p$
T	T	T	F	F
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

Данная конкретная схема доказательства известна под разными названиями: **косвенное рассуждение**, **правило модус толленс** и **закон контрапозиции**.

Правила модус поненс и модус толленс представляют собой правила логического вывода, иногда называемые **законами логического вывода**. Некоторые из законов логического вывода показаны в табл. 3.8.

Латинское слово modus (модус) означает “способ”, слово ponere означает “утверждать”, а слово tollere — “отрицать”. Настоящие названия этих правил модуса и буквальные значения показаны в табл. 3.9. Правила первых двух типов сокращенно именуются как модус поненс и модус толленс [89]. Номера правил вывода, приведенные в этой таблице, соответствуют номерам, которые были указаны в табл. 3.8.

Правила логического вывода могут применяться к доказательствам, в которых участвуют больше чем две посылки. Например, рассмотрим следующее доказательство:

```
Цены на микросхемы растут, только если растет курс иены.
Курс иены растет, только если курс доллара падает и
   если курс доллара падает, то курс иены растет.
Поскольку цены на микросхемы выросли,
   курс доллара должен был упасть.
```

Таблица 3.8. Некоторые правила вывода для пропозициональной логики

Закон логического вывода	Схемы	
1. Закон отделения (правило модус поненс)	$p \to q$ p $\therefore q$	
2. Закон контрапозиции	$p \to q$ $\therefore \sim q \to \sim p$	
3. Правило модус толленс	$p \to q$ $\sim q$ $\therefore \sim p$	
4. Цепное правило (закон силлогизма)	$p \to q$ $q \to r$ $\therefore p \to r$	
5. Закон дизъюнктивного логического вывода	$p \vee q$ $\sim p$ $\therefore q$	$p \vee q$ $\sim q$ $\therefore p$
6. Закон двойного отрицания	$\sim(\sim p)$ $\therefore p$	
7. Закон де Моргана	$\sim(p \wedge q)$ $\therefore \sim p \vee \sim q$	$\sim(p \vee q)$ $\therefore \sim p \wedge \sim q$
8. Закон упрощения	$p \wedge q$ $\therefore p$	$\sim(p \vee q)$ $\therefore q$
9. Закон конъюнкции	p q $\therefore p \wedge q$	
10. Закон дизъюнктивного добавления	p $\therefore p \vee q$	
11. Закон конъюнктивного доказательства	$\sim(p \wedge q)$ p $\therefore \sim q$	$\sim(p \wedge q)$ q $\therefore \sim p$

Таблица 3.9. Буквальные значения различных модусов (правил, или законов)

Номер правила вывода	Латинское название	Смысл, выраженный как “модус, который. . . ”
1	modus ponendo ponens	утверждая, утверждает
3	modus tollendo tollens	отрицая, отрицает
5	modus tollendo ponens	отрицая, утверждает
11	modus ponendo tollens	утверждая, отрицает

Предположим, что высказывания определены следующим образом:

C — цены на микросхемы растут
Y — курс иены растет
D — курс доллара падает

Как было указано в разделе 2.12, одним из значений условного выражения является "p, только если q". Такое высказывание, как "курс иены растет, только если курс доллара падает", имеет именно этот смысл, поэтому может быть представлено как $D \rightarrow Y$. Все доказательство имеет следующую форму:

$$C \rightarrow Y$$
$$(Y \rightarrow D) \wedge (D \rightarrow Y)$$
$$\underline{C \quad\quad}$$
$$\therefore D$$

Вторая посылка имеет очень интересную форму, поскольку может быть еще больше сокращена с использованием одного из вариантов условного выражения. Условное выражение $p \rightarrow q$ имеет несколько вариантов, которые представляют собой обратную, **противоположную** и **контрапозитивную** форму. Эти формы для полноты перечислены вместе с условным выражением в табл. 3.10.

Таблица 3.10. Условное выражение и его варианты

Название	**Запись**
Условное выражение	$p \rightarrow q$
Обратная форма	$q \rightarrow p$
Противоположная форма	$\sim p \rightarrow \sim q$
Контрапозитивная форма	$\sim q \rightarrow \sim p$

Как обычно, предполагается, что операция отрицания имеет более высокий приоритет по сравнению с другими логическими операциями, поэтому выражения $\sim p$ и $\sim q$ никогда не заключаются в круглые скобки.

Если условное выражение $p \rightarrow q$ и обратное ему выражение $q \rightarrow p$ одновременно являются истинными, то выражения p и q — эквивалентны. Это означает, что выражение $p \rightarrow q \wedge q \rightarrow p$ эквивалентно **двустороннему условному выражению** $p \leftrightarrow q$, или **эквивалентности** $p \equiv q$. Обратите внимание на то, что обычные знаки операций присваивания или проверки на равенство записываются с двумя короткими горизонтальными чертами ($=$), а знак операции эквивалентности записывается с тремя чертами ($\equiv$). Иными словами, выражения p и q, связанные

символом эквивалентности, всегда принимают одни и те же истинностные значения. Если p имеет значение T, то q истинно, а если p имеет значение F, то и q равно F. Приведенное выше доказательство принимает следующую форму:

(1) $C \rightarrow Y$
(2) $Y \equiv D$
(3) $\underline{C}$
$\therefore D$

В данном случае числа в круглых скобках используются для обозначения посылок. Поскольку, согласно посылке (2), выражения Y и D эквивалентны, можно подставить D вместо Y в посылку (1) и получить следующее выражение:

(4) $C \rightarrow D$

В данном случае выражение (4) — это результат логического вывода, сделанного на основе выражений (1) и (2). Посылки (3) и (4), а также заключение применяют такой вид:

(4) $C \rightarrow D$
(3) $\underline{C}$
$\therefore D$

Это доказательство распознается как имеющее схему правила модус поненс, поэтому оно является действительным.

Правило, согласно которому допускается подстановка одной переменной вместо другой, по отношению к которой первая является эквивалентной, представляет собой правило логического вывода, называемое **правилом подстановки** (rule of substitution). Двумя основными правилами дедуктивной логики являются правило модус поненс и правило подстановки.

Доказательство в формальной логике обычно записывается с применением нумерации посылок, заключения и этапов логического вывода, как в приведенным ниже примере.

1.	$C \rightarrow Y$	
2.	$(Y \rightarrow D) \wedge (D \rightarrow Y)$	
3.	C	$/ \therefore D$
4.	$Y \equiv D$	2 Эквивалентность
5.	$C \rightarrow D$	1 Подстановка
6.	D	$3, 5$ Правило модус поненс

В строках 1–3 показаны посылки и заключение, а в строках 4–6 — выполненные этапы логического вывода. В правом столбце указаны правила логического вывода и номера строк, используемых для обоснования этапов логического вывода.

3.7 Ограничения пропозициональной логики

Еще раз рассмотрим используемое в данной книге классическое доказательство:

Все люди смертны
Сократ — человек

Следовательно, Сократ смертен

Известно, что это — действительное доказательство, поскольку оно представляет собой действительный силлогизм. Сможем ли мы доказать его действительность с применением пропозициональной логики? Чтобы ответить на этот вопрос, вначале запишем доказательство в виде схемы, обозначив высказывание буквами следующим образом:

p — Все люди смертны
q — Сократ — человек
r — Сократ смертен

Таким образом, рассматриваемое доказательство имеет следующую схему:

$$\begin{array}{l} p \\ q \\ \hline \therefore r \end{array}$$

Обратите внимание на то, что в посылках и заключении нет логических связок, поэтому для каждой посылки и заключения должна применяться отдельная логическая переменная. Кроме того, в пропозициональной логике не предусмотрено применение кванторов, и поэтому отсутствует способ представления квантора “все” в первой посылке. Таким образом, единственным способом представления данного доказательства в пропозициональной логике является приведенная выше схема, состоящая из трех независимых переменных.

Для определения того, является ли это доказательство действительным, рассмотрим истинностную таблицу для трех независимых переменных, в которой используются все возможные комбинации значений T и F (табл. 3.11). Вторая строка этой истинностной таблицы показывает, что данное доказательство является недействительным, поскольку посылки истинны, а заключение — ложно.

Недействительность этого доказательства не следует интерпретировать в том смысле, что его заключение является неправильным. Любой разумный человек

Таблица 3.11. Истинностная таблица для схемы $p, q; \therefore r$

p	q	$\therefore r$
T	T	T
T	T	F
T	F	T
T	F	F
F	T	T
F	T	F
F	F	T
F	F	F

признает, что это — правильное доказательство, а то, что проверка по таблице показывает его недействительность, просто означает, что это доказательство не может быть обосновано в рамках пропозициональной логики. То, что данное доказательство является действительным, можно было бы обосновать, исследуя внутреннюю структуру посылок. А для этого пришлось бы, например, приписать определенный смысл квантору "все" и учесть, что слово "люди" — множественное число от слова "человек". Но такие области логики, как силлогизмы и пропозициональное исчисление, не позволяют исследовать внутреннюю структуру высказываний. Эти ограничения преодолены в логике предикатов, и поэтому рассматриваемое доказательство становится действительным в логике предикатов. Вся силлогистическая логика фактически является действительным подмножеством логики предикатов первого порядка, и можно показать, что она является действительной именно в рамках этой логики.

Единственная действительная силлогистическая форма рассматриваемого выше высказывания является таковой:

Если Сократ — человек, то Сократ смертен
Сократ — человек

Следовательно, Сократ смертен

Обозначим отдельные выражения этого высказывания следующим образом:

p — Сократ — человек
q — Сократ смертен

В таком случае доказательство принимает вид

$$\frac{\begin{array}{l} p \rightarrow q \\ p \end{array}}{\therefore q}$$

А это — действительная силлогистическая форма правила модус поненс.

В качестве еще одного примера рассмотрим следующее классическое доказательство:

$$\frac{\text{Все лошади — животные}}{\text{Следовательно, голова лошади — голова животного}}$$

Известно, что это — правильное доказательство, но оно все же не может быть доказано в рамках пропозициональной логики, хотя его можно доказать с помощью логики предикатов (см. задачу 3.12).

3.8 Логика предикатов первого порядка

Силлогистическая логика может быть полностью описана с помощью логики предикатов. В табл. 3.12 показаны четыре категорических утверждения и их представления в логике предикатов.

Таблица 3.12. Представление четырех категорических силлогизмов с помощью логики предикатов

Тип	Схема	Представление предиката
A	Все S есть P	$(\forall x)(S(x) \rightarrow P(x))$
E	Ни один S не есть P	$(\forall x)(S(x) \rightarrow \sim P(x))$
I	Некоторые S есть P	$(\exists x)(S(x) \rightarrow P(x))$
O	Некоторые S не есть P	$(\exists x)(S(x) \rightarrow \sim P(x))$

В логике предикатов в дополнение к правилам вывода, описанным выше, предусмотрены правила, относящиеся к кванторам.

В частности, в правиле универсальной конкретизации (rule of universal instantiation) по существу утверждается, что вместо универсального объекта может быть подставлен какой-то конкретный объект. Например, если ϕ — это некоторое высказывание или **пропозициональная функция**, то приведенное ниже доказательство, в котором a представляет собой экземпляр объекта, является действительным логическим выводом.

$$\frac{(\forall x)\phi(x)}{\therefore \phi(a)}$$

Таким образом, переменная a ссылается на конкретный отдельный объект, а переменная x пробегает по всем отдельным объектам. Например, следующий логический вывод, в котором обозначена как $H(x)$ пропозициональная функция, указывающая на то, что x — человек, позволяет доказать, что Сократ — человек:

$$\frac{(\forall x)H(x)}{\therefore H(\text{Socrates})}$$

Фактически выше сказано, что для каждого x, x — человек, и поэтому, согласно правилу логического вывода, Сократ — человек.

Ниже приведены другие примеры применения правила универсальной конкретизации.

$$\frac{(\forall x)A(x)}{\therefore A(c)}$$

$$\frac{(\forall y)(B(y) \vee C(b))}{\therefore B(a) \vee C(b)}$$

$$\frac{(\forall x)[A(x) \wedge (\exists x)(B(x) \vee C(y))]}{\therefore A(b) \wedge (\exists x)(B(x) \vee C(y))}$$

В первом примере вместо переменной x подставляется экземпляр c. Во втором примере заслуживает внимания то, что экземпляр a подставляется вместо переменной y, но не вместо переменной b, поскольку b не включена в **область определения** квантора. Это означает, что такой квантор, как $\forall x$, применяется только к вхождениям переменной x. Такие переменные, как x и y, используемые вместе с кванторами, называются **связанными**, а другие переменные называются **свободными**. В третьем примере в область определения квантифицированной переменной x входит только функция $A(x)$. Это означает, что область определения квантора всеобщности $\forall x$ не распространяется на квантор существования $\exists x$ и на область определения этого квантора в составе выражения $B(x) \vee C(y)$. Соглашение по использованию выражений с вложенными кванторами, подобных приведенным выше, состоит в том, что область определения первого квантора оканчивается с того места, где вступает в действие новый квантор, если в следующем кванторе применяется такая же переменная, как и в предыдущем (в рассматриваемом случае это — переменная x). Формальное доказательство следующего силлогизма:

$$\frac{\begin{array}{c}\text{Все люди смертны}\\ \text{Сократ — человек}\end{array}}{\therefore \text{Сократ смертен}}$$

в котором используется обозначения H — человек, M — смертен и s — Сократ, приведено ниже.

1.	$(\forall x)(H(x) \rightarrow M(x))$	
2.	$H(s)$	$/ \therefore M(s)$
3.	$H(s) \rightarrow M(s)$	1 Универсальная конкретизация
4.	$M(s)$	$2, 3$ Правило модус поненс

3.9 Логические системы

Логической системой называется коллекция таких объектов, как правила, аксиомы, утверждения и т.д., организованная в единообразной форме. Логические системы создаются для достижения нескольких целей.

Первая цель состоит в определении форм доказательств. С точки зрения семантики логические доказательства являются бессмысленными, поэтому для определения того, является ли доказательство обоснованным, чрезвычайно важно определить действительную форму доказательства. Таким образом, одним из наиболее важных назначений любой логической системы является определение **правильно построенных формул (well-formed formula — wff)**, используемых в доказательствах. В логических доказательствах разрешается применять только правильно построенные формулы. Например, в силлогистической логике выражение

Все S есть P

может рассматриваться как правильно построенная формула, а следующие выражения не являются правильно построенной формулой:

Все

Все есть S P

Есть S все

Безусловно, символы алфавита лишены смысла, но правильно построенная формула приобретает смысл благодаря применению строго определенной последовательности символов.

Вторая цель создания логической системы состоит в том, чтобы указать, какие правила вывода являются действительными. А третьей целью логической системы является обеспечение возможности расширения самой этой системы путем открытия новых правил вывода и тем самым увеличения разновидностей форм доказательств, которые могут быть обоснованы в этой системе. В результате расширения перечня разновидностей форм доказательств появляется возможность

обосновывать с помощью логического доказательства все новые и новые правильно построенные формулы, называемые **теоремами**.

Если логическая система хорошо разработана, то ее можно использовать для определения обоснованности любых доказательств с помощью способа, аналогичного вычислениям в таких научных системах, как арифметика, геометрия, исчисление, физика и инженерное дело. К настоящему времени были разработаны такие логические системы, как сентенциальное, или пропозициональное исчисление, исчисление предикатов и т.д. Каждая из этих систем основывается на формальных определениях ее **аксиом**, или **постулатов**, которые являются фундаментальными определениями данной системы. На основании этих аксиом люди (а иногда и компьютерные программы, такие как Automated Mathematician) пытаются определить, какое доказательство может быть обосновано. Любой, кто изучал евклидову геометрию в средней школе, знаком с аксиомами и с тем, как происходит вывод геометрических теорем. Так же как геометрические теоремы могут быть выведены из геометрических аксиом, логические теоремы могут быть выведены из логических аксиом.

Аксиома — это просто факт, или **утверждение**, которое невозможно доказать в рамках самой системы. Иногда мы принимаем некоторые аксиомы за истину, поскольку им можно придать "смысл", обращаясь к здравому смыслу или к наблюдениям. А другие аксиомы, такие как "параллельные линии в бесконечности пересекаются", не имеют интуитивного смысла, поскольку противоречат привычной аксиоме Евклида, согласно которой параллельные линии не пересекаются. Но указанная аксиома, согласно которой параллельные линии в бесконечности пересекаются, с чисто логической точки зрения является не менее приемлемой, чем аксиома Евклида, и действительно лежит в основе одного из типов неевклидовой геометрии.

Для определения формальной системы необходимо следующее.

1. Алфавит символов.
2. Множество конечных строк, состоящих из этих символов, — правильно построенных формул.
3. Аксиомы — определения системы.
4. Правила вывода, позволяющие вывести правильно построенную формулу A как заключение конечного множества G других правильно построенных формул, где $G = \{A_1, A_2 \ldots A_n\}$. Эти правильно построенные формулы должны представлять собой аксиомы или другие теоремы логической системы. Например, система пропозициональной логики может быть определена на основе использования только правила модус поненс для вывода новых теорем.

Если приведенное ниже доказательство является действительным, то A называется **теоремой** данной формальной логической системы, что обозначается

символом $\vdash$.

$$A_1, A_2, ...A_N; \therefore A$$

Например, $\Gamma \vdash A$ означает, что A — теорема, которая следует из множества правильно построенных формул Γ. Более явно определенная схема доказательства того, что A — теорема, является таковой:

$$A_1, A_2, ...A_N \vdash A$$

Символ $\vdash$, который показывает, что следующая правильно построенная формула представляет собой теорему, не является символом самой системы. Вместо этого символ $\vdash$ рассматривается как **метасимвол**, поскольку он используется для описания самой системы. В качестве аналогии можно рассмотреть такой язык программирования, как Java. Хотя программы на этом языке можно определять с применением синтаксических конструкций Java, в языке Java отсутствуют синтаксические конструкции, применимые для указания на то, что некоторая программа является действительной.

Любое правило вывода формальной системы точно указывает, как могут быть получены новые утверждения (теоремы) из аксиом и ранее выведенных теорем. Примером теоремы может служить уже рассматривавшийся силлогизм о Сократе, записанный в форме логики предикатов:

$$(\forall x)(H(x) \rightarrow M(x)), H(s) \vdash M(s)$$

В этом выражении H — предикативная функция, которая определяет понятие человека, а M — предикативная функция, определяющая понятие смертного. Поскольку выражение $M(s)$ можно доказать на основании аксиом, приведенных слева от этого выражения, оно представляет собой теорему, основанную на этих аксиомах. Но следует отметить, что выражение $M(\text{Zeus})$ не может быть теоремой, поскольку Зевс (греческий бог) — не человек и не предусмотрен альтернативный способ показать истинность выражения $M(\text{Zeus})$.

Если теорема представляет собой тавтологию, из этого следует, что Γ — множество меры нуль, поскольку правильно построенная формула всегда истинна и не зависит от любых других аксиом или теорем. Теорема, которая является тавтологией, обозначается символом $\models$, как в следующем примере:

$$\models A$$

Например, если $A \equiv p \vee \sim p$, то в следующем выражении утверждается, что $p \vee \sim p$ — теорема, а сама эта теорема представляет собой тавтологию:

$$\models p \vee \sim p$$

Обратите внимание на то, что, вне зависимости от значений, присваиваемых переменной p, будь то T или F, теорема $p \vee \sim p$ всегда является истинной. **Интерпретацией** правильно построенной формулы называется определенный вариант присваивания истинностных значений переменным формулы, а **моделью** называется интерпретация, в которой правильно построенная формула является истинной. Например, моделью правильно построенной формулы $p \vee q$ может служить $p = T$ и $q = T$. Правильно построенная формула называется **совместимой**, или **выполнимой**, если существует интерпретация, в которой эта формула принимает истинное значение. А правильно построенная формула, ложная во всех интерпретациях, называется **несовместимой**, или **невыполнимой**. В качестве примера несовместимой правильно построенной формулы можно указать $p \wedge \sim p$.

Правильно построенная формула является **действительной**, если она истинна во всех интерпретациях, а в противном случае она рассматривается как **недействительная**. Например, правильно построенная формула $p \vee \sim p$ является действительной, а $p \rightarrow q$ может служить примером недействительной правильно построенной формулы, поскольку не имеет истинного значения при $p = T$ и $q = F$. Правильно построенная формула считается **доказанной**, если можно показать, что она действительна. Все пропозициональные правильно построенные формулы могут быть доказаны с помощью метода, основанного на использовании истинностной таблицы, поскольку в пропозициональном исчислении каждая правильно построенная формула имеет лишь конечное количество интерпретаций. Это означает, что пропозициональное исчисление является **разрешимым**. С другой стороны, исчисление предикатов не является разрешимым, поскольку для него не существует общего метода доказательства для всех правильно построенных формул исчисления предикатов, подобного методу пропозиционального исчисления, основанному на истинностных таблицах.

В качестве одного из примеров действительной правильно построенной формулы исчисления предикатов можно указать следующую формулу, истинную для любого предиката B:

$$(\exists x)B(x) \rightarrow \sim[(\forall x) \sim B(x)]$$

Эта формула показывает, как можно заменить квантор существования квантором всеобщности. Поэтому данная правильно построенная формула исчисления предикатов представляет собой теорему.

Между такими выражениями, как $\vdash A$ и $\models B$, имеется значительное различие. Первое выражение означает, что A — теорема и поэтому может быть доказана на основании аксиом с помощью правил вывода. А во втором выражение утверждается, что B — правильно построенная формула, но может быть неизвестным доказательство, с помощью которого мог бы быть продемонстрирован вывод этой формулы. Пропозициональная логика является разрешимой, а логика предикатов не является таковой. Это означает, что не существует механической

процедуры, или алгоритма поиска доказательства теоремы логики предикатов за конечное число шагов. И действительно, теоретически доказано, что разрешимой процедуры для логики предикатов не существует. Тем не менее существуют разрешимые процедуры для таких подмножеств логики предикатов, как силлогизмы и пропозициональная логика. В связи с этим логику предикатов иногда называют **полуразрешимой**.

Следует отметить, что язык PROLOG основан на логике предикатов. Появление этого языка в 1970-х годах сразу же вызвало такой всплеск энтузиазма, что японские компании объявили о своих планах создания на базе данного языка ультрасовременных компьютерных систем 5-го поколения. Эти системы должны были обеспечивать возможность задавать входные и выходные данные с помощью естественного языка, а также действительно понимать все, что им сказано. Но к тому времени еще не был накоплен достаточный объем компьютеризированных знаний, основанных на здравом смысле, а микропроцессоры, выпускавшиеся в 1980-х годах, не были достаточно мощными, чтобы обеспечивать распознавание речи с высокой степенью точности. В настоящее время благодаря достижениям в области разработки онтологий, основанных на здравом смысле, подобных создаваемым в проекте `Open.Cyc`, и качественных средств распознавания речи без обучения вероятность успешного создания таких систем стала намного выше (хотя и не с использованием языка PROLOG).

Ниже приведен очень простой пример определения полной формальной системы.

- Алфавит. Единственный символ “1”.
- Аксиома. Строка “1” (которая по стечению обстоятельств совпадает с символом “1”).
- Правило вывода. Если какая-либо из строк $ является теоремой, то таковой является и строка $11. Это правило может быть записано как продукционное правило Поста следующим образом:

```
$ → $11.
   Если $ = 1, то применение этого правила приводит
   к получению $11 = 111.
   Если $ = 111, то применение этого правила приводит
   к получению $11 = 11111, и, вообще говоря,
   1, 111, 11111, 1111111, ...
```

Показанные в этом примере строки представляют собой теоремы рассматриваемой формальной системы.

Безусловно, строки, подобные 11111, не похожи на теоремы такого вида, с которыми мы привыкли сталкиваться, но они представляют собой в полном смысле слова действительные логические теоремы. Данные конкретные теоремы имеют

также семантический смысл, поскольку представляют собой нечетные числа, выраженные в **единичной системе счисления**, в которой применяется единственный символ — 1. В двоичной системе счисления имеются только символы алфавита 0 и 1, а в единичной системе счисления — лишь единственный символ 1. В табл. 3.13 приведен небольшой ряд чисел, представленных в единичной и десятичной системах счисления.

Таблица 3.13. Примеры чисел в единичной и десятичной системах счисления

Единичная	Десятичная
1	1
11	2
111	3
1111	4
11111	5

Следует отметить, что в рассматриваемой формальной системе невозможно представить такие строки, как 11, 1111 и т.д., поскольку этого не допускают правила вывода и аксиома данной системы. Это означает, что 11 и 1111, которые, безусловно, представляют собой строки, полученные с помощью алфавита формальной системы, не являются в этой системе теоремами или правильно построенными формулами, поскольку их невозможно доказать с использованием только имеющихся правила вывода и аксиомы. Данная формальная система допускает вывод только нечетных чисел, а не четных чисел. Для того чтобы иметь возможность выводить четные числа, требуется дополнительно ввести аксиому “11”.

Не менее важным свойством формальной системы является **полнота**. Множество аксиом является **полным**, если с его помощью можно доказать или **опровергнуть** любую правильно построенную формулу. Термин *опровергнуть* означает доказать, что некоторое утверждение является ложным. В полной системе каждая логически действительная правильно построенная формула является теоремой. Но, поскольку логика предикатов не разрешима, возможность получения доказательства зависит от удачи и находчивости того, кто стремится найти такое доказательство. Безусловно, еще одна возможность состоит в том, чтобы написать компьютерную программу, которая будет предпринимать попытки вывести доказательства, и загрузить ее работой.

Еще одним желательным свойством логической системы является ее **непротиворечивость**. Непротиворечивой называется система, в которой каждая теорема является логически действительной правильно построенной формулой. Иными словами, непротиворечивая система не позволяет вывести заключение, не являющееся логическим следствием из его посылок. Такая система не позволяет прий-

ти к выводу, что какое-либо недействительное доказательство является действительным.

Формальные системы образуют иерархию, определяемую тем, логика какого **порядка** в них используется. Язык **первого порядка** определен таким образом, что в нем действие кванторов распространяется на объекты, представляющие собой переменные, как в примере $\forall x$. А в языке **второго порядка** предусмотрены дополнительные средства, в частности, две разновидности переменных и кванторов. Кроме переменных и кванторов, значение которых зависит от того, какое место они занимают в правильно построенной формуле, в логике второго порядка могут применяться кванторы, переменные которых пробегают по множествам функций и предикативных символов. Примером выражения логики второго порядка может служить **аксиома равенства**, в которой утверждается, что два объекта равны, если они имеют одинаковую структуру и все применяемые в них предикаты равны. Если P — любой предикат с одним параметром, то следующее выражение представляет собой формулировку аксиомы равенства, в которой используется квантор второго порядка, $\forall P$, с областью определения в множестве всех предикатов:

$$x = y \equiv (\forall P)[P(x) \leftrightarrow P(y)]$$

3.10 Резолюция

Обычно в программах искусственного интеллекта, предназначенных для доказательства теорем, кроме всего прочего, применяется очень мощное правило вывода, называемое **резолюцией**, которое было предложено Робинсоном (Robinson) в 1965 году. Фактически резолюция является основным правилом вывода, используемым в языке PROLOG. Благодаря этому в языке PROLOG вместо различных многочисленных правил логического вывода, имеющих ограниченную область применения, таких как модус поненс, модус толленс, правило слияния (merging rule), цепное правило и т.д., используется единственное правило логического вывода общего назначения — резолюция. В результате такого использования резолюции программы автоматического доказательства теорем, такие как PROLOG, становятся практически применимыми инструментами решения задач. Таким образом, не приходится предпринимать попытки использования различных правил вывода в надежде на то, что одно из них позволит добиться успеха, поскольку достаточно воспользоваться единственным правилом резолюции. Этот подход позволяет существенно уменьшить область поиска.

Для того чтобы можно было проще ознакомиться с вводным описанием резолюции, вначале рассмотрим силлогизм о Сократе, представленный на языке PROLOG, как показано ниже; в этом фрагменте кода комментарии обозначены

знаком процента.

mortal(X):-man(X).	% Все люди смертны
man(socrates).	% Сократ — человек
:- mortal(socrates).	% Запрос — смертен ли Сократ?
yes	% Система PROLOG отвечает “да”

В языке PROLOG используется **бескванторная** система обозначений. Обратите внимание на то, что в утверждении о том, что все люди смертны, подразумевается применение квантора всеобщности, $\forall$.

Язык PROLOG основан на логике предикатов первого порядка. Но в этом языке предусмотрен также целый ряд расширений, позволяющих упростить программирование приложений. Эти специальные программные средства нарушают чистую логику предикатов, поэтому называются **экстралогическими средствами**. К ним относятся операторы ввода и вывода, оператор отсечения (который изменяет область поиска), а также операторы добавления и удаления предикатов в базе знаний (при использовании которых происходит изменение истинностных значений без каких-либо логических обоснований).

Прежде чем появится возможность применить резолюцию, правильно построенную формулу необходимо преобразовать в **нормальную**, или стандартную форму. Нормальные формы подразделяются на три основных типа: **конъюнктивная нормальная форма**, форма с логическими выражениями и подмножество последней формы с хорновскими выражениями. Основное назначение любой нормальной формы состоит в том, чтобы дать возможность представлять правильно построенные формулы в стандартной форме, в которой используются только знаки операции $\wedge$, $\vee$ и, возможно, $\sim$. После этого появляется возможность применять к правильно построенным формулам в нормальной форме, в которых удалены все прочие связки и кванторы, метод резолюции. Такое преобразование в нормальную форму является необходимым, поскольку резолюция — это операция, выполняемая над парами **дизъюнктов**, в результате которой создаются новые дизъюнкты, а правильно построенная формула упрощается.

Ниже показана правильно построенная формула в конъюнктивной нормальной форме, определенная как конъюнкция дизъюнкций, в состав которых входят **литералы**.

$$(P_1 \vee P_2 \vee ...) \wedge (Q_1 \vee Q_2 \vee \dots) \wedge \cdots \wedge (Z_1 \vee Z_2 \vee \dots)$$

Такие термы, как P_i, обязательно должны быть литералами, а это означает, что в них не должны содержаться какие-либо логические связки наподобие знаков операции импликации и **двусторонней импликации** или кванторов. Литерал — это атомарная формула или отрицаемая атомарная формула. Например, следующая

правильно построенная формула находится в конъюнктивной нормальной форме:

$$(A \vee B) \wedge (\sim B \vee C)$$

Термы в круглых скобках представляют собой выражения:

$$A \vee B \text{ и } \sim B \vee C$$

Как будет показано ниже, любая правильно построенная формула логики предикатов (включающей пропозициональную логику в качестве частного случая) может быть записана с применением выражений. Полная **форма с логическими выражениями** позволяет представить любую формулу логики предикатов, но может оказаться, что в таком виде формула выглядит неестественно или становится неудобной для чтения человеком. В основе синтаксиса языка PROLOG лежит подмножество хорновских выражений, которое позволяет значительно упростить и повысить эффективность механического доказательства теорем с помощью языка PROLOG по сравнению с такими системами, в которых используется стандартная система обозначений логики предикатов или полная форма с логическими выражениями. Как было указано в главе 1, в языке PROLOG допускается использование в голове оператора только одного выражения. А утверждения в полной форме с логическими выражениями, как правило, записываются в специальной форме, называемой формой Ковальского с логическими выражениями:

$$A_1, A_2, \ldots, A_N \rightarrow B_1; B_2; \ldots; B_M$$

Это утверждение интерпретируется как высказывание, согласно которому, если все подцели $A_1, A_2, \ldots, A_N$ являются истинными, то одно или несколько выражений из числа $B_1, B_2, \ldots, B_M$ также являются истинными. Следует отметить, что иногда в этой системе обозначений правая и левая части меняются местами и направление стрелки изменяется на противоположное. Приведенное выше выражение, записанное в стандартной системе обозначений логики предикатов, принимает такой вид:

$$A_1 \wedge A_2 \ldots A_N \rightarrow B_1 \vee B_2 \ldots B_M$$

Эту правильно построенную формулу можно представить в **дизъюнктивной форме** как дизъюнкцию литералов с использованием следующей эквивалентности:

$$p \rightarrow q \equiv \sim p \vee q$$

Это приводит к получению такой формулы:

$$\begin{aligned} A_1 \wedge A_2 \ldots A_N &\rightarrow B_1 \vee B_2 \ldots B_M \\ &\equiv \sim(A_1 \wedge A_2 \ldots A_N) \vee (B_1 \vee B_2 \ldots B_M) \\ &\equiv \sim A_1 \vee \sim A_2 \ldots \sim A_N \vee B_1 \vee B_2 \ldots B_M \end{aligned}$$

в которой для упрощения последнего выражения применяется закон де Моргана:

$$\sim(p \wedge q) \equiv \sim p \vee \sim q$$

Как было описано в главе 1, в языке PROLOG используется форма с логическими выражениями сокращенного типа, т.е. форма с хорновскими выражениями, в которой разрешена только одна голова:

$$A_1, A_2, \ldots A_N \rightarrow B$$

Приведенное выше выражение может быть записано в синтаксисе языка PROLOG следующим образом:

$$B\text{:-}\ A_1, A_2, \ldots A_N$$

При осуществлении попытки непосредственно доказать теорему возникает проблема, связанная с тем, что при осуществления дедуктивного вывода с использованием только правил вывода и аксиом системы обнаруживаются сложности. Это означает, что для вывода теоремы может потребоваться очень много времени, или даже у того, кто пытается ее доказать, может вообще не хватить временны́х или пространственных ресурсов, чтобы выполнить эту задачу. Для доказательства истинности теоремы применяется классический метод **приведения к абсурду**, или доказательства от противного. При использовании этого метода предпринимается попытка доказать, что теоремой является отрицаемая правильно построенная формула. А если в результате обнаруживается противоречие, это означает, что первоначальная неотрицаемая правильно построенная формула является теоремой.

По существу, целью операции резолюции является вывод нового выражения, **резольвенты**, из двух других выражений, называемых **родительскими выражениями**. Резольвента должна иметь меньше термов, чем родительские выражения. Продолжая процесс резолюции, можно в конечном итоге достичь противоречия; еще один вариант может состоять в том, что процесс будет окончен из-за того, что не обнаруживается какого-либо прогресса. Простой пример применения резолюции показан в следующем доказательстве:

$$\begin{array}{l} A \vee B \\ \underline{A \vee \sim B} \\ \therefore A \end{array}$$

Один из способов убедиться в том, что из этих посылок действительно следует указанное заключение, может предусматривать запись этих посылок в такой форме:

$$(A \vee B) \wedge (A \vee \sim B)$$

Для дальнейшего упрощения этой формулы может использоваться один из вариантов распределительного закона, который имеет следующий вид:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

Применяя его к посылкам, получим такое выражение:

$$(A \vee B) \wedge (A \vee \sim B) \equiv A \vee (B \wedge \sim B) \equiv A$$

Из этого следует, что мы вправе были выполнить последний этап вывода, поскольку выражение $(B \wedge \sim B)$ всегда имеет ложное значение. Правильность такого заключения подтверждается законом исключенного третьего, в котором утверждается, что ни одно выражение не может быть одновременно и истинным и ложным. Но, как показано в описании, приведенном в главе 5, в нечеткой логике этот закон не соблюдается. Еще один способ записи приведенного выше выражения может предусматривать использование терма **nil**, или **null**, который означает “пусто”, “ничто” или “ложно”. Например, пустой указатель в языке C не указывает на какой-либо действительный адрес памяти. С другой стороны, в законе исключенного третьего утверждается:

$$(B \wedge \sim B) \equiv \text{nil}$$

Приведенный выше пример выполнения операции резолюции показывает, каким образом могут быть упрощены родительские выражения $(A \vee B)$ и $(A \vee \sim B)$ и приведены к резольвенте A. В табл. 3.14 показаны некоторые простые примеры родительских выражений и их резольвент в системе обозначений с логическими выражениями, в которой запятая, разделяющая выражение, применяется вместо знака операции $\wedge$.

3.11 Системы резолюции и дедукция

Как было указано выше, если даны правильно построенные формулы $A_1, A_2, \ldots, A_N$ и логическое заключение, или теорема C, то имеет место следующее утверждение:

$$A_1 \wedge A_2 \ldots A_N \vdash C$$

Формулировка этого утверждения эквивалентна утверждению, что следующие выражения являются действительными:

(1)
$$\begin{aligned} A_1 \wedge A_2 \ldots A_N \rightarrow C &\equiv \sim(A_1 \wedge A_2 \ldots A_N) \vee C \\ &\equiv \sim A_1 \vee \sim A_2 \ldots \sim A_N \vee C \end{aligned}$$

Таблица 3.14. Выражения и резольвенты

Родительские выражения	Резольвента	Значение
$p \to q, p$ или $\sim p \vee q, p$	q	Правило модус поненс
$p \to q, q \to r$ или $\sim p \vee q, \sim q \vee r$	$p \to r$ или $\sim p \vee r$	Цепное правило или гипотетический силлогизм
$\sim p \vee q, p \vee q$	q	Правило слияния
$\sim p \vee \sim q, p \vee q$	$\sim p \vee p$ или $\sim q \vee q$	`TRUE` (тавтология)
$\sim p, p$	nil	`FALSE` (противоречие)

Предположим, что отрицание рассматриваемого взято выражения таким образом:

$$\sim[A_1 \wedge A_2 \ldots A_N \to C]$$

Теперь мы можем привести его к следующей формуле:

$$p \to q \equiv \sim p \vee q$$

поэтому приведенное выше выражение принимает такой вид:

$$\sim[A_1 \wedge A_2 \ldots A_N \to C] \equiv \sim[\sim(A_1 \wedge A_2 \ldots A_N) \vee C]$$

Согласно законам де Моргана, имеет место следующее:

$$\sim(p \vee q) \equiv \sim p \wedge \sim q$$

Таким образом, рассматриваемое выражение становится таковым:

$$\begin{aligned} \sim[A_1 \wedge A_2 \ldots A_N \to C] &\equiv [\sim \sim (A_1 \wedge A_2 \ldots A_N) \wedge \sim C] \\ &\equiv A_1 \wedge A_2 \ldots A_N \wedge \sim C \end{aligned} \tag{2}$$

Поэтому, если формула (1) является действительной, то ее отрицание (2) недействительно. Иными словами, если формула (1) — тавтология, то формула (2) должна представлять собой противоречие. Формулы (1) и (2) лежат в основе двух эквивалентных способов доказательства того, что формула C — теорема. Формула (1) может использоваться для доказательства теоремы путем проверки и определения того, является ли она истинной во всех случаях. Равным образом, формула (2) может применяться для доказательства теоремы путем демонстрации того, что формула (2) ведет к противоречию.

Как было указано в предыдущем разделе, доказательство теоремы путем демонстрации того, что ее отрицание ведет к противоречию, называется доказательством с помощью приведения к абсурду. Основной частью доказательства такого типа является **опровержение**. Опровергнуть нечто означает доказать его ложность. Резолюция — это непротиворечивое правило логического вывода, которое является также **обеспечивающим полноту опровержения**, поскольку при наличии противоречия в множестве выражений, преобразуемых с помощью этого правила, в конечном итоге результатом всегда становится пустое выражение. В данном случае слова "в конечном итоге" означают, что **опровержение по методу резолюции** завершается за конечное количество шагов, если существует противоречие. Безусловно, опровержение по методу резолюции не позволяет узнать, как сформулировать новую теорему, но определенно позволяет выяснить, является ли некоторая правильно построенная формула теоремой.

В качестве простого примера доказательства путем опровержения по методу резолюции рассмотрим следующее доказательство:

$$\begin{array}{l} A \rightarrow B \\ B \rightarrow C \\ \underline{C \rightarrow D} \\ \therefore A \rightarrow D \end{array}$$

Чтобы доказать с помощью опровержения по методу резолюции, что заключение $A \rightarrow D$ является теоремой, вначале преобразуем это выражение в дизъюнктивную форму с использованием следующей эквивалентности:

$$p \rightarrow q \equiv \sim p \vee q$$

Таким образом, получим следующую формулу:

$$A \rightarrow D \equiv \sim A \wedge D$$

Отрицание этой формулы принимает такой вид:

$$\sim(\sim A \vee D) \equiv A \wedge \sim D$$

Конъюнкция дизъюнктивных форм посылок и отрицаемого заключения приводит к созданию конъюнктивной нормальной формы, применимой для опровержения по методу резолюции:

$$(\sim A \vee B) \wedge (\sim B \vee C) \wedge (\sim C \vee D) \wedge A \wedge \sim D$$

Теперь появляется возможность применить метод резолюции к посылкам и заключению. На рис. 3.18 показано, как представить процесс опровержения по методу резолюции в форме **диаграммы дерева опровержения резолюции**, в котором

резолюция применяется к выражениям, находящимся на одном и том же уровне. Корнем дерева, представляющим собой конечную резольвенту, является выражение `nil` (в этом можно убедиться на основании данных для $\sim p, p$, приведенных в последней строке табл. 3.14). Это означает, что первоначальное заключение $A \rightarrow D$ представляет собой теорему.

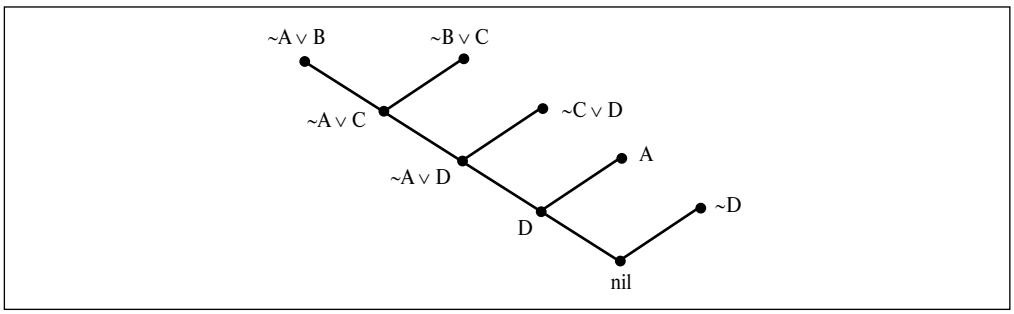


Рис. 3.18. Дерево опровержения резолюции

3.12 Поверхностные и причинные рассуждения

Для доказательства теорем часто применяются два таких подхода, как системы резолюции и продукционные системы. Безусловно, большинство людей считают, что теорема — предмет рассмотрения математики, но авторы придерживаются такого мнения, что на самом деле теорема представляет собой заключение действительного логического доказательства. Теперь рассмотрим экспертную систему, в которой используется цепь логических выводов. Вообще говоря, более длинная цепь выводов представляет знания, которые являются в большей степени причинными или глубокими, а в более поверхностных рассуждениях обычно используется либо единственное правило, либо всего лишь немного этапов логического вывода. Еще одним важным фактором проведения различий между глубокими и поверхностными рассуждениями, кроме длины цепи выводов, является также качество знаний, содержащихся в правилах. Для определения понятия поверхностных знаний иногда используется другое определение, согласно которому их называют **практическими знаниями**, т.е. знаниями, основанными на опыте.

Как показывает приведенный выше пример, заключением цепи логического вывода становится теорема, поскольку цепь логического вывода служит для нее доказательством:

$$A \rightarrow B, B \rightarrow C, C \rightarrow D \vdash A \rightarrow D$$

Фактически в экспертных системах, в которых для получения заключения применяется цепь логического вывода, используются теоремы. Этот результат очень важен, поскольку в противном случае мы не могли бы применять экспертные системы для причинного логического вывода. Вместо этого эксплуатация экспертных систем ограничивалась бы поверхностными логическими выводами на основании отдельно взятых правил, без формирования каких-либо цепей логического вывода.

Для того чтобы было проще различать поверхностные и глубокие рассуждения, рассмотрим несколько правил. В качестве первого примера разберем следующее правило, в котором число в круглых скобках служит только для обозначения правила:

```
(1) IF в комплект автомобиля входят
       исправный аккумулятор
       исправные свечи
       бензин
       исправные шины
    THEN автомобиль готов к эксплуатации
```

Это — вполне качественное правило, которое может применяться в экспертной системе.

Как было описано в главе 1, одной из важных характерных особенностей экспертных систем является наличие в них средства объяснения. Подход, в котором применяются экспертные системы, основанные на правилах, позволяет легко создавать системы, способные дать объяснение сформированным в них рассуждениям. А в данном случае, если пользователь задаст вопрос, как подготовить автомобиль к эксплуатации, то экспертная система может ответить, перечислив условные элементы правила, следующим образом:

```
исправный аккумулятор
исправные свечи
бензин
исправные шины
```

Это — простейший пример применения средства объяснения, поскольку в нем система перечисляет только условные элементы правила. Кроме того, могут быть спроектированы более сложные средства объяснения, обеспечивающие составление списка ранее запущенных правил и указания причин запуска текущего правила. Другие средства объяснения могут дать возможность пользователю задавать вопросы типа “что, если” и исследовать альтернативные пути формирования рассуждений.

Приведенное выше правило является также примером применения подхода, основанного на **поверхностных рассуждениях**. Таким образом, в поверхностных рассуждениях не обнаруживается понимание или проявляется лишь элементарное

понимание причин и результатов, поскольку цепь логического вывода коротка или вообще отсутствует. Рассматриваемое правило по существу представляет собой эвристику, в которой все знания содержатся в правиле. Правило активизируется после того, как выполняются его условные элементы, а не в связи с тем, что в экспертной системе обнаруживается какое-либо понимание того, какие функции реализуются условными элементами. В поверхностных рассуждениях обнаруживается лишь небольшая **причинная цепь** (или таковая вообще отсутствует), связывающая причины и результат одного правила с другим. А в простейшем случае информация о причинах и результате содержится только в одном правиле, не имеющем никакой связи с любыми другими правилами. Если правила рассматриваются в терминах фрагментов знаний, описанных в главе 1, то можно отметить, что в поверхностных рассуждениях не создаются соединения между такими фрагментами и поэтому рассуждение в большей степени напоминает простую рефлекторную реакцию.

Преимуществом подхода, в котором используются поверхностные рассуждения, по сравнению с подходом, основанном на причинных рассуждениях, является простота программирования. Упрощение программирования приводит к тому, что продолжительность разработки сокращается, а сама программа становится менее крупной и более быстродействующей, причем ее разработка обходится дешевле.

Для реализации подхода, основанного на причинных, или **глубоких рассуждениях**, могут применяться фреймы. Термин *глубокие рассуждения* часто используется как синоним термина причинные рассуждения. Это говорит о том, что для реализации такого подхода требуется глубокое понимание предмета. А из того, что понимание должно быть достаточно глубоким, следует, что необходимо понимать не только причинную цепь, согласно которой протекает процесс, но и понимать сам этот процесс в абстрактном смысле.

Для обеспечения возможности применения рассматриваемого правила в простых причинных рассуждениях можно ввести некоторые дополнительные правила, например, показанные ниже.

```
(2) IF аккумулятор исправен
    THEN обеспечивается подача электроэнергии
(3) IF обеспечивается подача электроэнергии
       и свечи исправны
    THEN свечи вырабатывают искровые импульсы
(4) IF свечи вырабатывают искровые импульсы
       и имеется бензин
    THEN двигатель будет работать
(5) IF двигатель работает и
       шины исправны
 THEN автомобиль готов к эксплуатации
```

Следует отметить, что если предусмотрена возможность формирования причинных рассуждений, то средство объяснения позволяет составить качественное объяснение того, какое назначение имеет каждый узел автомобиля, поскольку для каждого условного элемента правила (1) предусмотрено отдельное правило. Такая причинная система позволяет также проще создавать диагностические системы, позволяющие определить, к какому результату приводит выход из строя какого-либо компонента. Причинные рассуждения могут использоваться для усовершенствования функционирования системы, которое может продолжаться настолько долго, насколько позволяют ограничения по скорости выполнения, объему памяти и по финансовым ресурсам, выделенным на разработку.

Причинные рассуждения могут использоваться для создания **модели** реальной системы, которая ведет себя во всех отношениях подобно реальной системе. Такая модель может применяться для моделирования системы и изучения результатов гипотетических рассуждений, формируемых при поиске ответов на вопросы типа "что, если". Но причинные модели не всегда являются необходимыми или желательными. Например, классическая экспертная система MUD используется в качестве консультанта для инженеров, которые занимаются подготовкой и обработкой бурового раствора (mud, отсюда название системы). Буровой раствор является важным вспомогательным средством бурения по многим причинам; в частности, он обеспечивает охлаждение и смазывание бурового инструмента, находящегося в скважине. Система MUD диагностирует проблемы, связанные с использованием бурового раствора, и предлагает способы их устранения.

В данном случае причинная система не была бы очень полезной, поскольку обычно инженер-буровик не может наблюдать за причинной цепью событий, происходящих в земле на большой глубине. Вместо этого инженер может наблюдать только симптомы, обнаруживающиеся на поверхности, и не знает о происходящих ненаблюдаемых промежуточных событиях, имеющих потенциальную диагностическую значимость.

Совсем другая ситуация наблюдается в медицине, поскольку в распоряжении врачей имеется широкий набор диагностических проверок, которые могут использоваться для контроля над промежуточными событиями. Например, если пациент жалуется на плохое самочувствие, то врач может проверить, нет ли у него высокой температуры. А если есть высокая температура, то причиной ее может стать инфекция, поэтому потребуется провести анализ крови. Если же анализ крови показывает наличие столбнячной инфекции, то врач может проверить предположение о том, что пациент недавно получил рану в результате соприкосновения с ржавым предметом. В отличие от этого, если буровой раствор становится соленым, то инженер-буровик может выдвинуть предположение, что бур проходит через соляной купол. Но не существует простого способа проверки этого предположения, поскольку невозможно проникнуть на соответствующую глубину в скважину (кроме как с помощью робота), а сейсмическое испытание является дорогосто-

ящим и не всегда надежным. Это означает, что инженеры-буровики обычно не имеют возможности выполнять проверку промежуточных гипотез по такому же принципу, как это делают врачи, поэтому инженеры-буровики не могут использовать такой же подход к решению диагностических задач, какой применяется врачами, и это различие в подходах отражено в системе MUD.

Еще одна причина, по которой в системе MUD не используются причинные рассуждения, состоит в том, что в данной предметной области существует ограниченное количество диагностических возможностей и симптомов. Большая часть относящихся к этой области проверок, используемых в системе MUD, проводится простейшим образом, поэтому их данные вводятся заранее. Это означает, что организация интерактивных сеансов с вопросами и ответами с участием инженеров, в которых исследовались бы альтернативные диагностические гипотезы, практически не дает никаких преимуществ, поскольку системе известны все относящиеся к данной теме результаты проверок и диагностические пути. А если бы существовали многочисленные возможные диагностические пути, по которым можно было бы проследовать, исходя из подлежащей проверке промежуточной гипотезы, то наличие причинных знаний дало бы определенные преимущества, поскольку инженер получил бы возможность работать с системой и отсекать бесперспективные направления поиска по возможным путям.

Применение причинных рассуждений связано с реализацией повышенных требований к системе, поэтому может потребоваться объединить несколько правил в одно, обеспечивающее возможность проведения поверхностных рассуждений. А для доказательства того, что какое-то единственное правило является истинным заключением для нескольких правил, можно воспользоваться методом резолюции с опровержением. Это единственное правило становится теоремой, подлежащей доказательству с помощью резолюции.

В качестве примера предположим, что правило (1) является логическим следствием из правил (2)–(5). Ниже показаны пропозициональные определения, применяемые для представления правил, и сами правила, выраженные с их помощью.

B — аккумулятор исправен

C — автомобиль готов к эксплуатации

E — обеспечивается подача электроэнергии

F — свечи вырабатывают искровые импульсы

G — имеется бензин

R — двигатель будет работать

S — свечи исправны

T — шины исправны

(1) $B \wedge S \wedge G \wedge T \rightarrow C$

(2) $B \rightarrow E$

(3) $E \wedge S \rightarrow F$

(4) $F \wedge G \rightarrow R$

(5) $R \wedge T \rightarrow C$

Первый этап применения опровержения по методу резолюции состоит в том, что формируется отрицание заключения или целевого правила:

$$(1') \qquad \sim(B \wedge S \wedge G \wedge T \rightarrow C) = \sim[\sim(B \wedge S \wedge G \wedge T) \vee C] = \sim[\sim B \vee \sim S \vee \sim G \vee \sim T \vee C]$$

Теперь представим каждое из оставшихся правил в дизъюнктивной форме с использованием эквивалентностей, примерно таким образом:

$$p \rightarrow q \equiv \sim p \vee q \text{ и } \sim (p \wedge q) \equiv \sim p \vee \sim q$$

Выполнение этих преобразований приводит к получению следующих новых версий правил (2)–(5):

(2′) $\sim B \vee E$

(3′) $\sim(E \wedge S) \vee F = \sim E \vee \sim S \vee F$

(4′) $\sim(F \wedge G) \vee R = \sim F \vee \sim G \vee R$

(5′) $\sim(R \wedge T) \vee C = \sim R \vee \sim T \vee C$

Как описано в предыдущем разделе, удобный способ представления последовательно формируемых резольвент правил (1')–(5') состоит в использовании дерева опровержения резолюции, как показано на рис. 3.19. Начиная с вершины дерева, выражения представляются в виде узлов, к которым применяется правило резолюции для получения резольвент, находящихся на более низком уровне. Например, применение правила резолюции к следующим выражениям:

$$\sim B \vee E \text{ и } \sim E \vee \sim S \vee F$$

приводит к получению такой резольвенты:

$$\sim B \vee \sim S \vee F$$

Это выражение может использоваться в правиле резолюции вместе со следующим выражением:

$$\sim F \vee \sim G \vee R$$

для получения приведенной ниже резольвенты, и т.д.

$$\sim B \vee \sim S \vee \sim G \vee R$$

Для упрощения диаграммы в ней просто подразумевается наличие последних резольвент, а не изображается каждая отдельная резольвента.

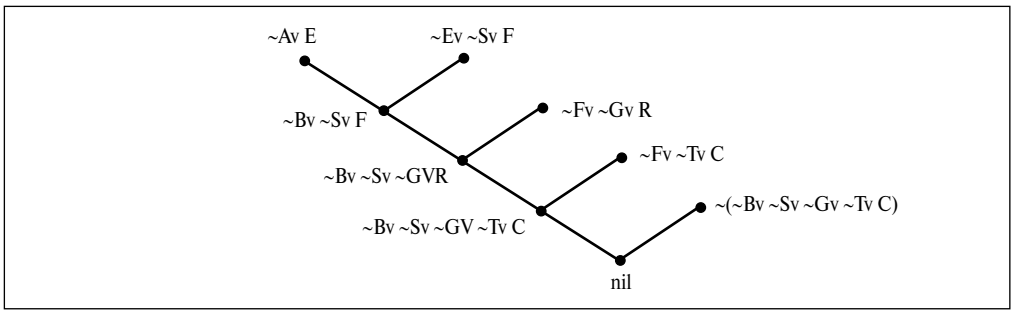


Рис. 3.19. Дерево опровержения резолюции для примера с автомобилем

Корнем этого дерева является выражение `nil`, представляющее собой противоречие. Согласно методу опровержения, первоначальное заключение — это теорема, поскольку отрицание этого заключения ведет к противоречию:

$$B \wedge S \wedge G \wedge T \rightarrow C$$

Итак, правило (1) логически следует из правил (2)–(5).

3.13 Резолюция и логика предикатов первого порядка

Метод резолюции используется также в логике предикатов первого порядка. Фактически этот метод представляет собой основной механизм логического вывода в языке PROLOG. Но, прежде чем появится возможность применить резолюцию, необходимо привести правильно построенную формулу в форму с логическими выражениями. В качестве примера рассмотрим следующее доказательство:

Некоторые программисты ненавидят все нарушения в работе
Ни один программист не ненавидит любой успех
∴ Ни одно нарушение в работе не является успехом

Определим такие предикаты:

$$P(x) = x - \text{программист}$$
$$F(x) = x - \text{нарушение в работе}$$
$$S(x) = x - \text{успех}$$
$$H(x, y) = x \text{ ненавидит } y$$

В таком случае посылки и отрицаемое заключение можно записать, как показано ниже, где к заключению была применена операция отрицания в целях подготовки его к применению в методе резолюции.

(1) $(\exists x)[P(x) \wedge (\forall y)(F(y) \rightarrow H(x, y))]$

(2) $(\forall x)[P(x) \rightarrow (\forall y)(S(y) \rightarrow \sim H(x, y))]$

(3) $(\forall y)(F(y) \rightarrow \sim S(y))$

Преобразование в форму с логическими выражениями

Девять описанных ниже шагов представляют собой алгоритм преобразования правильно построенных формул логики предикатов первого порядка в форму с логическими выражениями. Эта процедура иллюстрируется на примере приведенной выше правильно построенной формулы (1).

1. Устранить знаки условной операции, $\rightarrow$, с помощью следующей эквивалентности:

   $$p \rightarrow q \equiv \sim p \vee q$$

   В результате выполнения этого этапа правильно построенная формула (1) принимает такой вид:

   $$(\exists x)[P(x) \wedge (\forall y)(\sim F(y) \vee H(x, y))]$$

2. По возможности устранить знаки операции отрицания или свести область определения операции отрицания до одного атома, используя при этом такие эквивалентности, показанные в приложении А, как:

   $$\sim \sim p \equiv p$$
   $$\sim(p \wedge q) \equiv \sim p \vee \sim q$$
   $$\sim(\exists x)P(x) \equiv (\forall x) \sim P(x)$$
   $$\sim(\forall x)P(x) \equiv (\exists x) \sim P(x)$$

3. Стандартизировать переменные, применяемые в правильно построенных формулах, таким образом, чтобы связанные или фиктивные переменные

каждого квантора имели уникальные имена. Обратите внимание на то, что имена переменных в кванторах являются фиктивными. Это означает, что имеет место формула:

$$(\forall x)P(x) \equiv (\forall y)P(y) \equiv (\forall z)P(z)$$

поэтому стандартизированная форма выражения:

$$(\exists x) \sim P(x) \vee (\forall x)P(x)$$

принимает следующий вид:

$$(\exists x) \sim P(x) \vee (\forall y)P(y)$$

4. Устранить кванторы существования, $\exists$, с использованием **сколемовских функций**, названных в честь норвежского логика Торалфа Сколема (Thoralf Skolem). Рассмотрим следующую правильно построенную формулу, в которой выражение $L(x)$ определено как предикат, принимающий истинное значение, если $x < 0$:

$$(\exists x)L(x)$$

Эта правильно построенная формула может быть заменена следующей:

$$L(a)$$

В данном выражении a — константа, такая как -1, в результате применения которой выражение $L(a)$ принимает истинное значение. Такая константа a называется *сколемовской константой* и представляет собой частный случай сколемовской функции. В том случае, в котором перед квантором существования находится квантор всеобщности, например, как в следующем выражении, где предикат $L(x, y)$ принимает истинное значение, если целое число x меньше целого числа y:

$$(\forall x)(\exists y)L(x, y)$$

Эта правильно построенная формула означает, что для каждого целого числа x можно найти целое число y, которое больше по сравнению с x. Обратите внимание на то, что в этой формуле не указано, как вычислить y, если дано значение x. Предположим, что существует функция $f(x)$, которая вырабатывает число y, большее чем x. Поэтому приведенная выше правильно построенная формула становится сколемизированной таким образом:

$$(\forall x)L(x, f(x))$$

Сколемовская функция от переменной квантора существования, заданной в области определения квантора всеобщности, представляет собой функцию от всех переменных кванторов, находящихся в левой части выражения. Например, следующая формула:

$$(\exists u)(\forall v)(\forall w)(\exists x)(\forall y)(\exists z)P(u,v,w,x,y,z)$$

сколемизируется, как показано ниже, где a — та же константа, а вторая сколемовская функция, g, должна отличаться от первой функции, f.

$$(\forall v)(\forall w)(\forall y)P(a,v,w,f(v,w),y,g(v,w,y))$$

Таким образом, правильно построенная формула, применяемая в качестве примера, принимает следующий вид:

$$P(a) \wedge (\forall y)(\sim F(y) \vee H(a,y))$$

5. Преобразовать правильно построенную формулу в **предваренную форму**, представляющую собой последовательность кванторов Q, за которой следует **матрица** M. Вообще говоря, в качестве кванторов могут применяться либо $\forall$, либо $\exists$. Но в данном случае на этапе 4 уже должны были быть устранены все кванторы существования, поэтому в последовательности Q могут присутствовать только кванторы $\forall$. Кроме того, в каждом кванторе $\forall$ имеется его собственная фиктивная переменная. Таким образом, все кванторы $\forall$ могут быть передвинуты в левую часть правильно построенной формулы, и поэтому областью определения каждого квантора $\forall$ может стать вся правильно построенная формула.

   Рассматриваемый пример принимает следующий вид, где в качестве матрицы используется терм в круглых скобках:

$$(\forall y)[P(a) \wedge \sim F(y) \vee H(a,y)]$$

6. Преобразовать матрицу в конъюнктивную нормальную форму, представляющую собой конъюнкцию из выражений, в которой каждое выражение является дизъюнкцией. Рассматриваемый пример уже находится в конъюнктивной нормальной форме, в которой одним выражением служит $P(a)$, а другим — $(\sim F(y) \vee H(a,y))$. В случае необходимости можно использовать следующее распределительное правило для преобразования матрицы в конъюнктивную нормальную форму:

$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

7. Удалить как ненужные кванторы всеобщности, поскольку все переменные в правильно построенной формуле на этом этапе уже должны быть связанными. Теперь правильно построенная формула имеет вид матрицы. В данном примере правильно построенная формула становится таковой:

$$P(a) \wedge (\sim F(y) \vee H(a, y))$$

8. Удалить знаки операции $\wedge$, записав правильно построенную формулу как множество выражений. В данном примере формула имеет вид

$$\{P(a), \sim F(y) \vee H(a, y)\}$$

и должна быть записана без фигурных скобок как следующий ряд выражений:

$$P(a)$$
$$\sim F(y) \vee H(a, y)$$

9. В случае необходимости переименовать переменные в выражениях для того, чтобы одно и то же имя переменной использовалось только в одном выражении. Например, если в нашем распоряжении имеются такие выражения:

$$P(x) \wedge Q(x) \vee L(x, y)$$
$$\sim P(x) \vee Q(y)$$
$$\sim Q(z) \vee L(z, y)$$

то их можно было бы переименовать следующим образом:

$$P(x1) \wedge Q(x1) \vee L(x1, y1)$$
$$\sim P(x2) \vee Q(y2)$$
$$\sim Q(z) \vee L(z, y3)$$

После проведения этой процедуры для преобразования в форму с логическими выражениями второй посылки и отрицаемого заключения, рассматриваемых в данном примере, в конечном итоге будут получены выражения, номера которых соответствуют номерам исходных посылок и отрицаемого заключения:

$(1a)$ $P(a)$

$(1b)$ $\sim F(y) \vee H(a, y)$

$(2a)$ $\sim P(x) \vee \sim S(y) \vee \sim H(x, y)$

$(3a)$ $F(b)$

$(3b)$ $S(b)$

Таким образом, посылка (1) и заключение (3) преобразуются соответственно в два выражения с суффиксами, (1a), (1b) и (3a), (3b), а посылка (2) — в одно выражение, (2a).

Унификация и правила

После преобразования правильно построенных формул в форму с логическими выражениями обычно возникает необходимость найти подходящие **подстановочные экземпляры** для переменных. Дело в том, что, например, следующие два выражения нельзя привести по методу резолюции к предикату F до тех пор, пока параметры F в обоих выражениях не будут совпадать:

$$\sim F(y) \vee H(a, y)$$
$$F(b)$$

Процесс поиска переменных, которые могут быть подставлены для того, чтобы параметры стали совпадающими, называется **унификацией**.

Унификация — это одна из особенностей, которая отличает экспертные системы от систем с простыми деревьями решений. Без унификации можно было бы согласовывать в условных элементах правил только константы. Это означает, что приходилось бы записывать конкретное правило для каждого возможного факта. Например, предположим, что разработчики системы противопожарной защиты решили предусмотреть включение звукового сигнала пожарной тревоги при обнаружении дыма одним из датчиков. Причем, если бы количество датчиков было равно N, то потребовалось бы отдельное правило для каждого датчика, как в следующем примере:

```
IF датчик 1 показывает наличие дыма THEN включить пожарную
   сирену 1
IF датчик 2 показывает наличие дыма THEN включить пожарную
   сирену 2
...
IF датчик N показывает наличие дыма THEN включить пожарную
   сирену N
```

Но благодаря унификации появляется возможность использовать в качестве идентификатора датчика единственную переменную $?N$, что позволяет записать одно правило, следующим образом:

```
IF датчик ?N показывает наличие дыма THEN включить
   пожарную сирену ?N
```

После унификации двух взаимно дополнительных литералов их можно удалить с помощью резолюции. Например, если рассматриваются два приведенных

выше выражения, то подстановка b вместо y позволяет получить следующее:

$$\sim F(b) \vee H(a,b)$$
$$F(b)$$

Теперь предикат F унифицирован и может быть удален с помощью резолюции, что приводит к преобразованию этих двух выражений в одно:

$$H(a,b)$$

Вообще говоря, подстановка определяется как одновременная замена переменных термами, которые рассматриваются как отличные от переменных. В качестве термов могут использоваться константы, переменные и функции. Конкретный вариант подстановки термов вместо переменных представляется в виде множества, примерно так:

$$\{t_1/v_1, t_2/v_2, \ldots, t_N/v_N\}$$

Если θ — такое множество, а A — доказательство, то $A\theta$ определяется как подстановочный экземпляр A. Например, если заданы следующие подстановка и доказательство:

$$\theta = \{a/x, f(y)/y, x/z\}$$
$$A = P(x) \vee Q(y) \vee R(z)$$

то подстановочный экземпляр принимает такой вид:

$$A\theta = P(a) \vee Q(f(y)) \vee R(x)$$

Обратите внимание на то, что подстановка выполняется одновременно во всех выражениях, поэтому был получен терм $R(x)$, а не $R(a)$.

Предположим, что имеются два выражения, C_1 и C_2, определенные следующим образом:

$$C_1 = \sim P(x)$$
$$C_2 = P(f(x))$$

Одной из возможных унификаций предиката P является следующая:

$$C_1' = C_1\{f(x)/x\} = \sim P(f(x))$$

Еще одна возможная унификация для предиката P может предусматривать две подстановки с использованием константы a:

$$C_1'' = C_1\{f(a)/x\} = \sim P(f(a))$$
$$C_2' = C_2\{a/x\} = P(f(a))$$

Следует отметить, что подстановочный экземпляр $P(f(x))$ является более общим, чем $P(f(a))$, поскольку подстановка некоторой константы вместо x позволит в дальнейшем получить бесконечное количество экземпляров $P(f(x))$. Поэтому выражение, подобное C_1, называется **наиболее общим выражением**.

Вообще говоря, подстановка θ называется **унификатором** для множества выражений $\{A_1, A_2, \ldots, A_n\}$ тогда и только тогда, когда $A_1\theta = A_2\theta = \ldots A_n\theta$. Множество выражений, для которого может быть предусмотрен некоторый унификатор, называется **унифицируемым**. А **наиболее общим унификатором** (**most general unifier — mgu**) называется такой унификатор, по отношению к которому все прочие унификаторы являются экземплярами. Это определение можно выразить более формально, указав, что θ представляет собой наиболее общий унификатор тогда и только тогда, когда для каждого унификатора α множества выражений имеется подстановка β, такая, что

$$\alpha = \theta\beta$$

В этом выражении $\theta\beta$ — это составная подстановка, созданная путем применения вначале подстановки θ, а затем подстановки β. **Алгоритмом унификации** называется метод поиска наиболее общего унификатора для конечного унифицируемого множества доказательств.

Для рассматриваемого примера, состоящего из выражений (1a)–(3b), результаты подстановки и унификации показаны в дереве опровержения резолюции (рис. 3.20). Корнем этого дерева является выражение `nil`, а это означает, что отрицаемое заключение является ложным, поэтому заключение, что “ни одно нарушение в работе не является успехом”, является действительным.

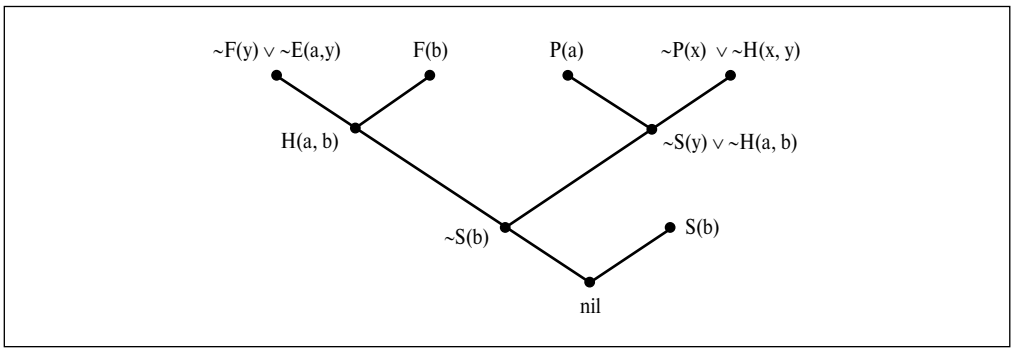


Рис. 3.20. Дерево опровержения резолюции, применяемое для доказательства того, что ни одно нарушение в работе не является успехом

Рассматриваемые до сих пор примеры были простыми, а выполняемые в них операции резолюции — несложными, хотя и утомительными для человека. Но во многих других ситуациях процесс резолюции может зайти в тупик, что приводит к возникновению необходимости выполнить возврат, чтобы попытаться найти

альтернативные варианты выбора выражений для резолюции. Безусловно, метод резолюции является очень мощным и служит основой языка PROLOG, но при решении некоторых задач он может оказаться неэффективным. Одним из недостатков метода резолюции является то, что в нем не предусмотрена действенная встроенная стратегия поиска, поэтому программист вынужден предусматривать применение эвристических средств, таких как операторы отсечения языка PROLOG, для обеспечения эффективного поиска.

Был исследован целый ряд модифицированных версий резолюции, таких как повышение приоритета единичных выражений, резолюция по входным данным, линейная резолюция и применение множества поддержки. Основным преимуществом резолюции является то, что она представляет собой единственный мощный метод, подходящий для многих случаев. Благодаря использованию резолюции становится проще создавать универсальные системы логического вывода, такие как PROLOG, чем системы, в которых предпринимается попытка реализовать много разных правил логического вывода. Еще одно преимущество резолюции состоит в том, что в случае успешного завершения процесса резолюции автоматически формируется доказательство, в котором демонстрируется последовательность этапов, приведших к получению выражения `nil`.

3.14 Прямой и обратный логический вывод

Группа из нескольких этапов логического вывода, соединяющих задачу с ее решением, называется **цепью** (логического вывода). Цепь, поиск которой или переход по которой осуществляется от задачи к ее решению, называется *прямой цепью*. Для описания процесса построения прямой цепи (называемого также *формированием прямого логического вывода*) применяется еще одно определение — проведение рассуждений от фактов к заключениям, которые следуют из этих фактов. А цепь, переход по которой происходит от гипотезы обратно к фактам, поддерживающим эту гипотезу, называется *обратной цепью*. Еще одним способом описания процесса построения обратной цепи (называемого также *формированием обратного логического вывода*) является определение, в котором речь идет о цели, достигаемой путем выполнения подцелей. Как показывают эти определения, выбор терминологии, используемой для описания прямого и обратного логического вывода, зависит от рассматриваемой задачи.

Процессы построения прямой или обратной цепи можно легко описать в терминах логического вывода. Например, предположим, что в нашем распоряжении имеются следующие правила типа модус поненс:

$$\frac{\begin{array}{l} p \rightarrow q \\ p \end{array}}{\therefore q}$$

А с помощью этих правил формируется примерно такая цепь логического вывода:

$$\text{elephant}(x) \rightarrow \text{mammal}(x)$$
$$\text{mammal}(x) \rightarrow \text{animal}(x)$$

Эти правила могут использоваться в причинной цепи прямого логического вывода, позволяющей прийти к дедуктивному заключению, согласно которому, если дано, что Клайд — слон, из этого следует, что Клайд — животное. Такая цепь логического вывода показана на рис. 3.21. Обратите внимание на то, что та же диаграмма может также служить иллюстрацией процесса обратного логического вывода.

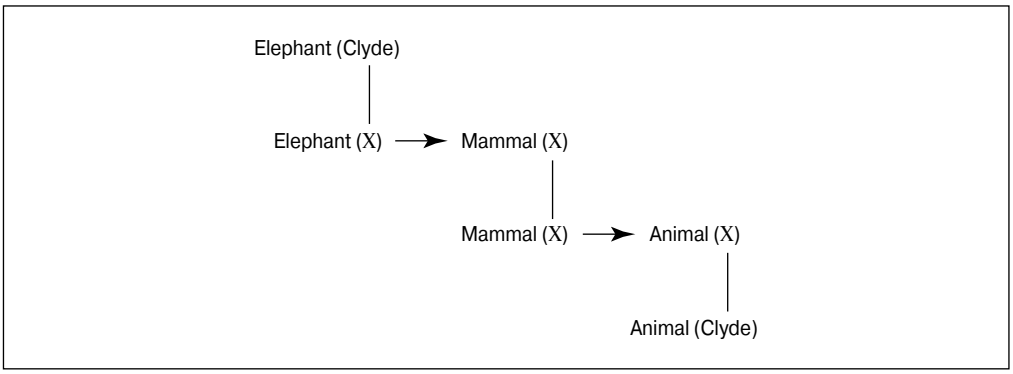


Рис. 3.21. Причинная прямая цепь

На рис. 3.21 причинная цепь представлена в виде последовательности **вертикальных черт** (|), соединяющих консеквент одного правила с антецедентом другого. Вертикальная черта обозначает также унификацию переменных с фактами. Например, прежде чем появится возможность применить правило для слонов (elephant), необходимо вначале унифицировать переменную x в предикате $\text{elephant}(x)$ с учетом факта $\text{elephant}(\text{Clyde})$. Данная причинная цепь фактически представляет собой последовательность операций импликации и унификации, как показано на рис. 3.22.

Обратный логический вывод представляет собой противоположный процесс. Предположим, что требуется доказать гипотезу $\text{animal}(\text{Clyde})$. Центральная задача обратного логического вывода состоит в поиске цепи, связывающей свидетельство с гипотезой. В проблематике обратного логического вывода факт $\text{elephant}(\text{Clyde})$ называется **свидетельством** для указания на то, что этот факт используется для обоснования гипотезы, по такому же принципу, как свидетельство в суде применяется для доказательства виновности или невиновности ответчика.

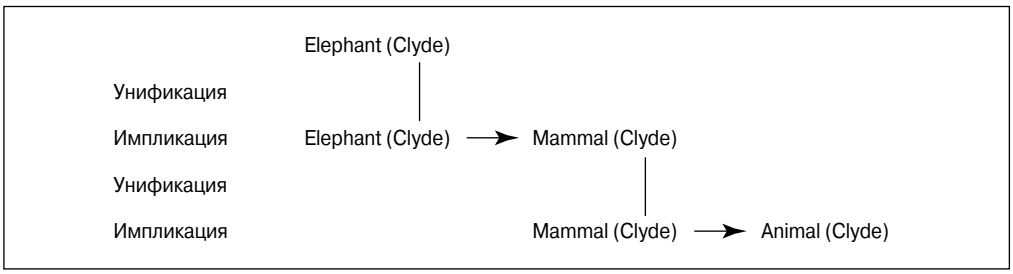


Рис. 3.22. Явная причинная цепь

В качестве простого примера прямого и обратного логического вывода предположим, что вы управляете автомобилем и внезапно обнаруживаете полицейский автомобиль с мерцающими проблесковыми маячками и включенной сиреной. Применяя прямой логический вывод, вы можете прийти к заключению, что полицейские требуют, чтобы остановились вы или кто-либо другой. Это означает, что в данном случае исходные факты служат обоснованием для двух возможных заключений. Если же полицейский автомобиль пристроится прямо за вашим автомобилем или вам сделает знак жезлом полицейский, то дополнительные этапы логического вывода позволят прийти к заключению, что требование остановиться относится к вам, а не к кому-либо другому. Приняв это за рабочую гипотезу, вы можете применить обратный логический вывод для рассуждений о том, чем вызвано это требование.

В качестве некоторых возможных промежуточных гипотез можно принять предположение, что вы оставили мусор, превысили скорость, используете неисправное оборудование или ведете краденый автомобиль. Теперь вы можете приступить к изучению свидетельств, позволяющих обосновать такие промежуточные гипотезы. Была ли причиной преследования вашего автомобиля та бутылка из-под пива, которую вы выбросили из окна, проезд на скорости 100 миль в час по участку дороги с ограничением скорости в 30 миль в час, разбитые задние сигнальные огни или номерные знаки автомобиля, которые указывает на то, что вы ведете краденый автомобиль? Предположим, что в данном случае каждый фрагмент свидетельства поддерживает промежуточную гипотезу, поэтому все эти фрагменты являются истинными. Это означает, что любая из промежуточных гипотез или все гипотезы являются возможными основаниями, позволяющими доказать рабочую гипотезу, что требование остановиться относится именно к вам.

Часто бывает полезно визуально представить ход прямого и обратного логического вывода в терминах пути через пространство задач, в котором промежуточные состояния соответствуют промежуточным гипотезам при обратном логическом выводе или промежуточным заключениям при прямом логическом выводе. В табл. 3.15 приведены некоторые общие сведения о характерных особенностях

прямого и обратного логического вывода. Следует учитывать, что характерные особенности, описанные в этой таблице, могут применяться только в качестве общего руководства. Безусловно, возможно решать задачи диагностирования в системе прямого логического вывода и задачи планирования — в системе обратного логического вывода. В частности, возможность получения объяснений в цепи обратного логического вывода обеспечивается за счет того, что система может легко сформировать точное объяснение того, какую цель она пытается выполнить. А система прямого логического вывода не позволяет так же легко получить объяснение, поскольку подцели не становятся явно известными до тех пор, пока не будут обнаружены.

Таблица 3.15. Некоторые характерные особенности прямого и обратного логического вывода

Прямой логический вывод	**Обратный логический вывод**
Планирование, текущий контроль, управление	Диагностика
От настоящего к будущему	От настоящего к прошлому
От антецедента к консеквенту	От консеквента к антецеденту
Управляемые данными, восходящие рассуждения	Управляемые целями, нисходящие рассуждения
Поиск в прямом направлении для определения того, какие решения следуют из фактов	Поиск в обратном направлении для определения фактов, обосновывающих текущую гипотезу
Обеспечивает поиск в ширину	Обеспечивает поиск в глубину
Поиск определяется антецедентами	Поиск определяется консеквентами
Не способствует формированию объяснения	Способствует формированию объяснения

Основные понятия прямого логического вывода в системе, основанной на правилах, иллюстрируются на рис. 3.23. Правила активизируются с учетом наличия фактов, которые соответствуют антецедентам или левым частям (Left-Hand-Side — LHS) этих правил. Например, для того, чтобы было активизировано правило R_1, ему должны быть поставлены в соответствие факты B и C. Но в системе присутствует только факт C, и поэтому правило R_1 не активизируется. Правило R_2 активизируется фактами C и D, которые присутствуют в системе, поэтому правило R_2 вырабатывает промежуточный факт H. Другими выполненными правилами являются R_3, R_6, R_7, R_8 и R_9. В результате выполнения правил R_8 и R_9 вырабатываются заключения данного процесса прямого логического вывода. Этими заключениями могут быть другие факты, выходные данные и т.д.

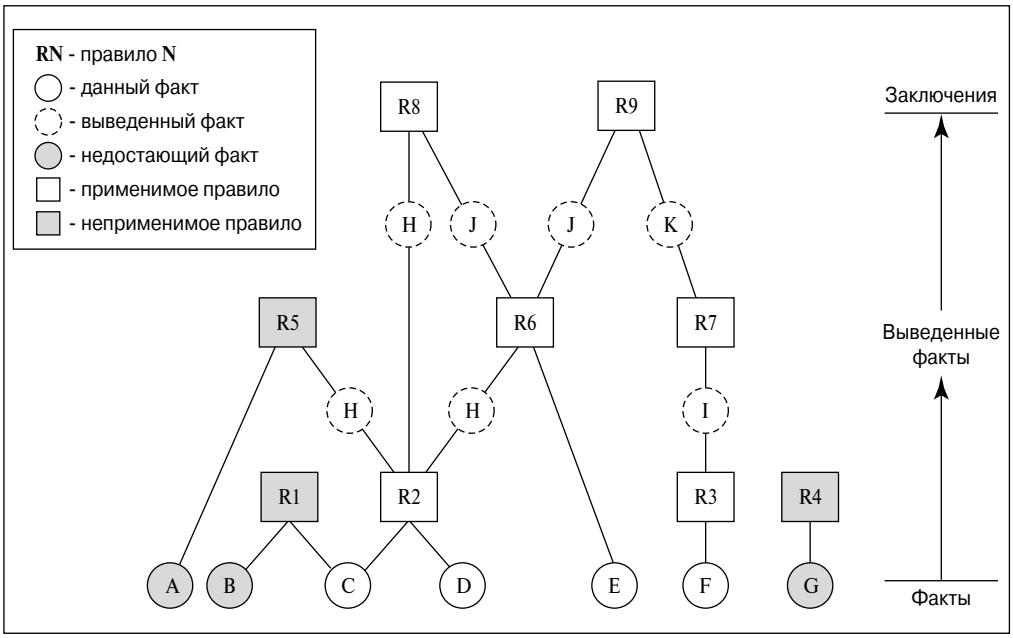


Рис. 3.23. Прямой логический вывод

Процесс прямого логического вывода называется **восходящими рассуждениями**, поскольку в нем рассуждения осуществляются от свидетельств нижнего уровня, т.е. фактов, к заключениям верхнего уровня, которые основаны на этих фактах. Процесс формирования восходящих рассуждений в экспертной системе аналогичен обычному восходящему программированию, о котором речь шла в главе 1. В подходе, основанном на знаниях, факты рассматриваются как элементарные единицы знаний, поскольку невозможна их декомпозиция на какие-либо меньшие единицы знаний, имеющие смысл. Например, факт "`duck`" имеет определенный смысл как существительное ("утка") и как глагол ("нырять"). Но результатом его дальнейшей декомпозиции становятся буквы "`d`", "`u`", "`c`" и "`k`", которые не имеют какого-либо конкретного смысла. А в обычных программах основными единицами значения являются **данные**.

Обычно применяется соглашение, в соответствии с которым конструкции высокого уровня, состоящие из конструкций низкого уровня, располагаются в верхней части условной схемы. Поэтому рассуждения, проводимые от таких конструкций высокого уровня, как гипотезы, в направлении вниз, к фактам низкого уровня, позволяющим обосновать гипотезы, называются **нисходящими рассуждениями**, или обратным логическим выводом. Понятие обратного логического вывода иллюстрируется на рис. 3.24. Для того чтобы доказать или опровергнуть гипотезу H, необходимо доказать по меньшей мере одну из промежуточных гипотез, H_1,

H_2 или H_3. Обратите внимание на то, что диаграмма представлена в виде дерева AND–OR, поскольку это позволяет показать, что в некоторых случаях, таких как H_2, для обоснования H_2 должны выполняться все гипотезы низкого уровня. А в других случаях, таких как гипотеза верхнего уровня H, необходимо доказать только одну гипотезу низкого уровня. При использовании обратного логического вывода в системе обычно предусматривается возможность запрашивать у пользователя дополнительные свидетельства, способствующие доказательству или опровержению гипотез. В этом состоит принципиальное отличие таких систем от систем прямого логического вывода, в которых все относящиеся к делу факты обычно известны заранее.

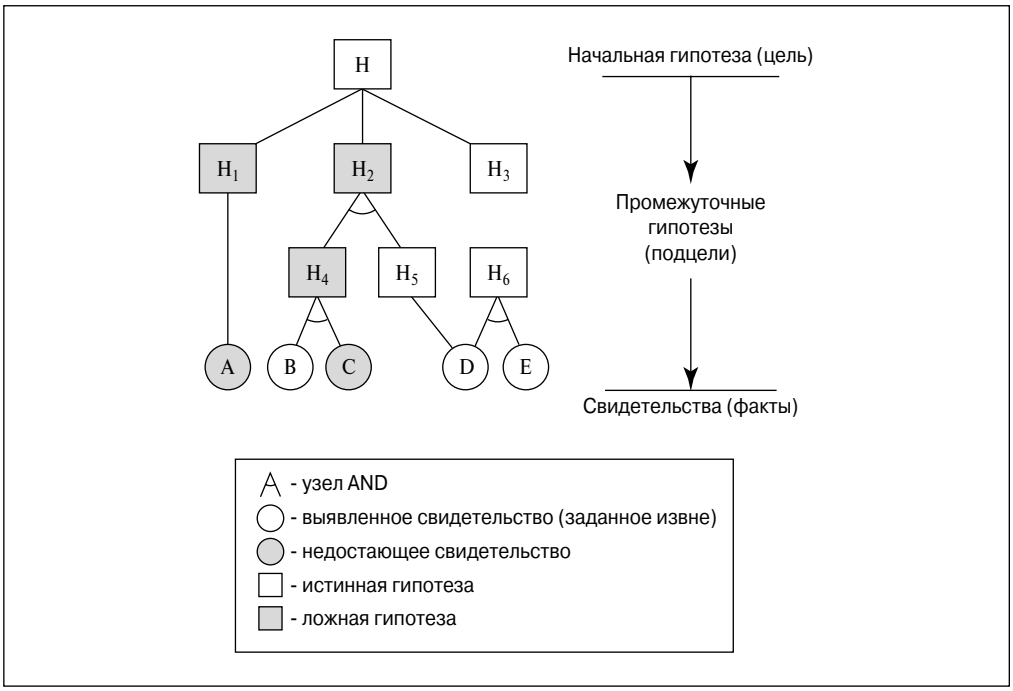


Рис. 3.24. Обратный логический вывод

Еще раз вернувшись к рис. 3.4, приведенному в разделе 3.2, можно заметить, что представленная на рисунке сеть решений организована так, что очень хорошо подходит для обратного логического вывода. Гипотезы верхнего уровня представляют различные типы кустарников малины, таких как душистая малина, западная малина, японская малина и обыкновенная малина. Свидетельства, позволяющие обосновать эти гипотезы, показаны ниже. Могут быть легко сформулированы правила, позволяющие идентифицировать любой из видов кустарников малины. В качестве примера можно привести следующее правило:

```
IF листья - простые THEN душистая малина
```

Одним из важных аспектов выявления необходимых дополнительных свидетельств является способность задавать правильные вопросы. Правильными называются такие вопросы, которые способствуют повышению эффективности процесса определения правильного ответа. Для решения этой задачи необходимо выполнить одно очевидное требование, согласно которому экспертная система должна задавать вопросы, касающиеся только тех гипотез, которые она пытается доказать. Безусловно, количество вопросов, которые могут быть заданы системой, исчисляются сотнями или тысячами, но на получение свидетельств, позволяющих отвечать на вопросы, все равно приходится затрачивать время и деньги. Кроме того, накопление свидетельств некоторого типа, таких как результаты медицинских анализов, может представлять собой неприятный и даже, возможно, опасный для пациента процесс (и действительно, трудно представить себе, что какой-то пациент будет испытывать удовольствие от проведения медицинских анализов).

В идеальном случае экспертная система должна также позволять пользователю добровольно предлагать дополнительные свидетельства, даже если система их не запрашивает. Предоставление пользователю возможности добровольно вводить дополнительные свидетельства способствует ускорению процесса обратного логического вывода и позволяет сделать систему более удобной для пользователя. Добровольно предоставляемое свидетельство может дать системе возможность пропустить некоторые звенья в причинной цепи или выбрать полностью новый подход. А недостаток рассматриваемого варианта состоит в том, что потребуется создание более сложной программы экспертной системы, поскольку в такой системе необходимо будет предусмотреть возможность пропуска отдельных звеньев цепи.

Схемы организации приложений, наиболее подходящих для использования в них прямого и обратного логического вывода, показаны на рис. 3.25. Для упрощения соответствующие диаграммы представлены в виде деревьев, а не сетей общего вида. Наиболее подходящим для прямого логического вывода является такое приложение, схема организации которого имеет вид широкого и не очень глубокого дерева. Это связано с тем, что прямой логический вывод способствует применению поиска в ширину. Это означает, что прямой логический вывод является удобным, если поиск заключений выполняется путем последовательного перехода с одного уровня на другой. В отличие от этого, обратный логический вывод способствует применению поиска в глубину. А дерево, наиболее подходящее для поиска в глубину, является узким и глубоким.

Обратите внимание на то, что выбор наиболее подходящего способа поиска решения зависит от структуры правил. Это означает, что эффективность процесса активизации правил зависит от того, какие шаблоны будут применяться для согласования в проектируемом правиле. Шаблоны в левой части правила позво-

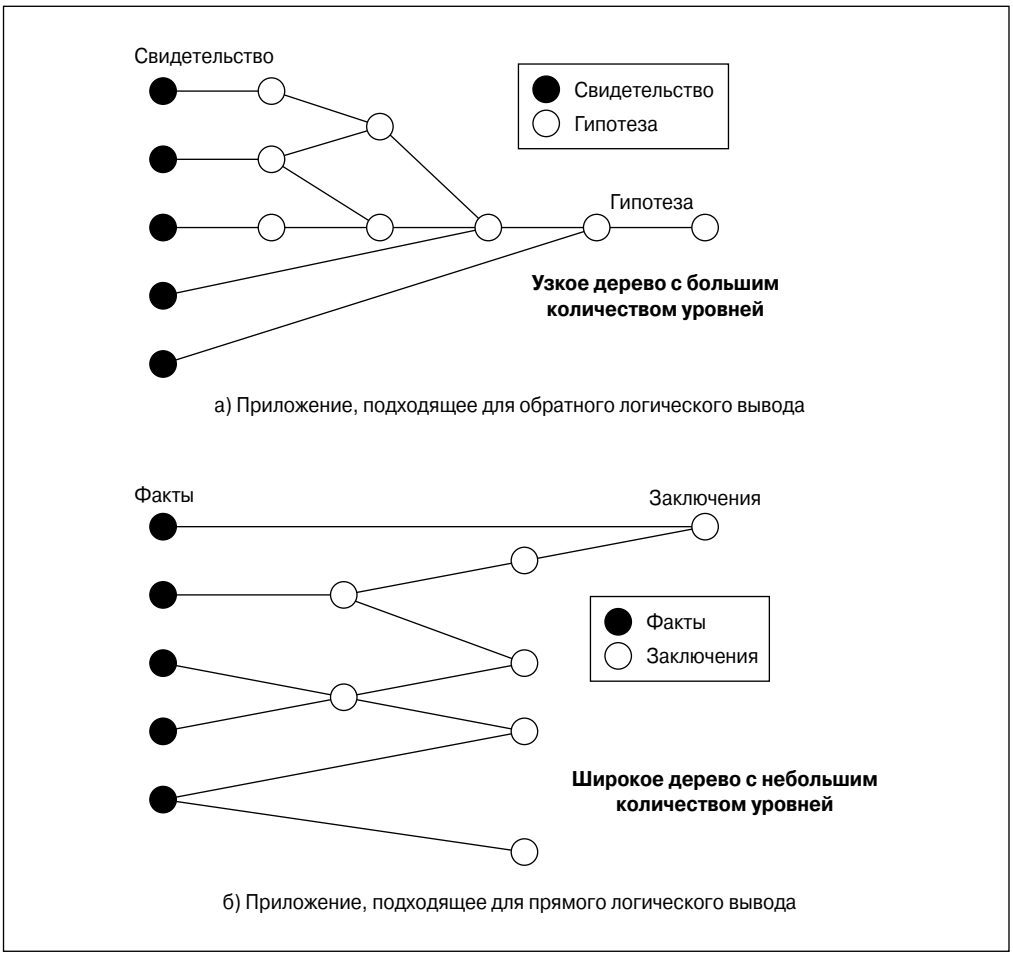


Рис. 3.25. Прямой и обратный логический вывод

ляют определить, может ли правило стать активизированным с помощью фактов. А действия, заданные в правой части правила, определяют, какие факты должны быть введены в базу знаний и удалены из нее, и поэтому влияют на другие правила. Аналогичная ситуация имеет место применительно к обратному логическому выводу, не считая того, что вместо правил используются гипотезы. Безусловно, любая промежуточная гипотеза может представлять собой какое-то правило, при согласовании с которым применяется консеквент, а не антецедент.

В качестве очень простого примера рассмотрим следующие правила типа IF-THEN:

```
IF A THEN B
IF B THEN C
```

```
IF C THEN D
```

Если дан факт A и машина логического вывода спроектирована с учетом согласования фактов с антецедентами, то в базу знаний будут введены промежуточные факты B и C, а также заключение D. Этот процесс соответствует прямому логическому выводу. Такая машина логического вывода представляет собой основу экспертной системы.

В отличие от этого, если в базу знаний вводится факт D (который в действительности представляет собой гипотезу), а машина логического вывода согласует факты с консеквентами правил, то полученный результат соответствует обратному логическому выводу. В системах, предназначенных для обратного логического вывода, таких как PROLOG, механизм обратного логического вывода включает широкий набор средств, позволяющих упростить построение обратной цепи, таких как автоматический перебор с возвратами.

Обратный логический вывод можно осуществить в системе прямого логического вывода (и наоборот) путем перепроектирования правил. Например, приведенные выше правила для прямого логического вывода можно перезаписать следующим образом:

```
IF D THEN C
IF C THEN B
IF B THEN A
```

Теперь факты C и B рассматриваются как подцели или промежуточные гипотезы, которые должны быть выполнены для выполнения гипотезы D. С другой стороны, свидетельство A рассматривается как факт, который обозначает конец процесса выработки подцелей. Если факт A существует, то гипотеза D имеет основание и рассматривается согласно этой цепи обратного логического вывода как истинная. Если же факт A не существует, то гипотеза D не имеет основания и рассматривается как ложная.

Одним из затруднений, связанных с использованием данного подхода, является обеспечение эффективности. Система обратного логического вывода способствует применению поиска в глубину, а система прямого логического вывода — поиска в ширину. Безусловно, приложение, требующее обратного логического вывода, можно написать в системе прямого логического вывода и наоборот, но эта система не будет показывать такую эффективность при поиске решения, как более подходящая для этого система. А вторая сложность является концептуальной. Она связана с тем, что знания, выявленные с помощью эксперта, нужно будет модифицировать таким образом, чтобы они соответствовали требованиям машины логического вывода. Например, машина прямого логического вывода согласовывает факты с антецедентами правил, а машина обратного логического вывода согласовывает факты с консеквентами. Это означает, что если знания эксперта естественным образом подходят для построения обратной цепи, то их придется

полностью реструктурировать, чтобы привести к форме, подходящей для использования в режиме прямого логического вывода, и наоборот.

3.15 Другие методы логического вывода

В экспертных системах иногда используются некоторые другие типы логического вывода, Безусловно, эти методы не являются столь же универсальными, как дедукция, но иногда бывают очень полезными.

Аналогия

Еще одним мощным методом логического вывода, кроме дедукции и индукции, является **аналогия**. Основная идея рассуждений по аналогии состоит в том, чтобы попытаться связать ранее встретившиеся ситуации с новой ситуацией для использования ранее достигнутых результатов в качестве руководства. В процессе своего существования все живые создания проявляют весьма хорошие способности к применению рассуждений по аналогии; эти способности являются крайне важными, поскольку количество новых ситуаций, с которыми им приходится сталкиваться в реальном мире, является просто поразительным. Но вместо того, чтобы трактовать каждую новую ситуацию как уникальную, часто бывает полезно попытаться найти аналогии между новой ситуацией и какими-либо старыми, в отношении которых известно, как действовать в этих ситуациях. Рассуждения по аналогии тесно связаны с индуктивными рассуждениями. Но индукция предусматривает проведение в определенной ситуации логического вывода от частного к общему, а проведение аналогии в той же ситуации связано с попыткой формирования логических выводов с учетом ситуаций, которые не являются точно такими же, как новая. Рассуждения по аналогии не позволяют получать такие формальные доказательства, которые создаются с помощью дедукции. Вместо этого рассуждения по аналогии представляют собой эвристическое инструментальное средство формирования рассуждений, которое иногда позволяет выполнить необходимую работу. Вообще говоря, такие рассуждения служат основным инструментальным средством формирования рассуждений на основе прецедентов в судебных доказательствах и медицинской диагностике.

Одним из примеров рассуждений по аналогии является медицинская диагностика. После того как пациент приходит на прием к врачу, обнаружив какие-то проблемы со здоровьем, врач получает от него необходимую информацию и регистрирует симптомы наблюдаемой проблемы. Если эти симптомы идентичны или в значительной степени аналогичны наблюдавшимся у других людей с заболеванием X, врач может вывести по аналогии, что поступивший к нему пациент имеет заболевание X. Рассуждения по аналогии — это наименее дорогостоящий способ формирования рассуждений.

Обратите внимание на то, что полученный таким путем диагноз не является результатом дедуктивного вывода, поскольку каждый пациент является уникальным. Лишь то, что осмотр какого-то другого пациента с таким же заболеванием позволяет выявить некоторые симптомы, отнюдь не означает, что данный пациент с тем же заболеванием покажет наличие у него таких же симптомов. Вместо этого врач принимает предположение, что обнаруженные у текущего пациента симптомы позволяют провести аналогию между ним и лицом с подобными симптомами и с известным заболеванием. Этот начальный диагноз представляет собой гипотезу, которая может быть либо доказана, либо опровергнута с помощью медицинских анализов. Но крайне важно иметь начальную рабочую гипотезу, поскольку именно она позволяет свести весь перечень, состоящий из тысяч потенциальных заболеваний, к одному или нескольким заболеваниям. Если бы врач решил начать свое обследование с выписки направлений на все возможные медицинские анализы, не выдвинув начальной гипотезы, то такое решение было бы просто невыполнимым из-за слишком высоких затрат денег и времени.

В качестве примера того, какими полезными могут быть рассуждения по аналогии, предположим, что два человека ведут между собой игру, называемую игрой в 15. Эти игроки по очереди выбирают число от 1 до 9, руководствуясь таким ограничением, что одно и то же число не может использоваться дважды. Побеждает тот, кто первым сумеет получить сумму выбранных цифр, равную 15. Безусловно, на первый взгляд может показаться, что это — очень сложная игра, для победы в которой необходимо хорошо подумать, но благодаря аналогии ее можно превратить в игру, ведение которой становится очень несложным.

Рассмотрим следующую приведенную ниже доску для игры в крестики-нолики, в которой в каждую клетку вписаны числа, как показано схематически на доске.

6 1 8
7 5 3
2 9 4

Это — типичный **магический квадрат**, поскольку в нем сумма значений по строкам, столбцам и диагоналям является постоянной. Доску для игры в крестики-нолики, заполненную значениями чисел из магического квадрата, можно рассматривать как аналогию для игры в 15. После того как игрок начнет рассматривать игру в 15 в терминах игры в крестики-нолики и применит показанную стратегию выигрыша к игре в 15, ведение последней становится очень простым.

Данный конкретный магический квадрат называется **стандартным квадратом третьего порядка**. Термин *порядок* обозначает количество строк или столбцов в квадрате. Существует только один уникальный квадрат третьего порядка. Путем вращения или создания зеркального отображения по отношению к стандартному квадрату могут быть созданы другие магические квадраты. Еще один

способ создания магического квадрата на основе других магических квадратов состоит в добавлении одной и той же числовой константы к значению в каждой клетке исходного квадрата. Знание этой информации позволяет также логическим путем определить стратегии выигрыша для игры в 18, в которой числа должны выбираться из следующего множества:

$$\{2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

или для игры в 21, в которой используется такое множество:

$$\{3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

Теперь мы можем воспользоваться индукцией для вывода логическим путем стратегии выигрыша для игры в $15 + 3N$, где N — любое натуральное число $1, 2, 3, \ldots$, рассматривая эту игру как аналогичную игре на доске для игры в крестики-нолики, которая, в свою очередь, связана аналогией с магический квадратом, состоящим из следующих значений:

$$\{1 + N, 2 + N, 3 + N, 4 + N, 5 + N, 6 + N, 7 + N, 8 + N, 9 + N\}$$

Используя аналогию, согласно которой квадрат третьего порядка можно применять для ведения игр с тремя ходами, по индукции можно прийти к логическому выводу, что для ведения игр, в ходе которых должно быть сделано больше трех ходов, можно использовать магические квадраты более высокого порядка. Например, приведенный ниже магический квадрат четвертого порядка 4 позволяет найти стратегию выигрыша для четырехходовой игры в 34, рассматривая эту игру в терминах игры в крестики-нолики.

$$\begin{array}{rrrr} 16 & 3 & 2 & 13 \\ 5 & 10 & 11 & 8 \\ 9 & 6 & 7 & 12 \\ 4 & 15 & 14 & 1 \end{array}$$

Но в отличие от того, что количество стандартных квадратов третьего порядка равно 1, количество стандартных квадратов четвертого порядка равно 880, а это позволяет вести значительно больше непохожих друг на друга игр.

Рассуждения по аналогии составляют важную часть рассуждений на основе здравого смысла, а задача проведения таких рассуждений является очень сложной для компьютеров (и детей). Другими приложениями, в которых широко применяются рассуждения по аналогии, стали учебные программы.

Метод формирования и проверки

Еще одним методом логического вывода является классическая стратегия **формирования и проверки**, применяемая в искусственном интеллекте, которую иногда называют выработкой и проверкой. Применение данного метода предусматривает формирование вероятного решения и последующую его проверку для определения того, соответствует ли предложенное решение всем требованиям. Если решение является удовлетворительным, работа прекращается, в противном случае формируется новое решение, которое опять проверяется, и т.д. Этот метод использовался в первой экспертной системе, DENDRAL, проект которой был задуман в 1965 году для содействия в определении структур органических молекул. С помощью массового спектрометра подготавливались данные, полученные в результате обработки образца неизвестного органического вещества, и вводились в систему DENDRAL, которая вырабатывала гипотезы, касающиеся всех потенциальных молекулярных структур, обработка которых в массовом спектрометре могла привести к получению рассматриваемой спектрограммы неизвестного вещества. После этого система DENDRAL проверяла молекулы, являющиеся наиболее вероятными кандидатами, моделируя их масс-спектрограммы и сравнивая полученные результаты с первоначальной спектрограммой неизвестного вещества. Еще одной программой, в которой используется метод формирования и проверки, является программа Artificial Mathematician, применявшаяся для вывода новых математических понятий.

Чаще всего количество потенциальных решений чрезвычайно велико. Для сокращения общего количества проверяемых решения метод формирования и проверки обычно используется в сочетании с программой планирования, позволяющей лимитировать количество подлежащих формированию потенциальных решений. Этот вариант рассматриваемого метода называется **методом планирования, формирования и проверки** и применяется в целях повышения эффективности во многих системах. Например, в медицинской диагностической экспертной системе MYCIN предусмотрена также возможность планирования терапевтического медикаментозного лечения после получения диагноза заболевания пациента. Составление **плана** по существу сводится к поиску цепей правил или этапов логического вывода, которые соединяют задачу с решением или цель со свидетельством, которое ее обосновывает. Планирование осуществляется наиболее эффективно путем одновременного поиска в прямом направлении, от фактов, и в обратном направлении, от цели.

В планировщике MYCIN вначале создается распределенный по приоритетам список терапевтических медикаментов, к которым чувствителен пациент. При этом для уменьшения нежелательных взаимодействий различных лекарственных веществ в организме пациента лучше всего ограничить количество разных принимаемых медикаментов, даже если есть основания полагать, что пациент страдает

от заболеваний, вызванных различными инфекциями. Затем планировщик передает организованный по приоритетам список генератору, который по возможности формирует подсписки, состоящие из одного или двух медикаментов. После этого подсписки проверяются с учетом эффективности их действия против рассматриваемых инфекций, аллергических реакций пациента и других соображений. Наконец, вырабатывается решение о том, какие медикаменты следует назначить пациенту.

Метод формирования и проверки может также рассматриваться как основной подход к логическому выводу правил. Если выполняются условные элементы какого-то правила, с помощью этого метода вырабатываются некоторые действия, такие как новые факты. Машина логического вывода проверяет полученные факты по условным элементам правил, находящихся в базе знаний. Затем правила, нашедшие свои соответствия, помещаются в рабочий список правил, с помощью правила с наивысшим приоритетом вырабатываются его действия, которые затем проверяются, и т.д. Таким образом, метод формирования и проверки позволяет создать цепь вывода, которая может привести к правильному решению.

Абдукция

Еще одним методом, широко используемым при решении задач диагностики, является логический вывод по абдукции. Схема абдукции внешне напоминает схему правила модус поненс, но фактически имеет весьма значительные отличия, как показано в табл. 3.16.

Таблица 3.16. Сравнение абдукции и правила модус поненс

Абдукция	Правило модус поненс
$p \rightarrow q$	$p \rightarrow q$
q	p
$\therefore p$	$\therefore q$

Абдукция — это еще одно название для ошибочного метода доказательства, который был описан в разделе 3.6 под названием “ошибочное обращение посылки и заключения”. Безусловно, абдукция не представляет собой действительное дедуктивное доказательство, но может служить удобным методом логического вывода, поэтому используется в экспертных системах. Как и аналогия, которая также не позволяет формировать действительное дедуктивное доказательство, но является ускоренным и недорогим методом (с точки зрения затрат времени на проведение рассуждений, а не денег), абдукция может оказаться применимой в качестве эвристического правила логического вывода. Таким образом, если не может применяться дедуктивный метод логического вывода, то может оказаться полезной абдукция, но получение с ее помощью качественных результатов не гарантирует-

ся. Все методы, подобные аналогии, формированию и проверке, а также абдукции, не являются дедуктивными, поэтому не гарантируют получение нужных результатов при любых обстоятельствах. Даже при наличии истинных посылок эти методы не позволяют доказывать истинные заключения. Тем не менее с помощью указанных методов появляется возможность сократить пространство поиска путем выработки приемлемых гипотез, которые затем могут использоваться в сочетании с дедукцией.

Абдукцию иногда называют рассуждениями от наблюдаемых фактов к наилучшему объяснению. В качестве примера применения абдукции рассмотрим следующие рассуждения:

```
IF x - слон THEN x - животное
IF x - животное THEN x - млекопитающее
```

Если известно, что Клайд — млекопитающее, можно ли сделать вывод, что Клайд — слон?

Ответ на этот вопрос зависит от того, идет ли речь о реальном мире или о некоторой экспертной системе. В реальном мире указанное заключение невозможно сделать с какой-либо допустимой степенью достоверности. Клайд может оказаться собакой, кошкой, коровой или животным любого другого вида, которое является млекопитающим, но не слоном. В действительности, учитывая то, как много в мире разновидностей животных, можно понять: если больше нет какой-либо другой информации о Клайде, вероятность того, что Клайд — слон, довольно низка.

Но если речь идет об экспертной системе, содержащей перечисленные выше правила, то можно сделать вывод с помощью абдукции со 100%-ной уверенностью: если Клайд — млекопитающее, то Клайд — слон. Этот вывод следует из предположения о замкнутости мира, согласно которому предполагается, что за пределами замкнутого мира рассматриваемой экспертной системы больше ничего не существует. А в замкнутом мире все, что не может быть доказано, считается ложным. В системе, действующей согласно предположению о замкнутости мира, известны все возможности. А поскольку база знаний экспертной системы состоит только из двух приведенных правил и только слон может быть млекопитающим, то Клайд, если он — млекопитающее, должен быть слоном.

Предположим, что в эту экспертную систему будет введено такое третье правило:

```
IF x - собака THEN x - животное
```

В результате этого по-прежнему сохранится возможность эксплуатировать рассматриваемую экспертную систему в соответствии с предположением о замкнутости мира. Но теперь нельзя будет сделать вывод с достоверностью на 100%, что Клайд — слон. Единственное, в чем можно быть уверенным, — то, что Клайд либо слон, либо собака.

Для того чтобы иметь возможность выбрать одну из этих двух гипотез, необходимо получить дополнительную информацию. Например, если в системе есть еще одно правило:

```
IF x - собака THEN x лает
```

и получены свидетельства, согласно которым Клайд лает, то рассматриваемые правила можно модифицировать следующим образом:

```
(1) IF x - животное THEN x - млекопитающее
(2) IF x лает THEN x - животное
(3) IF x - собака THEN x лает
(4) IF x - слон THEN x - животное
```

После этого появляется возможность составить обратную цепь абдуктивного логического вывода с использованием правил (1)–(3) и показать, что Клайд должен быть собакой.

Обратная цепь абдукции не представляет собой то, что принято рассматривать как цепь, сформированную с помощью метода обратного логического вывода (метода построения обратной цепи). Термин *обратный логический вывод* означает, что в процессе создания обратной цепи предпринимается попытка доказать гипотезу, отыскивая свидетельства, которые обосновывают эту гипотезу. Обратный логический вывод действительно можно было бы использовать для доказательства того, что Клайд — млекопитающее. Безусловно, в этой небольшой системе не рассматриваются животные других видов. Но ничто не препятствует возможности введения других классификаций, относящихся к рептилиям, птицам, и т.д. Если известно, что Клайд — млекопитающее, абдукция может использоваться для определения того, является ли Клайд слоном или собакой. А если бы было известно, что Клайд — слон, и потребовалось бы узнать, является ли он млекопитающим, то можно было воспользоваться методом прямого логического вывода. Итак, очевидно, выбор метода логического вывода зависит от того, что должно быть определено. Но поскольку дедуктивным является только прямой логический вывод, всегда гарантируется правильность лишь тех заключений, которые получены с его помощью. Итоговые данные о каждом из трех рассматриваемых методов логического вывода приведены в табл. 3.17.

Во многих системах искусственного интеллекта и в экспертных системах абдукция на основе фреймов используется для решения задач диагностики. В таких системах база знаний содержит **причинные ассоциации** между нарушениями в работе и их симптомами. А логический вывод осуществляется на основе формирования и проверки гипотез, объясняющих причины нарушений в работе.

Таблица 3.17. Итоговые сведения о назначении методов прямого логического вывода, обратного логического вывода и абдукции

Логический вывод	Исходная ситуация	Целевая ситуация
Прямой логический вывод	Факты	Заключения, которые должны следовать из фактов
Обратный логический вывод	Неопределенное заключение	Факты, обосновывающие заключение
Абдукция	Истинное заключение	Факты, которые могут следовать из заключения

Немонотонный вывод

Обычно введение новых аксиом в логическую систему приводит к тому, что появляется возможность доказывать больше теорем, поскольку при этом увеличивается количество аксиом, на основании которых могут быть выведены теоремы. Такое свойство увеличения количества доказуемых теорем с увеличением количества аксиом называется **монотонностью**, а системы, обладающие этим свойством, такие как системы дедуктивной логики, называются **монотонными системами**.

Но если вновь введенная аксиома частично или полностью противоречит одной из ранее заданных аксиом, могут возникнуть проблемы. В таком случае могут стать больше не действительными уже доказанные теоремы. Таким образом, в **немонотонной системе** с увеличением количества аксиом не обязательно увеличивается количество теорем.

Понятие немонотонности имеет важное значение для экспертных систем. По мере выработки новых фактов (а этот процесс аналогичен доказательству теорем) монотонная экспертная система продолжает накапливать факты. Если же один или несколько фактов становятся ложными, то может возникнуть серьезная проблема, поскольку монотонная система неспособна справиться с такими ситуациями, в которых происходят изменения истинностного значения аксиом и теорем. В качестве очень простого примера предположим, что в системе имеется факт, регистрирующий значение времени в секундах. Но, как только время изменяется на одну секунду, этот факт становится устаревшим и больше не является действительным. Монотонная система не обладает способностью справиться с подобной ситуацией. В качестве еще одного примера рассмотрим случай, в котором в экспертной системе регистрируется факт, полученный от системы идентификации самолетов, что некоторая отметка на экране радиолокатора относится к нарушителю воздушного пространства (чужой самолет), а в дальнейшем появляются новые свидетельства, которые показывают, что под этой отметкой скрывается свой самолет. В монотонной системе невозможно изменить первоначальную идентификацию результатов радиолокационного наблюдения как относящуюся к чужому самолету. С другой

стороны, немонотонная система допускает возможность удаления фактов из базы знаний.

Рассмотрим еще один пример. Предположим, что требуется создать средство объяснения для экспертной системы, которое позволяло бы пользователю возвращаться к предыдущим этапам логического вывода и исследовать альтернативные пути логического вывода, задавая системе вопросы типа “что, если”. При этом результаты всех этапов логического вывода, проведенных после рассматриваемого предыдущего этапа, должны быть изъяты из системы. Для этого может потребоваться удалить из базы знаний системы не только факты, но и правила; таким образом, используемая база знаний, позволяющая удалять (или резервировать) ранее полученные результаты, должна обладать свойством немонотонности. В таких системах, как OPS5, в которых правила могут создаваться автоматически во время выполнения программы с использованием правых частей активизируемых правил, возникают дополнительные сложности. Дело в том, что в таких системах для обеспечения немонотонности все выведенные правила, которые были созданы после применения всех этапов, находящихся за тем этапом логического вывода, к которому желает вернуться пользователь, также потребуется удалить из системы. Для этой цели необходимо сохранять информацию обо всех выполненных этапах логического вывода, для чего требуется большой объем памяти, а работа системы существенно замедляется.

Для обеспечения немонотонности необходимо закрепить обоснование за каждым фактом и правилом или указать, чем было вызвано его появление; такое обоснование позволяет объяснить, по каким причинам факт или правило считается достоверным. В таком случае в процессе выработки немонотонного решения машина логического вывода может проверять обоснование каждого факта и правила для определения того, считается ли он все еще достоверным. При этом появляется также возможность восстанавливать удаленные правила и изъятые факты, если есть основание снова считать их достоверными.

На проблему обоснования фактов впервые обратили внимание ученые, занимавшиеся решением **проблемы окружения** (frame problem). Слово frame в англоязычном варианте этого термина не относится к понятию фреймов, которое рассматривалось в главе 2. Термин frame problem является описательным и был сформулирован в результате изучения проблемы определения того, что изменилось или не изменилось в очередном кадре кинофильма (movie frame). Киноленты представляют собой последовательности неподвижных фотографических изображений, называемых кадрами. При воспроизведении кинолент с частотой 24 кадра в секунду или с более высокой частотой человеческий глаз теряет способность различать отдельные кадры, и поэтому при наличии последовательных изменений в кадрах возникает иллюзия движения. А в искусственном интеллекте проблема окружения рассматривается как задача распознавания того, что изменяется со временем в рассматриваемой среде и что остается неизменным. В качестве примера

рассмотрим описанную выше задачу с обезьяной и бананами. Предположим, что обезьяна, чтобы достать бананы, должна встать на ящик красного цвета, поэтому очередным действием, которое она должна выполнить, является “пододвинуть красный ящик под бананы”. В таком случае проблема окружения сводится к следующему — как определить, остался ли ящик после выполнения указанного действия по-прежнему красным? Очевидно, что в результате перемещения ящика рассматриваемая среда не должна измениться в такой форме. Но к указанному изменению среды могут привести другие действия, такие как “покрасить ящик в синий цвет”. В некоторых инструментальных средствах экспертных систем принято называть рассматриваемую среду **миром** и считать, что мир определяется множеством взаимосвязанных фактов. В экспертной системе может быть предусмотрена возможность слежения за многочисленными мирами и одновременного проведения гипотетических рассуждений. Проблема поддержания правильности, или истинности, системы называется проблемой **поддержания достоверности**. Средства поддержания достоверности и разновидность этих средств, называемая **поддержанием достоверности на основе предположений**, являются очень важными, поскольку позволяют поддерживать каждый мир в состоянии, свободном от противоречий, изымая необоснованные факты.

В качестве простого примера немонотонного вывода рассмотрим классический пример с птицей Твити. В отсутствии какой-либо другой информации можно полагать, что Твити обладает способностью летать, поскольку Твити — птица. Это — пример **рассуждений по умолчанию**, которые во многом напоминают операции обработки слотов фреймов с использованием значений по умолчанию. Рассуждения по умолчанию могут рассматриваться как применение правила, в котором выполняется логический вывод в отношении правил, **метаправило**, т.е. как некоторое правило, в котором указано:

```
IF X достоверно не известно и
   нет свидетельств, противоречащих X
THEN сделать предварительное заключение Y
```

Метаправила обсуждаются более подробно в следующем разделе.

В данном случае метаправило имеет следующую более конкретную форму:

```
X - правило "Все птицы обладают способностью летать" и
   есть факт "Твити - птица"
Y - логический вывод "Твити обладает способностью летать"
```

Оно может быть выражено в терминах продукционных правил как правило в базе знаний, которое указывает следующее:

```
IF X - птица THEN X обладает способностью летать
```

и факт, существующий в рабочей памяти:

```
Твити - птица
```

Унификация данного факта с антецедентом данного правила приводит к получению логического вывода, согласно которому Твити обладает способностью летать.

Но в данной ситуации обнаруживаются контуры проблемы. Предположим, что в рабочую память вводится дополнительный факт, в котором утверждается, что Твити — пингвин. Известно, что пингвины не обладают способностью летать, поэтому логический вывод, согласно которому Твити обладает способностью летать, является неправильным. Безусловно, что в системе должно быть также правило, в котором формулируются соответствующие знания, так как в противном случае указанный факт будет проигнорирован.

Для поддержания правильности системы необходимо удалять неправильные результаты логического вывода. Но этого может оказаться недостаточным, если на неправильных результатах логического вывода были основаны другие этапы логического вывода. Это означает, что в других правилах могли быть использованы неправильные результаты логического вывода в качестве свидетельств для выполнения дополнительных этапов логического вывода, и т.д. Это — еще одна из проблем поддержания достоверности. Логический вывод, согласно которому Твити обладает способностью летать, оказался **правдоподобным логическим выводом**, основанном на рассуждениях по умолчанию. Термин *правдоподобный* означает не невозможный и рассматривается более подробно в главе 4.

Один из способов обеспечения немонотонного логического вывода состоит в определении сентенциального предиката операции M, который может быть неформально описан как “является совместимым”. Например, приведенное ниже выражение можно сформулировать таким образом: “Для каждого x, если x — птица, и этот факт является совместимым с тем утверждением, что птицы обладают способностью летать, то x обладает способностью летать”.

$$(\forall x)[\text{Bird}(x) \wedge M(\text{Can_fly}(x)) \rightarrow \text{Can_fly}(x)]$$

Более неформальный способ трактовки этого выражения состоит в том, что “Большинство птиц обладают способностью летать”. В данном случае термин *совместимый* означает, что совместимый факт не вступает в противоречие с другими знаниями. Но указанная интерпретация была подвергнута критике, поскольку фактически она сводится к утверждению, что единственными птицами, которые не обладают способностью летать, являются те птицы, для которых доказано, что они не обладают способностью летать. В этом состоит пример **аутоэпистемических рассуждений**, которые буквально означают рассуждения о своих собственных знаниях. И рассуждения по умолчанию, и аутоэпистемические рассуждения используются в **рассуждениях на основе здравого смысла**, которые обычно весьма успешно осуществляются людьми, но являются очень сложными для компьютеров.

Аутоэпистемическими называются рассуждения о собственных знаниях, в отличие от рассуждений о знаниях в целом. Вообще говоря, люди способны проводить аутоэпистемические рассуждения очень успешно, поскольку знают пределы своих знаний. Например, предположим, что к вам подошло абсолютно неизвестное лицо и заявило, что вы находитесь с ним в супружеских отношениях. Любой человек (если он не страдает амнезией) сразу же определит, что у него нет никаких супружеских отношений с этим неизвестным лицом, поскольку о нем нет никаких знаний. Общее метаправило аутоэпистемического рассуждения состоит в следующем:

```
IF я не обладаю какими-либо знаниями об X
THEN X - ложный факт
```

Обратите внимание на то, насколько существенно в аутоэпистемических рассуждениях используется предположение о замкнутости мира. Любой неизвестный факт рассматривается как ложный. В аутоэпистемических рассуждениях замкнутым миром являются знания человека о самом себе.

И аутоэпистемические рассуждения, и рассуждения по умолчанию являются немонотонными. Но причины такого положения разные. Рассуждения по умолчанию немонотонны, поскольку являются **отменяемыми**. Термин *отменяемый* означает, что любые логические выводы являются предварительными и могут быть отменены после того, как станет доступной новая информация. Но чисто аутоэпистемические рассуждения нельзя считать отменяемыми из-за наличия предположения о замкнутости мира, согласно которому человеку, проводящему аутоэпистемические рассуждения, уже известны все истинные знания. Например, человек, находящийся в браке, хорошо знает, с кем он состоит в супружеских отношениях (если только не горит желанием об этом забыть), поэтому не признает, что состоит в браке с полностью незнакомым лицом, даже если ему об этом заявят с полной уверенностью. Тем не менее большинство людей признают, что память у них не идеальна, поэтому не полагаются исключительно на аутоэпистемические рассуждения. Безусловно, компьютеры не страдают от этой проблемы до тех пор, пока не происходит отказ жесткого диска.

Аутоэпистемические рассуждения являются немонотонными, поскольку смысл аутоэпистемического оператора является **контекстно зависимым**. Термин *контекстно зависимый* обозначает имеющий смысл, который изменяется вместе с контекстом. В качестве простого примера контекстной зависимости рассмотрим, как произносится слово “read” в следующих двух предложениях:

```
I have read the book
I will read the book
```

Очевидно, что произношение слова “read” зависит от контекста.

Теперь рассмотрим систему, состоящую из следующих двух аксиом:

$$(\forall x)[\text{Bird}(x) \wedge M(\text{Can_fly}(x)) \rightarrow \text{Can_fly}(x)]$$
$$\text{Bird}(\text{Tweety})$$

В этой логической системе выражение Can_fly(Tweety) представляет собой теорему, выведенную путем унификации константы Tweety с переменной x и последующей импликации.

После этого примем предположение, что добавлена новая аксиома, которая указывает, что Твити не обладает способностью летать, и поэтому противоречит ранее выведенной теореме:

$$\sim\text{Can_fly}(\text{Tweety})$$

Использование предиката M должно привести к изменению результатов его применения к тому же параметру, поскольку теперь значение $M(\text{Can_fly}(\text{Tweety}))$ не совместимо с новой аксиомой. В этом новом контексте, состоящем из трех аксиом, применение предиката M не должно приводить к получению результата `TRUE` для Can_fly(Tweety), поскольку при этом возникает конфликт с новой аксиомой. В данном новом контексте возвращенное значение предиката M должно быть равно `FALSE`, и поэтому конъюнкция также имеет значение `FALSE`. Таким образом, операция импликации, с помощью которой должна быть выведена теорема Can_fly(Tweety), не выполняется и конфликт не возникает.

Один из способов реализации такого замысла с помощью правил показан ниже.

```
IF x - птица AND x - типичная птица
THEN x обладает способностью летать
IF x - птица AND x - нетипичная птица
THEN x не обладает способностью летать
Твити - птица
Твити - нетипичная птица
```

Обратите внимание на то, что в этой системе не объявляется недействительным заключение Can_fly(Tweety), а скорее вообще предотвращается запуск неверного правила. Это — гораздо более эффективный способ поддержания достоверности по сравнению с тем способом, в котором применялось одно правило и специальная аксиома $\sim$Can_fly(Tweety). Таким образом, создается гораздо более общая система, позволяющая легко справляться с ситуациями, в которых обнаруживаются другие птицы, неспособные летать, такие как страусы, без необходимости непрерывно вводить новые этапы логического вывода, предназначенные для выполнения системой.

3.16 Метазнания

В классической программе Meta-DENDRAL для вывода новых правил, касающихся химической структуры, используется индукция. Создание системы Meta-DENDRAL явилось попыткой преодолеть узкое место в процессе приобретения знаний, которое обнаружилось при проведении работ по выявлению знаний о правилах определения молекулярной структуры, которыми обладают эксперты-люди. А теперь программы логического вывода правил входят в состав некоторых инструментальных средств экспертных систем.

Например, следующее классическое метаправило взято из программы приобретения знаний TEIRESIAS для экспертной системы MYCIN, которая предназначена для диагностирования инфекционных заболеваний кровеносной системы и менингита:

```
METARULE 2
IF
The patient is a compromised host, and
There are rules which mention in their premise
   pseudomonas, and
There are rules which mention in their premise
   klebsiellas
THEN
There is suggestive evidence (0.4) that the former
   should be done before the latter
```

Число `0.4` в части правила, соответствующей действию, представляет собой степень достоверности и обсуждается в одной из следующих глав.

Программа TEIRESIAS выявляет у эксперта знания в интерактивном режиме. Если системой MYCIN был установлен неправильный диагноз, то программа TEIRESIAS сопровождает эксперта в процессе нахождения в обратном направлении через цепь неправильных рассуждений до тех пор, пока эксперт не укажет, с чего начались неправильные рассуждения. Во время прохождения в обратном направлении через цепь рассуждений программа TEIRESIAS взаимодействует также с экспертом для модификации неверных правил или приобретения новых правил.

Знания о новых правилах не вводятся в систему MYCIN немедленно. Вместо этого программа TEIRESIAS проводит проверку для определения того, совместимо ли новое правило с подобными правилами. Например, если новое правило описывает, как инфекция попадает в организм, а в других принятых на вооружение правилах имеется условный элемент, указывающий конкретный путь проникновения в организм, то программа отмечает, что и в новом правиле должно быть учтено это условие. Если в новом правиле не указан путь проникновения инфекции в организм, программа TEIRESIAS передает запрос пользователю, чтобы он устранил

такое расхождение. В программе TEIRESIAS имеется **шаблон модели правил** для подобных правил, о которых ей известно, поэтому программа предпринимает попытки привести новое правило в соответствие с имеющейся моделью правил. Иными словами, модель правил представляет собой знания о собственных знаниях, которыми обладает программа TEIRESIAS. Ситуация, аналогичная попытке ввести в рассматриваемую систему неверное правило, возникает в обыденной жизни, например, если покупатель приходит к агенту по продаже легковых автомобилей, чтобы купить новый автомобиль, а агент пытается ему продать автомобиль с тремя колесами вместо четырех.

Метазнания системы TEIRESIAS подразделяются на два типа. Выше были описаны метазнания первого типа, применяемые в стратегии управления META-RULE 2, которая позволяет определить, как должны использоваться новые правила. С другой стороны, метазнания второго типа, относящиеся к модели правил, определяют, находится ли новое правило в форме, подходящей для ввода в базу знаний. В экспертной системе, основанной на правилах, определение того, находится ли новое правило в подходящей правильной форме, называется **верификацией** (verification) правила. А определение того, ведет ли цепь правильных этапов логического вывода к правильному ответу, называется **аттестацией** (validation). Задачи аттестации и верификации настолько тесно взаимосвязаны, что для обозначения обеих этих задач широко используется аббревиатура **V&V** (validation and verification). Более описательное определение этих терминов, которое было сформулировано в индустрии разработки программного обеспечения, приведено ниже.

```
Верификация: "Правильно ли я создаю продукт?"
Аттестация: "Создаю ли я правильный продукт?"
```

По существу, верификация имеет отношение к внутреннему аспекту правильности, а аттестация — к внешнему. Тема аттестации и верификации рассматривается более подробно в главе 6.

3.17 Скрытые марковские модели

В настоящее время область применения метазнаний существенно расширилась. В качестве одного из примеров рассмотрим планирование маршрута для робота. Если робот не оборудован средствами глобальной системой позиционирования (Global Positioning System — GPS) или эти средства не являются достаточно точными (что обычно относится к коммерческим системам, которые позволяют определять местонахождение лишь с точностью около 3 метров), то должны применяться другие средства. В качественных приложениях планирования маршрута используется **марковский процесс принятия решений** (**Markov decision pro-**

cess — MDP) [55]. В других методах применяются классический алгоритм A*, фильтры Калмана и другие технологии [64].

В реальном мире всегда возникают неопределенности, а если имеет место неопределенность, то чистая логика не может служить хорошим руководством. В том случае, если имеется лишь частичная или скрытая информация о состоянии и параметрах, а также возникает необходимость в планировании, большие возможности открывает применение марковского процесса принятия решений. Такие процессы не нацеливаются исключительно на физическое планирование маршрута и охватывают любые типы планирования в частично известной среде, такие как разведка месторождений нефти, решение логистических задач транспортировки и управление производственными процессами, в которых могут отказывать датчики и возникать другие нарушения. Подобное управление процессами может оказаться особенно важным, если управляемым процессом является работа атомной электростанции.

Наиболее сложные задачи управления в условиях наличия лишь частичной информации возникают, когда робот исследует поверхность другой планеты, например Марса. Если цель робота состоит в том, чтобы добраться до определенной скалы, но будущий маршрут является наблюдаемым лишь отчасти, то робот должен попытаться выработать оптимальные решения, касающиеся того, как достичь цели, с учетом таких условий, как имеющийся у него ограниченный запас энергии, а также, возможно, неспособность вернуться на правильный маршрут после его попадания в глубокий кратер.

Марковский процесс принятия решений (MDP) можно определить как кортеж $\{\text{States}, \text{Actions}, \text{Transitions}, \text{Rewards}\}$ ($\{\text{Состояния}, \text{Действия}, \text{Переходы}, \text{Вознаграждения}\}$). Более формально определение марковского процесса принятия решений можно записать как $\text{MDP} \equiv \{S, A, T, R\}$. В этом выражении элементы кортежа определены следующим образом.

- S — множество состояний среды.
- A — множество аксиом.
- $T : SxA \equiv \prod(S)$ — множество переходов. В этом выражении $\prod$ именуется **функцией перехода между состояниями** и определяет, к какому состоянию происходит переход при выполнении каждого конкретного действия; "x" — операция декартового произведения. Итак, функция $\prod$ определяет множество всех возможных состояний и действий. Очевидно, что некоторые из состояний и действий могут оказаться неосуществимыми или нежелательными, но сама операция декартова произведения не позволяет выделить лишь те состояния и действия, которые являются оптимальными. Для определения того, какие действия приводят к наилучшим состояниям и, например, не позволяют роботу зайти в тупик, необходимо учитывать вознаграждения.

- $R : SxA \rightarrow R(S)$ — множество вознаграждений. В этом выражении $\boldsymbol{R}$ — **функция вознаграждения**, которая обеспечивает немедленное получение **агентом** вознаграждения за выполнение определенного действия. Термин *агент* используется для обозначения любой сущности, которая действует в интересах других. Таким образом, рассматриваемый робот — агент человека на другой планете. Другие состояния связаны с другими ожидаемыми вознаграждениями. При осуществлении поиска оптимального пути к цели одним из способов определения оптимума может служить максимизация ожидаемой суммы вознаграждений. Немедленное вознаграждение предоставляется сразу же, но лишь сумма вознаграждений позволяет определить, достигнет ли агент цели или потерпит неудачу. Например, допустим, что если робот сумеет повернуть в нужном месте, то сумеет найти путь, свободный от препятствий. К сожалению, не исключена возможность, что этот открытый путь не приведет к цели, которая заключается в том, чтобы добраться до нужной скалы. Возникающая при этом проблема напоминает рассматриваемую ранее проблему верификации. Вместе с тем в данном случае под успешным решением проблемы аттестации подразумевается достижение роботом нужной скалы. В следующей главе приведены более полные примеры применения вознаграждений для осуществления действий в условиях неопределенности, в которых рассматриваются функции полезности и теорема Байеса.

В рассматриваемом примере с роботом может оказаться, что, во-первых, робот не обладает достоверной информацией о своем местонахождении (т.е. неизвестно состояние), или, во-вторых, робот не может определить, какой путь следует выбрать (т.е. неизвестен какой-то параметр среды). В подобных случаях является очень полезной скрытая марковская модель (Hidden Markov Model — HMM). Разработано программное обеспечение, с помощью которого могут создаваться приложения на основе скрытой марковской модели, такое как инструментарий HTK (`http://htk.eng.cam.ac.uk/`). Этот инструментарий был успешно использован для распознавания и синтеза речи, распознавания символов и расшифровки ДНК (после определения структуры биологических молекул в результате расшифровки входящего в их состав генетического материала появляется возможность синтезировать или модифицировать такие молекулы для распознавания и лечения заболеваний). Еще одним примером программного обеспечения такого типа является HMMER, свободно распространяемая реализация скрытой марковской модели для анализа последовательности белков (`http://hmmer.wustl.edu/`).

Инструментарий, предназначенный для реализации скрытой марковской модели, который поддерживает различные типы логического вывода, применяемые в науке и технике и, в частности, обеспечивающие анализ сигналов, входит в широко применяемый пакет Matlab (`http://www.ai.mit.edu/~murphyk/`

Software/HMM/hmm.html). Классической областью применения такого инструментария является распознавание в речи фонем (и, таким образом, слов), когда пользователь говорит в микрофон, создавая акустический сигнал для анализа на компьютере. Безусловно, задача распознавания раздельной или замедленной речи уже была успешно решена с помощью многих методов, таких как искусственные нейронные системы, но задача распознавания слитной речи любого диктора без обучения системы (распознавания, независимого от диктора) остается сложной.

Кроме того, во многих видеоиграх, основанных на том, что ее персонажи добираются из одного места в другое, преодолевая многочисленные заграждения и препятствия, важной задачей является планирование пути. Для этой цели часто используется широко известный алгоритм A*.

3.18 Резюме

В настоящей главе обсуждались широко применяемые методы логического вывода для экспертных систем. В экспертных системах логический вывод имеет особое значение, поскольку для решения задач в них используется именно этот метод. Рассматривались также вопросы применения деревьев, графов и решеток для представления знаний. Кроме того, показаны преимущества использования этих структур в процедурах логического вывода.

В данной главе подробно рассматривалась дедуктивная логика, начиная с простой силлогистической логики. Вслед за этим описывалась пропозициональная логика и логика предикатов первого порядка. В качестве способов доказательства теорем и утверждений рассматривались истинностные таблицы и правила вывода. Кроме того, затрагивались такие важные характеристики логических систем, как полнота, непротиворечивость и разрешимость.

Кроме того, описан способ применения резолюции для доказательства теорем в пропозициональной логике и логике предикатов первого порядка. На примере продемонстрировано выполнение девяти этапов процедуры, с помощью которой можно преобразовать любую правильно построенную формулу в форму с логическими выражениями. В контексте преобразования правильно построенной формулы в форму с логическими выражениями рассматривались такие темы, как сколемизация, предваренная нормальная форма и унификация.

Обсуждался также еще один мощный метод логического вывода — аналогия. Аналогия используется в экспертных системах недостаточно широко из-за сложностей ее реализации, но по аналогии обычно рассуждают врачи и адвокаты, поэтому часто бывает необходимо предусматривать возможность ее применения в проектах экспертных систем. Кроме того, был описан метод формирования и проверки на примере его использования в системе MYCIN. Наконец, описано

применение метазнаний в системе TEIRESAS и показана связь понятия метазнаний с процедурами верификации и аттестации экспертных систем.

Задачи

3.1. Напишите программу, основанную на использовании деревьев решений, которая является самообучающейся. Проведите обучение этой программы, введя в нее знания о животных, показанные на рис. 3.3.

3.2. Напишите программу, которая автоматически преобразует знания, хранимые в бинарном дереве решений, в правила типа `IF-THEN`. Проверьте работу этой программы на примере дерева решений, предназначенного для распознавания животных, о котором шла речь в условии задачи 3.1.

3.3. Нарисуйте семантическую сеть, содержащую знания о разновидностях кустарников малины, показанные на рис. 3.4.

3.4. Нарисуйте диаграмму состояний, позволяющую найти решение классической задачи с крестьянином, волком, козой и капустой. В этой задаче четыре объекта транспортировки — крестьянин, волк, коза и капуста — находятся на одном берегу реки и должны быть переправлены на другой берег. Для этого необходимо воспользоваться лодкой. Но лодка позволяет перевозить одновременно только два объекта (и только крестьянин может грести). Если волк останется наедине с козой и крестьянин не будет при этом присутствовать, то волк съест козу. А если один на один с капустой останется коза, то коза съест капусту.

3.5. Нарисуйте диаграмму состояний для следующей хорошо структурированной задачи выбора путешествия:

   а) три метода оплаты: наличные деньги, чек или расчетный счет;

   б) две разновидности мест отдыха: морской курорт или горнолыжный курорт;

   в) четыре возможных места назначения, выбор которых зависит от интересов и финансовых возможностей путешественника;

   г) три вида транспорта.

   Запишите правила `IF-THEN`, позволяющие предоставить путешественнику рекомендацию по выбору места отдыха в зависимости от его финансовых возможностей и интересов. Получите информацию о реально предлагаемых местах назначения и узнайте, сколько денег потребуется, чтобы добраться туда из того населенного пункта, в котором вы проживаете.

3.6. Определите, являются ли следующие доказательства действительными или недействительными:

а) $p \rightarrow q, \sim q \rightarrow r, r; \therefore p$

б) $\sim p \vee q, p \rightarrow (r \wedge s), s \rightarrow q; \therefore q \vee r$

в) $p \rightarrow (q \rightarrow r), q; \therefore p \rightarrow r$

3.7. Используя процедуру принятия решений по диаграмме Венна, определите, являются ли следующие силлогизмы действительными или недействительными:

а) AEE-4

б) AOO-1

в) OAO-3

г) AAI-1

д) OAI-2

3.8. Определите, представляют ли собой следующие доказательства логические ошибки или правила вывода. Приведите пример применения каждого доказательства.

а) Сложная конструктивная дилемма

$$\begin{array}{l} p \rightarrow q \\ r \rightarrow s \\ \underline{p \vee r} \\ \therefore q \vee s \end{array}$$

б) Сложная деструктивная дилемма

$$\begin{array}{l} p \rightarrow q \\ r \rightarrow s \\ \underline{\sim q \vee \sim s} \\ \therefore \sim p \vee \sim r \end{array}$$

в) Простая деструктивная дилемма

$$\begin{array}{l} p \rightarrow q \\ p \rightarrow r \\ \underline{\sim q \vee \sim r} \\ \therefore \sim p \end{array}$$

г) Противоположный вывод

$$\begin{array}{l} p \rightarrow q \\ \underline{\sim p} \\ \therefore \sim q \end{array}$$

3.9. Выведите заключение из перечисленных ниже посылок, взятых из известной книги Льюиса Кэрролла “Алиса в стране чудес”.

а) Все обозначенные датой письма в этой комнате написаны на бумаге голубого цвета.

б) Ни одно из этих писем не написано черными чернилами, кроме тех, которые написаны от третьего лица.

в) Я не подшил в папку ни одного из тех писем, которые я смог прочитать.

г) Ни одно из этих писем, которые написаны на одном листе, не обозначено датой.

д) Все из этих писем, которые не являются перечеркнутыми, написаны черными чернилами.

е) Все из этих писем, написанные Брауном, начинаются со слов “Dear Sir”.

ж) Все из этих писем, написанные на бумаге голубого цвета, подшиты в папку.

з) Ни одно из этих писем, написанных больше чем на одном листе, не является перечеркнутым.

и) Ни одно из этих писем, начинающихся со слов “Dear Sir”, не написано от третьего лица.

Подсказка. Используйте закон контрапозитивных высказываний, чтобы определить следующее:

A — письма, начинающиеся со слов “Dear Sir”
B — перечеркнутые письма
C — письма, обозначенные датой
D — письма, подшитые в папку
E — письма, написанные черными чернилами
F — письма, написанные от третьего лица
G — письма, которые я смог прочитать
H — письма, написанные на бумаге голубого цвета
I — письма, написанные на одном листе
J — письма, написанные Брауном

3.10. Воспользуйтесь формальной логикой предикатов для доказательства приведенного ниже силлогизма.

Никакое программное обеспечение не гарантирует своей работоспособности
Все программы представляют собой программное обеспечение
$\therefore$ Ни одна программа не гарантирует своей работоспособности

3.11. Примените следующие сокращения, чтобы записать приведенные ниже высказывания в виде квантифицированных формул логики предикатов первого порядка:

$P(x) - x$ — программист
$S(x) - x$ интеллектуален
$L(x, y) - x$ любит y

а) Все программисты интеллектуальны.
б) Некоторые программисты интеллектуальны.
в) Ни один программист не интеллектуален.
г) Некто не является программистом.
д) Не каждый является программистом.
е) Каждый является не программистом.
ж) Каждый является программистом.
з) Некоторые программисты не интеллектуальны.
и) Существуют программисты.
к) Каждый кого-то любит.

Подсказка. Воспользуйтесь материалами приложения Б.

3.12. Рассмотрим следующее доказательство в логике предикатов:

Лошадь — животное
Следовательно, голова лошади — голова животного.

Определите следующее:

$H(x, y) - x$ — голова y
$A(x) - x$ — животное
$S(x) - x$ — лошадь

Посылка и заключение доказательства являются следующими:

$$(\forall x)(S(x) \rightarrow A(x))$$
$$(\forall x)\{[(\exists y)(S(y) \wedge H(x, y))] \rightarrow [(\exists z)(A(z) \wedge H(x, z))]\}$$

Докажите заключение с использованием опровержения резолюции. Покажите все девять этапов преобразование заключения в форму с логическими выражениями.

3.13. Выполните перечисленные ниже задания.

а) Выясните в банке или ссудо-сберегательной ассоциации, какими критериями они руководствуются при предоставлении ссуды на покупку автомобиля. Запишите систему правил для обратного логического вывода, позволяющую определить, следует ли предоставить ссуду на покупку автомобиля некоторому претенденту. Постарайтесь быть настолько конкретным, насколько это возможно.

б) Выясните, что требуется для оформления ссуды на приобретение жилья. Внесите в свою программу, касающуюся получения ссуды на покупку автомобиля, такие изменения, чтобы она позволяла принимать решения о предоставлении не только ссуд на покупку автомобиля, но и ссуд на приобретение жилья.

в) Выясните, что требуется деловому предприятию для получения ссуды. Внесите в свою программу, касающуюся получения ссуды на покупку автомобиля, такие изменения, чтобы она позволяла принимать решения о предоставлении ссуд деловым предприятиям.

3.14. Разработайте продукционную систему, состоящую из причинных правил, которая позволяла бы моделировать работу топливной системы автомобиля. Необходимая для этого информация содержится в руководстве по ремонту автомобиля. Постарайтесь разработать правила настолько подробно, насколько это возможно.

3.15. Используя руководство по ремонту автомобиля, разработайте продукционную систему, способную диагностировать и давать рекомендации по устранению неисправностей электрической системы автомобиля после ввода сведений об обнаруженных признаках нарушений в работе.

3.16. Если заключение ложно и требуется определить, из каких фактов оно следует, нужно ли использовать абдукцию или другой метод логического вывода? Объясните, почему.

3.17. Напишите правила `IF-THEN` для идентификации разновидностей кустарников малины с использованием части дерева решений, показанного на рис. 3.4.

3.18. Отец и два сына находятся на левом берегу реки и хотят переправиться на правый берег. Отец весит 100 килограммов, а каждый из сыновей — по 50 килограммов. Имеется только одна лодка вместимостью 100 килограммов. В лодке должен находиться по меньшей мере один человек, чтобы грести. Нарисуйте дерево решений, на котором показаны все варианты переправы через реку для этих людей, и укажите путь через дерево, позволяющий им всем успешно переправиться. Применяйте в качестве меток

в дереве следующие обозначения:

SL — один сын переправляется с левого берега реки на правый

SR — один сын переправляется с правого берега реки на левый

BL — оба сына переправляются с левого берега реки на правый

BR — оба сына переправляются с правого берега реки на левый

PL — отец переправляется с левого берега реки на правый

PR — отец переправляется с правого берега реки на левый

3.19. Выполните следующие задания.

а) На рис. 3.11 приведено дерево решений для задачи с автомобилем, состоящее из узлов И–исключительное ИЛИ. Нарисуйте версию этого дерева в виде логической схемы. Обратите внимание на то, что логические элементы “исключительное ИЛИ” необходимо реализовать с использованием стандартных элементов AND, OR и NOT.

б) Запишите правила `IF-THEN` прямого логического вывода для определения того, какую гипотезу следует принять, т.е. продать или отремонтировать автомобиль.

3.20. Какой тип логического вывода применяется в следующих высказываниях (если это действительно правильный логический вывод)?

а) “Причина, по которой я продолжаю настаивать, что были отношения между Ираком, Саддамом и аль-Каедой, состоит в том, что были отношения между Ираком и аль-Каедой”.

б) “Если есть основания полагать, что правосудие Верховного Суда может быть куплено настолько дешево, то наш народ находится в еще более глубокой беде, чем я мог себе представить”.

Глава 4

Рассуждения в условиях неопределенности

4.1 Введение

В настоящей главе обсуждаются некоторые методы формирования рассуждений в условиях **неопределенности** с использованием **теории вероятностей** и **нечеткой логики**. Эти темы очень важны, поскольку одной из самых сильных сторон любой экспертной системы является ее способность справляться с неопределенностью так же успешно, как это делают настоящие эксперты. Дело в том, что если известен приемлемый алгоритм или дерево решений, то не требуется вся мощь экспертной системы. Способность справляться с неопределенностью представляет собой одно из основных преимуществ экспертной системы над простым деревом решений, в котором все факты должны быть известны заранее, чтобы можно было достичь результата. Основой некоторых теорий неопределенности является теория вероятностей, поэтому в данной главе приведены элементарные сведения о теории вероятностей, которые касаются применения вероятностной неопределенности и нечеткой логики в экспертных системах.

В следующей главе будут приведены вводные сведения о некоторых других теориях, позволяющих осуществлять необходимые действия в условиях неопределенности, и нечеткая логика будет рассматриваться более подробно под формальным названием *приближенных рассуждений*. Безусловно, было бы удобно иметь возможность использовать только одну теорию неопределенности, такую как классическая теория вероятностей, но есть и другие важные теории, имеющие свои преимущества и недостатки. Выбор подходящей теории аналогичен таким решениям, которые приходится принимать программисту, который выбирает алгоритм и структуру данных, предназначенные для применения в обыч-

ной компьютерной программе. Понимание преимуществ и недостатков каждого подхода к учету неопределенности позволяет создавать экспертные системы, наиболее подходящие для моделирования конкретных рассматриваемых экспертных знаний. В некоторых инструментальных средствах экспертных систем средства учета неопределенности встроены в сам язык. В качестве примера можно назвать две версии языка CLIPS, которые были доработаны в целях применения нечеткой логики, — язык FuzzyClips, разработанный Национальным научно-исследовательским советом Канады (National Research Council of Canada — NRCC), и язык FuzzyClips компании Togai InfraLogic. В предисловии к настоящей книге указаны также некоторые другие специализированные версии.

Прежде чем приступать к использованию такого специализированного инструментального средства, необходимо убедиться в том, что оно действительно предоставляет подходящий метод учета неопределенности. Дело в том, что может оказаться, что вам просто требуются некоторые конкретные правила, в которых вы сами сможете закодировать нужный метод или определить функцию неопределенности, а не использовать все специализированное инструментальное средство. Еще один вариант состоит в том, что может потребоваться моделировать неопределенность больше чем одного типа, а в вашем распоряжении не окажется одного такого инструментального средства, которое позволяло бы справиться со всеми этими типами. Некоторые программные инструменты и информационные ресурсы, касающиеся проблематики вероятностей, приведены в приложении Ж. Одни из описанных в этом приложении программ представляют собой полноценные инструментальные средства, а другие ресурсы имеют вид специальных классов, например, на языке C++, которые позволяют модифицировать и перекомпилировать программное обеспечение CLIPS для более легкого добавления средств учета неопределенности или других необходимых средств. Такая возможность еще раз демонстрирует одно из самых значительных преимуществ языка CLIPS — то, что этот язык имеет открытый исходный код, а это позволяет настраивать используемый инструмент таким образом, чтобы он соответствовал вашим потребностям, а не пытаться приспособить свои потребности к возможностям инструментального средства. На официальном Web-узле CLIPS можно также найти версии CLIPS, написанные на других языках, таких как Java.

4.2 Неопределенность

Неопределенность может рассматриваться как нехватка адекватной информации для принятия решения. Неопределенность становится проблемой, поскольку может помешать выработке наилучшего решения и даже стать причиной того, что будет принято некачественное решение. В медицине из-за неопределенности может быть не обнаружен наиболее подходящий способ лечения пациента или

выбраны не совсем подходящие терапевтические средства, а в бизнесе неопределенность может привести вместо прибыли к финансовым убыткам.

Разработан целый ряд теорий, позволяющих успешно действовать в условиях неопределенности. К ним относятся теории, основанные на классическом определении вероятностей и на **байесовской вероятности** [13]; теория Хартли, основанная на классическом определении множеств; теория Шеннона, основанная на понятии вероятности; **теория Демпстера–Шефера**; марковские модели; а также **теория нечетких множеств** Заде. В частности, теория байесовских вероятностей и теория нечетких множеств оказались весьма широко применимыми во многих областях, таких как биология, психология, музыка и физика. Нечеткая логика применяется также во многих потребительских приборах, начиная со стиральных машин, способных стирать бельё, измеряя по ходу дела чистоту ткани, а не отсчитывая время по электрическому таймеру, изобретенному еще в XIX веке, и заканчивая фотокамерами, позволяющими автоматически получать великолепные изображения. Нечеткая логика, называемая также *приближенными рассуждениями*, будет рассматриваться более подробно в следующей главе.

Все живые существа являются настоящими экспертами в области учета неопределенности, поскольку в противном случае они не смогли бы выжить в реальном мире. А людям приходится справляться с неопределенностью, касающейся дорожного трафика, погоды, работы, учебы и в целом своей жизни. После небольшого опыта мы становимся экспертами по вождению в различных условиях, начинаем понимать, как бороться с холодом, и выбираем самый легкий способ усвоения знаний. Некоторые люди умудряются даже стать экспертами по выбору самого легкого пути во всем. Для того чтобы уметь справляться с неопределенностью, человек должен научиться рассуждать в условиях неопределенности и иметь много здравого смысла. Единственным недостатком здравого смысла является то, что он не обязательно означает истинную прозорливость, а позволяет лишь определить, что следует делать в обычных ситуациях. Но иногда обычный способ — это не лучший способ, поэтому так важно уметь рассуждать в условиях неопределенности. Например, если вы до сих пор никогда не играли на игорном автомате и вдруг выиграли немного денег, а затем начали проигрывать, здравый смысл шепнет вам на ухо, что если вы будет играть достаточно долго, то в конечном итоге обязательно снова выиграете. К сожалению, здравый смысл ничего не говорит о том, что вы проиграете все наличные, исчерпаете все возможности своих кредитных карточек, продадите свой автомобиль и заложите обручальное кольцо своей супруги, прежде чем вам снова удастся выиграть.

Дедуктивный метод формирования рассуждений, описанный в главе 3, называется **строгими рассуждениями**, поскольку он распространяется на точные факты и точные заключения, которые следуют из этих фактов. Как было сказано выше в этой книге, дедуктивное доказательство является таким строгим потому, что заключение обязательно должно быть истинным, если истинны все посыл-

ки. Аналогичным образом, если посылки ложны, то и заключение должно быть ложным.

С другой стороны, индуктивное доказательство не гарантирует истинности заключения так же надежно, как дедуктивное. Посылки индуктивного доказательства предоставляют некоторое обоснование для заключения, но не гарантируют истинности заключения. Например, предположим, что дана последовательность целых чисел — 1, 2, 3. По методу индукции можно прийти к предположению, что следующим числом должно быть 4. А что вы скажете, если фактически следующим числом является 5? Это вполне возможно, если последовательность из трех чисел, "1,2,3", взята из знаменитой последовательности Фибоначчи, в которой каждое число является суммой двух предыдущих чисел, а рассматриваемой последовательностью является 0,1,1,2,3,5. Индуктивное доказательство обладает таким свойством, что по мере увеличения количества посылок, обосновывающих заключение, повышается степень уверенности в том, что заключение истинно.

Но при использовании указанной интерпретации индукции, согласно которой по мере увеличения посылок, обосновывающих заключение, возрастает вероятность истинности заключения, возникает одна проблема. Предположим, что вы — правокрылый ворон (правокрылыми называются во́роны, имеющие на правом крыле лишнее перо) и в вашем присутствии левокрылый ворон (имеющий лишнее перо на левом крыле — различие между воронами состоит в количестве перьев на крыльях) безапелляционно заявляет, что все во́роны — черные. Услышав это, вы начинает считать воронов и после продолжительного времени замечаете, что весь миллион осмотренных вами воронов действительно имели черный цвет. Безусловно, в этот момент какие-то слабонервные представители правокрылых объявили бы о своей капитуляции, но поскольку эту теорию выдвинул левокрылый, вы думаете: "Не следует думать, что нет нечерных воронов, лишь потому, что я не видел ни одного из них. Возможно, все они просто скрываются". Разумеется, такое предположение вполне может быть истинным (или даже причина состоит в том, что всех нечерных воронов за одну ночь похитили существа с неопознанных летающих объектов), но само утверждение, что "все вороны — черные", логически эквивалентно утверждению "все нечерные предметы не являются воронами".

Например, красное яблоко, безусловно, не является черным вороном. Поэтому, по мере того, как вы будете наблюдать все больше и больше красных яблок, тем выше и выше будет становиться вероятность того, что все вороны — черные, поскольку будут поступать дополнительные подтверждения посылок, т.е. увеличиваться количество нечерных предметов, не являющихся воронами. Эта смешная история представляет собой проявление так называемого **парадокса ворона**, поскольку она показывает, в чем применение индукции может противоречить интуиции.

Итак, анализ этой ситуации может вызвать у правокрылого ворона полное разочарование, поскольку количество красных яблок очень велико, а из этого сле-

дует, что левокрылые вороны были правы, утверждая, что вороны — черные. Но еще не время сдаваться! Достаточно подумать о том, что на Земле есть также много черных ягод ежевики. Поэтому, если вы начнете считать черные предметы, не являющиеся черными воронами, то ослабите тем самым посылку, что некоторые из черных предметов обязательно являются черными воронами. А после того как вы отправитесь в овощной магазин, в котором продается ежевика, и начнете считать ягоды, то обнаружите, что даже при наличии небольших запасов в магазине всегда намного больше ягод ежевики, чем красных яблок (ведь вы определяете количество поштучно, а не на килограммы). А поскольку ягод ежевики намного больше, чем яблок, из этого следует вывод, что левокрылые не правы и должны действительно существовать нечерные вороны, даже если вы их еще не обнаружили!

Попытка подсчета красных яблок или каких-то других нечерных предметов для повышения правдоподобия гипотезы, согласно которой все вороны — черные, может привести к неправильным заключениям. Были предложены различные решения парадокса ворона, но самым лучшим из них, по-видимому, является решение, основанное на байесовской теории, которая рассматривается ниже в данной главе.

Если речь идет о неопределенных фактах, строгие рассуждения становятся неприменимыми. Тем не менее люди могут ошибочно думать иначе, как показано на примере человека, играющего роль правокрылого ворона или азартного игрока, описанного перед этим. Дело в том, что из-за неопределенности возрастает количество возможных результатов, поэтому может оказаться не просто трудно, но и вообще невозможно найти наилучшее решение. Что еще хуже, неопределенность может привести к тому, что мы выберем неправильный способ рассуждений и в конечном итоге, отыскивая решение, станем целыми днями перебирать ягоды ежевики.

К сожалению, задача определения “наилучшего” заключения может оказаться непростой. Предложен целый ряд различных способов организации действий в условиях неопределенности и средств, способствующих выбору наилучшего заключения. Но если речь идет о том, как действовать в условиях неопределенности, то может потребоваться, чтобы мы довольствовались просто достаточно качественным решением, а не стремились к наилучшему решению. С другой стороны, качественное решение, достижимое на 99% в реальном времени, может оказаться более приемлемым, чем оптимальное решение, для вычисления которого потребуется миллион лет. Именно проектировщик экспертной системы отвечает за выбор метода, наиболее подходящего для данного конкретного приложения. Безусловно, есть такие инструментальные средства экспертных систем, в которых предусмотрены механизмы формирования рассуждений в условиях неопределенности, но, как правило, они не обладают достаточной гибкостью, чтобы дать возможность использовать другие методы. Есть такая пословица, что если един-

ственным инструментом является молоток, то все остальное кажется гвоздем. Выбор инструмента, применимого только в одной области, напоминает выбор для работы только молотка.

Безусловно, многие приложения экспертных систем могут быть основаны на использовании строгих рассуждений, но для значительного количества других приложений требуются **нестрогие рассуждения**, поскольку сами факты или знания точно не известны. В качестве одного из наиболее ярких примеров можно указать коммерческую медицину. Для оптимизации прибыли необходимо стремиться к такой цели — найти приемлемый способ лечения с помощью минимального количества анализов, поскольку анализы стоят денег. В экспертных системах могут применяться неопределенные факты, правила, или то и другое. Классическими примерами экспертных систем, успешно действующих в условиях неопределенности, являются система MYCIN, предназначенная для медицинской диагностики, и система PROSPECTOR, применяемая для разведки полезных ископаемых.

Основная причина, по которой эти системы так часто упоминаются в литературе, состоит в том, что они имели очень качественную документацию и предназначались для демонстрации применимости экспертных систем в качестве консультантов, эквивалентных или лучших по сравнению с экспертами-людьми, что и было достигнуто в обеих системах. Это было давно, а в наши дни вы не сможете получить доступ к точным правилам, по которым экспертные системы бюро кредитования определяют вашу оценку кредитоспособности, а также не узнаете, почему в обслуживании вашей кредитной карточки было отказано тем же банком, который недавно выдал вам разрешение на крупную ссуду. К тому же экспертные системы — это прежде всего программы, поэтому разве не может быть шансов, что в коде самих этих программ имеются ошибки? Тем не менее вам все равно не позволят ознакомиться с кодом, чтобы определить, не была ли причиной того, что вам отказали в кредите или ссуде, сама программа.

В системах MYCIN и PROSPECTOR предусмотрена возможность выработки заключений даже в тех случаях, если не известны все факты, необходимые для абсолютно надежного доказательства каждого заключения. Безусловно, если речь идет о медицине, то было бы возможно получить более надежное заключение, выполнив больший объем анализов, но при этом возникает проблема, связанная с увеличением затрат времени и денег на выполнение анализов. Ограничения, касающиеся времени и денег, особенно важны в случае медицинского обслуживания. Задержка в предоставлении лечения в целях проведения дополнительных анализов приводит к существенному увеличению затрат; между тем пациент, не дождавшись помощи, может умереть. А если речь идет о разведке полезных ископаемых, то стоимость дополнительных исследований также является существенным фактором. Может оказаться, что более экономически выгодное решение состоит в том, чтобы приступить к бурению скважины, если вы уверены в успе-

хе на 95%, чем потратить еще сотни тысяч долларов, чтобы добиться 98%-ной уверенности.

4.3 Типы ошибок

Неопределенность может увеличиваться под влиянием **ошибок** многих типов. Разработан целый ряд теорий неопределенности, в которых предпринимается попытка устранения некоторых или даже всех ошибок и обеспечения наиболее надежного логического вывода. Упрощенная классификационная схема ошибок показана на рис. 4.1.

Строго говоря, схема на этом рисунке должна быть представлена в виде решетки, поскольку между различными типами ошибок могут возникать дополнительные взаимосвязи. Например, субъективная ошибка может быть связана с неоднозначностью, ошибкой измерения, ошибкой рассуждения и т.д. Примеры подобных ошибок приведены в табл. 4.1.

Первым типом ошибки, показанным в таблице, является **неоднозначность**. Возникновение такой ошибки связано с тем, что некоторая информация может интерпретироваться несколькими разными способами. Ошибкой второго типа является **неполнота**, которая связана с отсутствием некоторой информации. Ошибкой третьего типа является **неадекватность**, которая обусловлена применением информации, не отражающей сложившуюся ситуацию. Возможными причинами неадекватности являются субъективные ошибки, такие как случайное ошибочное чтение показаний приборов или данных, ложь или дезинформация, а также неисправность оборудования.

Гипотеза — это предположение, подлежащее проверке. **Нулевая гипотеза** — это предположение, принятое первоначально, такое как “вентиль заблокирован”. Один из типов неправильной информации называется **ложно положительным** и означает принятие гипотезы, не являющейся истинной. Аналогичным образом, применение неправильной информации, которая относится к **ложно отрицательному** типу, означает, что отвергается гипотеза, являющаяся истинной. Таким образом, если вентиль в действительности не заблокирован, то принятие гипотезы, что он заблокирован, является ложно положительным. В статистике такая ошибка называется **ошибкой первого рода**. Аналогичным образом, если вентиль в действительности заблокирован, а гипотеза “вентиль заблокирован” отвергается, то такая ошибка называется ложно отрицательной, или **ошибкой второго рода**.

Следующими двумя типами ошибок, показанных в табл. 4.1, являются **погрешности измерения**. Эти ошибки могут касаться **точности** и **правильности**. Безусловно, данные термины иногда используются как синонимы, но фактически между ними имеется важное различие. Рассмотрим две линейки, одна из которых градуирована в миллиметрах, а другая — в сантиметрах. Безусловно, миллимет-

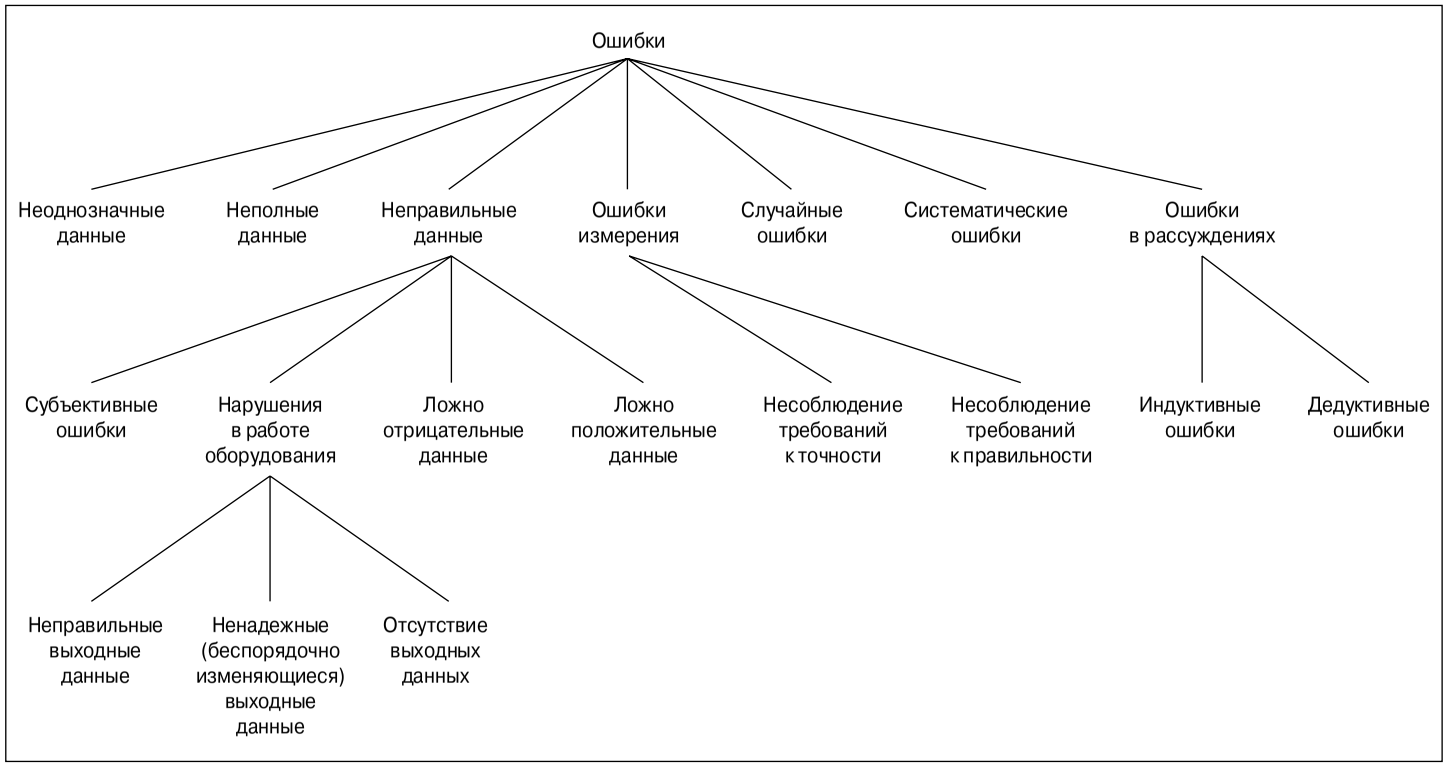


Рис. 4.1. Типы ошибок

Таблица 4.1. Примеры наиболее распространенных типов ошибок

Пример	Ошибка	Причина
Закрыть вентиль	Неоднозначность	Какой вентиль?
Повернуть вентиль 1	Неполнота	В какую сторону?
Закрыть вентиль 1	Неадекватность	Указанием, адекватным сложившейся ситуации, было бы “открыть”
Вентиль заблокирован	Ложно положительный диагноз	Вентиль не заблокирован
Вентиль не заблокирован	Ложно отрицательный диагноз	Вентиль заблокирован
Повернуть вентиль 1 в положение 5	Неточность	Точным указанием было бы — в положение 5,4
Повернуть вентиль 1 в положение 5,4	Неправильность	Правильным указанием было бы — в положение 9,2
Повернуть вентиль 1 в положение 5,4, или 6, или 0	Ненадежность	Ошибка оборудования
Положение вентиля 1 — 5,4, или 5,5, или 5,1	Случайная ошибка	Статистическая флуктуация
Вентиль 1 находится в положении 7,5	Систематическая ошибка	Неправильная калибровка шкалы
Вентиль 1 не заблокирован, поскольку никогда еще до сих пор не оказывался в заблокированном состоянии	Недействительная индукция	Вентиль заблокирован
Выходной сигнал в норме, поэтому вентиль 1 — в хорошем состоянии	Недействительная дедукция	Вентиль заблокирован в открытом положении

ровая линейка является более точной, чем сантиметровая. Но предположим, что разметка шкалы на миллиметровой линейке сделана неправильно. В таком случае применение миллиметровой линейки приводит к получению неправильных результатов измерения, и этим результатам нельзя доверять, не зная коэффициента корректировки. Таким образом, правильность соответствует истине, а точность — тому, насколько хорошо известна истина. В рассматриваемом примере двух линеек миллиметровая линейка измеряет в десять раз точнее, чем сантиметровая, но если измеренные ею данные являются неправильными, это может привести к возникновению серьезных проблем.

Еще одним типом ошибки является **ненадежность**. Если измерительное оборудование, которое служит источником фактов, работает ненадежно, то полученные с ее помощью данные **изменяются беспорядочно**. Беспорядочно изменяющимися показаниями прибора называются показания, которые не являются постоянными, а непрерывно колеблются. Иногда такие показания могут оказаться правильными, а иногда — нет.

Колебания показаний измерительного прибора могут быть вызваны тем, что сама изучаемая система по существу имеет случайных характер, такой как распад атомов радиоактивных изотопов. Причиной распада служат явления квантовой механики, поэтому результаты измерения интенсивности распада изменяются случайным образом, отклоняясь от среднего значения. Случайные колебания относительно среднего значения представляют собой **случайную ошибку** и приводят к неопределенности среднего. Случайные ошибки других типов могут быть вызваны броуновским движением, электронным шумом, обусловленным тепловыми эффектами, и т.д. Но беспорядочное изменение показаний может быть также вызвано некачественными электрическими соединениями.

Систематическими ошибками называются такие ошибки, которые не являются случайными и возникают из-за какого-то смещения. Например, неправильная разметка шкалы линейки, из-за которой ее деления становятся меньше обычных, приводит к систематической ошибке, выражающейся в том, что результаты измерения становятся больше обычных.

4.4 Ошибки и индукция

Следующим типом ошибок является **недействительная индукция**, в рамках которой могут, например, проводиться такие рассуждения: “Вентиль не может быть заблокирован, поскольку до сих пор еще ни разу не возникали случаи, в которых он был бы заблокирован”. Процесс проведения **индукции** является противоположным дедукции. Обычно принято считать, что дедукция осуществляется от общего к частному, как в следующем примере, для вывода конкретного заключения о том, что Сократ смертен:

Все люди смертны
Сократ — человек
∴ Сократ смертен

Кроме того, часто утверждают, что процесс индукции осуществляется от частного к общему, как в следующем примере:

Жесткий диск моего компьютера до сих пор еще никогда не отказывал
∵ Жесткий диск моего компьютера никогда не откажет

В этом примере символ треугольника, повернутый вершиной вниз (∴), обозначает "**индуктивное заключение**", в отличие от обычного треугольника (.·.), обозначающего "**дедуктивное заключение**". Приведенное выше ошибочное индуктивное доказательство, касающееся дисков, показывает, что применение старомодных взглядов, согласно которым дедукция рассматривается как рассуждения от частного к общему, приводит к возникновению проблем. В действительности выше приведено ошибочное дедуктивное доказательство, поскольку в нем рассуждения проводятся от общего случая (согласно которому жесткий диск на рассматриваемом компьютере еще никогда не отказывал) к частному случаю (согласно которому он не должен когда-либо отказать).

Безусловно, читателю все еще могут встретиться некоторые книги, содержащие утверждения, что дедукция осуществляется от общего к частному, а индукция — от частного к общему, но в современной логике считается, что дедукция отличается тем, что гарантирует истинность заключения, если истинны посылки. А то, что доказательство является индуктивным, просто означает следующее: чем строже посылки, тем проще доказать по индукции, что заключение должно быть истинным.

Рассмотрим приведенные ниже утверждения.

Двумя моими домашними животными являются Смоки и Бутс

У Смоки есть хвост

У Бутса есть хвост

∴ У всех моих домашних животных есть хвосты

Это — пример действительного дедуктивного доказательства, в котором применение частных посылок ведет к общему заключению; это противоречит тому, чему учили раньше, — будто дедукция осуществляется от общего к частному, причем такое определение все еще можно найти в некоторых старых словарях. Мораль этого повествования заключается в том, что даже математика и логика не остаются неизменными. Доказательство продемонстрированного выше типа называется также **непротиворечивым**, поскольку оно одновременно является и действительным, и основанным на истинных посылках. В данном случае под истинностью подразумевается семантически действительное понятие; это означает, что рассматриваемое доказательство имеет смысл и в реальном мире, поскольку оба моих домашних животных действительно имеют хвосты.

Теперь рассмотрим следующее доказательство:

Двумя моими домашними животными являются Смоки и Бутс

У Смоки есть хобот

У Бутса есть хобот

∴ У всех моих домашних животных есть хоботы

Это доказательство все еще остается действительным, но больше не является непротиворечивым, поскольку Смоки и Бутс — не слоны, поэтому не имеют хоботов. (Я мог бы приготовить для них даже сундуки с приданым, но из-за этого мои домашние животные не стали бы другими.)

Правильность индуктивных доказательств невозможно доказать, кроме как при использовании математической индукции. Вместо этого индуктивные доказательства позволяют только достичь определенной степени уверенности в том, что заключение является правильным. Мы не можем испытывать значительного доверия к приведенному выше индуктивному доказательству, касающемуся отказа жесткого диска. В качестве еще одного примера рассмотрим следующее доказательство:

Раздался звук пожарной сирены

∴ Происходит пожар

или даже еще более строгое индуктивное доказательство:

Раздался звук пожарной сирены

Я чувствую запах дыма

∴ Происходит пожар

Безусловно, последнее доказательство является более строгим, но на его основании нельзя заключить, что происходит пожар. Причиной появления дыма может стать поджаривание сосисок для гамбургеров на гриле, а сигнал пожарной тревоги может быть включен случайно. Дедуктивным доказательством пожара является следующее:

Раздался звук пожарной сирены

Я чувствую запах дыма

Моя одежда начала тлеть

∴ Происходит пожар

Обратите внимание на то, что данное доказательство — дедуктивное (поскольку вы можете наблюдать действие пламени), даже несмотря на то, что оно проводится от частных посылок, а не общих.

Экспертные системы могут состоять из дедуктивных и индуктивных правил; при этом индуктивные правила имеют эвристический характер. Кроме того, как будет описано в одной из последующих глав, посвященных приобретению знаний, индукция применяется также для автоматической выработки правил.

Безусловно, неопределенность может быть связана с фактами, но может также присутствовать в правилах экспертной системы, если эти правила основаны на эвристике. Эвристические правила часто называют эмпирическими правилами (“rule of thumb” — буквально “правило большого пальца”), поскольку они основаны на личном опыте. (Например, если вы хотите усвоить важное правило,

касающееся правильного забивания гвоздя, вначале стукните по своему большому пальцу. Это и есть обучение на основе “правила большого пальца”. Эффект от такого обучения является просто потрясающим по сравнению с сидением на лекции в течение трех часов и обучением в полусне по методу “гипнопедии”.)

Иногда опыт служит хорошим наставником. Но может оказаться, что опыт не применим в 100% случаев, поскольку он является эвристическим. Например, можно предположить, что любой, кто выпрыгнет из окна пятидесятиэтажного здания, погибнет. Но люди могут спрыгивать с высоты пятидесятого этажа и не погибать, если у них есть парашюты.

Одной из привлекательных характеристик экспертов-людей служит то, что они успешно проводят свои рассуждения в условиях неопределенности. Даже если неопределенность весьма существенна, эксперты обычно способны сформулировать качественные суждения, поскольку в противном случае им недолго приходилось бы играть роль экспертов.

Еще одной характерной особенностью является то, что эксперты обычно способны легко пересматривать свое мнение, если обнаружится, что некоторые из исходных фактов оказались неправильными. Как описано в главе 3, такое поведение эксперта относится к категории немонотонных рассуждений. Задача разработки программного обеспечения экспертных систем в целях обеспечения возврата в процессе формирования рассуждений является более сложной, поскольку для этого необходимо запоминать все промежуточные факты и накапливать хронологические данные о запуске правил. А что касается экспертов-людей, то создается впечатление, что они пересматривают свои рассуждения с очень малыми усилиями (это особенно касается политических деятелей).

Как было описано в главе 3, кроме индуктивных ошибок могут также возникать дедуктивные ошибки, которые называют логическими ошибками. В частности, в главе 3 рассматривалась следующая ошибочная схема:

$$\begin{array}{l} p \rightarrow q \\ \underline{q} \\ \therefore p \end{array}$$

Например, приведенное ниже доказательство является ошибочным.

Если вентиль — в хорошем состоянии, то его выходной сигнал — в норме
Выходной сигнал вентиля — в норме
$\therefore$ Вентиль — в хорошем состоянии

Вентиль может оказаться заблокированным в открытом положении, поэтому выходной сигнал будет нормальным. Но если потребуется закрыть вентиль, проблема сразу же станет очевидной. Такая неисправность может оказаться очень

серьезной, если вентиль нужно закрыть быстро, допустим, в случае аварийной остановки ядерного реактора.

В отличие от ошибок, описанных в предыдущем разделе, индуктивные и дедуктивные ошибки — это ошибки рассуждений. Возникновение ошибок такого типа приводит к неправильной формулировке правил.

Вообще говоря, создается впечатление, что люди не обрабатывают неопределенную информацию наилучшим способом из всех возможных. Иммунитетом против ошибок не обладают даже эксперты; это особенно проявляется в условиях неопределенности. Такая проблема становится очень важной в процессе приобретения знаний, в ходе которого необходимо представить знания эксперта качественно и количественно в виде правил. Во время этого могут обнаруживаться несогласованности, неточности и другие возможные ошибки, обусловленные неопределенностью. После этого экспертам приходится корректировать знания, полученные с их помощью, что может привести к задержке окончательного выпуска экспертной системы.

4.5 Классическая вероятность

Одним из самых старых и все еще очень важных инструментальных средств решения задач искусственного интеллекта является **вероятность** [41]. Вероятность — это количественный способ учета неопределенности. Понятие вероятности зародилось в XVII веке, после того как некоторые азартные игроки во Франции обратились за помощью к ведущим математикам, таким как Паскаль, Ферма и др. В то время азартные игры нашли очень широкое распространение, а поскольку в ходе игры привлекались крупные суммы денег, игроки стремились овладеть методами, позволяющими вычислять шансы на выигрыш. В действительности классическая задача разорения игрока — это доказательство с помощью теории вероятностей эмпирического факта, замеченного игроками: игорный дом всегда выигрывает, если вы состязаетесь с ним достаточно долго [79]. (К сожалению, больше всех извлекли бы пользы из изучения теории вероятностей именно те игроки, которые больше всех и проигрывают.)

Классическая вероятность рассматривалась в теории, которая была впервые предложена Паскалем и Ферма в 1654 году. С того времени была проведена большая работа в области изучения вероятностей и создано несколько новых научных направлений. Многочисленные приложения вероятностей обнаруживаются в науке, технике, бизнесе, экономике и практически во всех других областях.

Определение классической вероятности

Классическую вероятность называют также **априорной вероятностью**, поскольку ее определение относится к идеальным играм или системам. Как было

описано в главе 2, термин *априорный* означает "предшествующий" (имеется в виду предшествующий опыту), т.е. принятый без учета того, что происходит в реальном мире. Понятие априорной вероятности распространяется на игры, в которых рассматриваются результаты выпадения очков на игральных костях, раздачи карт и подбрасывания монет, а также прочие события, происходящие в идеальных системах, не подверженных износу.

Идеальные системы не обнаруживают износа, характерного для реальных систем, поскольку в противном случае невозможно было бы изучать точно воспроизводимые характеристики идеальных систем. Это означает, что настоящая игральная кость может обнаруживать смещение в сторону некоторых конкретных результатов, после того как одна ее грань станет изношенной из-за многочисленных бросков. Аналогичным образом, в зависимости от изготовителя, настоящая игральная кость может обнаруживать преимущественное выпадение большего числа очков, поскольку чем больше очков, тем больше точек (небольших заполненных краской отверстий) просверлено на грани кости. Указанные смещения действительно были обнаружены в результате анализа данных одного миллиона бросков настоящей игральной кости. После каждых 20 тысяч бросков использовалась новая игральная кость для предотвращения смещения, вызванного неравномерным износом граней. В табл. 4.2 показано относительное количество выпадений различного количества очков.

Таблица 4.2. Результаты анализа миллиона бросков игральной кости

Количество очков	1	2	3	4	5	6
Относительное количество выпадений	0,155	0,159	0,164	0,169	0,174	0,179

А в идеальной системе выпадение любого количества очков происходит одинаково, благодаря чему анализ становится намного проще.

Фундаментальная формула классической вероятности определена как следующая вероятность:

$$P = \frac{W}{N}$$

В этой формуле W — количество ожидаемых событий, а N — общее количество равновероятных **событий**, которые являются возможными результатами эксперимента, или **испытания**. Например, один бросок игральной кости является единственным испытанием, в результате которого наступает одно событие из шести возможных. Игральная кость после броска прекращает свое движение, и на верхней грани обнаруживается количество очков 1, 2, 3, 4, 5 или 6. Согласно классическому определению вероятности, предполагается, что любое из этих шести событий является равновозможным, и поэтому вероятность выпадения количества

очков 1, $P(1)$, составляет следующее:

$$P(1) = \frac{1}{6}$$

Аналогичным образом, вероятность выпадения количества очков, равного 2, измеряется следующей формулой, и т.д.:

$$P(2) = \frac{1}{6}$$

Вероятность проигрыша, Q, т.е. невыпадения требуемого количества очков, показана ниже.

$$Q = \frac{N - W}{N} = 1 - P$$

Фундаментальная формула для P представляет собой априорное определение, поскольку вероятность вычисляется еще до проведения игры. Термин априорный означает "предшествующий", или "происходящий до события", поэтому если речь идет о вероятностях, то в определении априорной вероятности предполагается, что все возможные события известны и возникновение каждого события является равновероятным.

Например, известно, что после броска игральной кости количество отметок на каждой грани, которая может выпасть, равно 1, 2, 3, 4, 5 и 6. Кроме того, если игральная кость является правильной (не имеет смещенный центр тяжести), то выпадение каждой грани будет одинаково вероятным. Аналогичным образом, известно, что в "правильной" колоде карт имеются все 52 разные карты и извлечение из колоды любой из этих карт после правильной тасовки происходит с одинаковой вероятностью. Вероятность выпадения любой грани игральной кости равна $1/6$, а извлечения любой карты — $1/52$.

Если все повторяющиеся испытания дают точно один и тот же результат, то система рассматривается как **детерминированная**. Если же система не является детерминированной, то на нее распространяется определение **недетерминированной**. Но, строго говоря, понятие *недетерминированный* не является точно таким же, как понятие *случайный*. Дело в том, что термин *случайный* может иметь положительную или отрицательную окраску. Например, такое случайное событие, как бросок игральной кости в Лас-Вегасе, может сделать вас миллионером или нищим, в зависимости от результата. В отличие от этого, если созданная система оказалась недетерминированной, это означает, что в ней может быть предусмотрено несколько способов достижения одной или большего количества целей, при наличии одних и тех же входных данных.

Например, после ввода некоторой цифры недетерминированный конечный автомат может перейти либо в состояние 1, либо в состояние 2. Если вводятся только цифры, в конечном итоге такой автомат все равно распознает вводимые

целые числа. А если вводятся вещественные числа в системе обозначений с десятичной точкой или с показателем степени, то конечный автомат в результате распознает и это, перейдя в другое заключительное состояние. Вообще говоря, проект недетерминированного конечного автомата предусматривает использование меньшего количества состояний по сравнению с детерминированным, а сам недетерминированный конечный автомат может быть преобразован в детерминированный.

Люди, использующие машины поиска, хорошо представляют себе, как проявляется недетерминированность. При одних и тех же входных данных на выходе машины поиска при разных попытках появляется список, начинающийся каждый раз с другой ссылки (если это — не "спонсируемая" ссылка, которая всегда появляется первой). Например, перейдите на узел `google.com`, введите поисковый запрос "`giarratano expert systems`" и отметьте, какие результаты приведены в списке на первом месте. После этого попытайтесь ввести запрос "`expert systems giarratano`", и вы увидите, что получены другие результаты, даже несмотря на то, что машине поиска были переданы одни и те же искомые термины.

В качестве еще одного примера можно указать такую ситуацию, когда у человека появляется головная боль. Чтобы попытаться вылечить головную боль, можно воспользоваться многими способами, а выбор конкретного способа зависит от состояния человека или от того, какие средства находятся в его распоряжении. В инструментальных средствах экспертных систем некоторых типов, таких как CLIPS, недетерминированность применяется для предотвращения преимущественного запуска одних и тех же правил. Если выполняются шаблоны многочисленных правил и отсутствуют явные предпочтения в отношении того, какое правило должно быть выполнено, то машина логического вывода осуществляет произвольный выбор правила, подлежащего запуску. Благодаря этому исключается тенденциозность, из-за которой всегда происходил бы запуск одного и того же правила, скажем, первого. Такая ситуация могла бы возникнуть, если бы все знания были представлены на обычном языке программирования в виде правил `IF-THEN`, а программа просто выбирала бы эти правила в том порядке, в каком они были введены в исходном коде. В действительности в этом состоит одна из причин, по которым экспертные системы не разрабатываются на обычном языке программирования, — в таких языках слишком трудно преодолеть детерминированность! Машина логического вывода экспертной системы в своей работе действует подобно человеку и не всегда принимает одно и то же решение при наличии нескольких правил с равным приоритетом. Так же поступает человек, размышляя над вопросом: "Разболелась голова, и что мне делать — принять аспирин или водку с томатным соком ("кровавую Мери")"?

Выборочные пространства

Результатом испытания становится **элемент выборки**, а множество всех возможных элементов выборки определяет **выборочное пространство**. Например, если осуществляется бросок единственной игральной кости, то может обнаружиться любой из элементов выборки 1, 2, 3, 4, 5 или 6. На рис. 4.2, *а* показано выборочное пространство для испытания с броском единственной игральной кости. Выборочным пространством является множество $\{1, 2, 3, 4, 5, 6\}$.

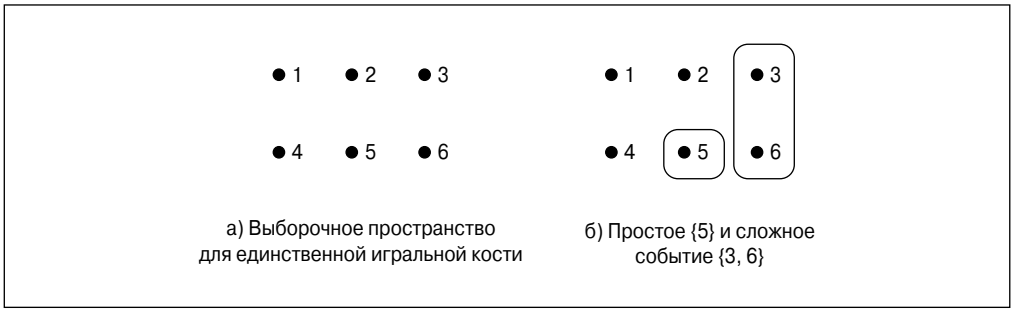


Рис. 4.2. Выборочное пространство и события

Событием называется подмножество выборочного пространства. Например, событие $\{1\}$ происходит, если на игральной кости выпадает количество очков 1. Как показано на рис. 4.2, **элементарное событие** включает только один элемент, а **сложное событие** — больше одного.

Графический способ определения выборочного пространства состоит в построении **дерева событий**. В качестве простого примера предположим, что имеются два компьютера, которые могут работать, W, или не работать, D. На рис. 4.3 показано дерево событий, а в табл. 4.3 приведено описание выборочного пространства в табличной форме. Обратите внимание на то, что сложные события перечисляются в определенной последовательности, например {computer1, computer2}. Выборочным пространством является множество $\{WW, WD, DW, DD\}$.

Таблица 4.3. Выборочное пространство с бинарными событиями

		Компьютер 2	
		W	D
Компьютер 1	W	WW	WD
	D	DW	DD

Рассматриваемое в данном примере дерево событий относится к типу бинарных деревьев, поскольку в нем участвуют только бинарные вероятности. Это

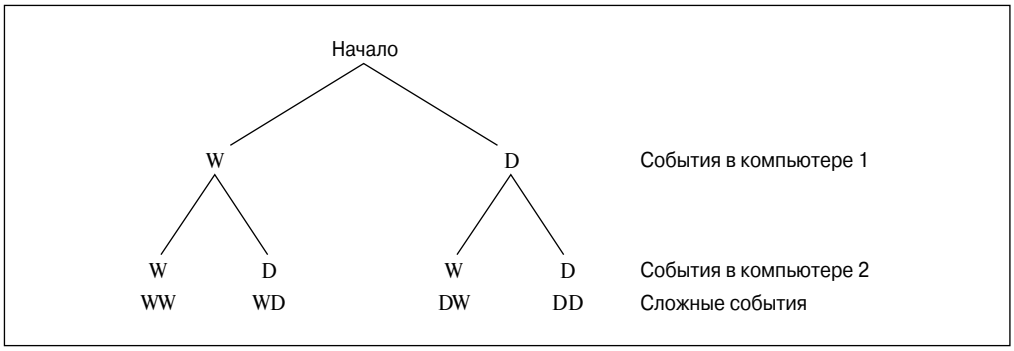


Рис. 4.3. Дерево событий для сложного события

означает, что в рассматриваемых событиях компьютер либо работает, либо не работает. Деревья такого типа могут применяться для представления задач многих типов. Например, изучение бросков монеты приводит к созданию бинарного дерева событий, поскольку при каждом броске могут быть только два возможных результата.

В теории вероятностей и статистике широко используется общий математический аппарат, но статистика главным образом посвящена сбору и анализу данных о **совокупностях** — множествах, из которых извлекаются выборки. Одна из задач, которые решаются в рамках статистики, состоит в формировании выводов относительно генеральной совокупности, из которой извлекается совокупность выборок. Типичным примером статистического расчета является определение на основании опроса того, какая часть зарегистрированных избирателей отдаст свои голоса за определенного кандидата.

Одним из приложений теории вероятностей является определение того, действительно ли рассматриваемая выборка, которая включает часть избирателей, является представительной для всех избирателей, или эта выборка сформирована тенденциозно, в пользу лишь определенной партии. Безусловно, этот пример относится к реальным объектам, таким как люди, участвующие в выборах, но могут рассматриваться и другие, гипотетические примеры, касающиеся таких совокупностей, как возможное количество бросков монеты. Каждый бросок монеты становится выборкой из гипотетической совокупности всех бросков монеты.

Как показано на рис. 4.4, основой формирования рассуждений о статистических совокупностях являются дедукция и индукция. Если дана известная совокупность, то дедукция позволяет формировать логические выводы, касающиеся неизвестной выборки. Соответствующим образом, если дана известная выборка, то индукция позволяет формировать логические выводы, касающиеся неизвестной совокупности.

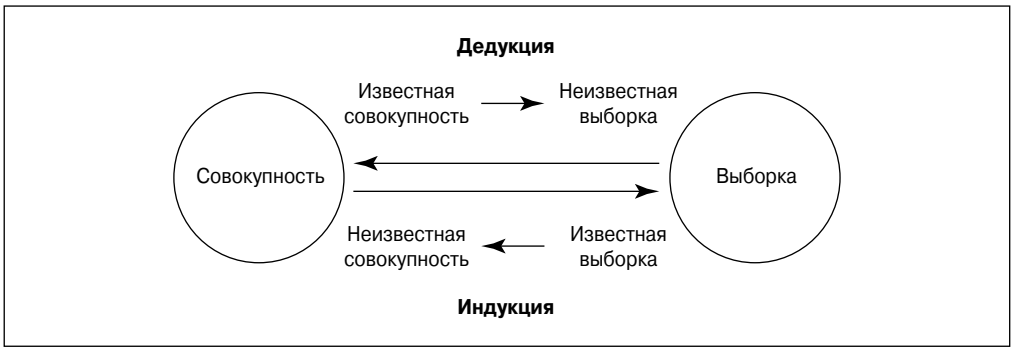


Рис. 4.4. Дедуктивные и индуктивные рассуждения о совокупностях и выборках

Теория вероятностей

Формальная теория вероятностей может быть создана на основе трех описанных ниже аксиом.

Аксиома 1 : $$0 \leqslant P(E) \leqslant 1$$

В этой аксиоме утверждается, что областью определения вероятностей являются вещественные числа от 0 до 1. Отрицательные значения вероятностей не допускаются. **Достоверному событию** присваивается вероятность 1, а **невозможному событию** — вероятность 0.

Аксиома 2 : $$\sum_i P\left(E_i\right) = 1$$

В данной аксиоме утверждается, что сумма вероятностей всех событий, не зависящих друг от друга (называемых **взаимоисключающими событиями**), равна 1. Взаимоисключающие события не имеют каких-либо общих элементов выборки. Например, компьютер не может одновременно работать правильно и неправильно (если только это не квантовый компьютер).

Из этой аксиомы следует приведенное ниже заключение, в котором символ E' обозначает дополнение события E.

$$P(E) + P(E') = 1$$

Согласно данному следствию, сумма вероятностей того, что некоторое событие произойдет и не произойдет, равна 1. Таким образом, появление и непоявление события являются взаимоисключающими событиями и полное выборочное пространство, в котором E_1 и E_2 — взаимоисключающие события, определяется следующим образом:

Аксиома 3 : $$P(E_1 \cup E_2) = P(E_1) + P(E_2)$$

В этой аксиоме указано, что если события E_1 и E_2 не могут возникать одновременно (т.е. являются взаимоисключающими событиями), то вероятность возникновения того или другого события равна сумме вероятностей этих событий.

Как описано ниже, на основании вышеприведенных аксиом могут быть выведены теоремы, касающиеся вычисления вероятностей в других ситуациях, например, при наличии невзаимоисключающих событий. Безусловно, эти аксиомы позволяют заложить фундамент теории вероятностей, но заслуживает внимания то, что в них не рассматриваются основные вероятности $P(E)$. Основные вероятности событий определяются с использованием других методов, например, с помощью априорных вероятностей.

Описанные выше аксиомы позволяют перевести изучение вероятностей на надежное теоретическое основание. Фактически рассматриваемая аксиоматическая теория называется также **объективной теорией вероятностей**. Данные конкретные аксиомы были предложены Колмогоровым, а Реньи разработал эквивалентную теорию с использованием аксиом условных вероятностей.

4.6 Экспериментальные и субъективные вероятности

Изучение классических вероятностей позволяет отвечать лишь на вопросы, касающиеся идеальных игр с равным правдоподобием, но не дает возможности узнать, какова вероятность того, что на вашем компьютере завтра произойдет отказ жесткого диска, или какова средняя вероятная продолжительность жизни вашего ближайшего родственника (если только вы не направили на него заряженный пистолет).

В отличие от априорного подхода при изучении подобных проблем с использованием **экспериментальной вероятности** применяется способ определения вероятности некоторого события $P(E)$ как предела распределения частот:

$$P(E) = \lim_{N\to\infty} \frac{f(E)}{N}$$

В этой формуле $f(E)$ обозначает частоту появления некоторого события среди N наблюдаемых общих результатов. Вероятность такого типа называется также **апостериорной вероятностью**, т.е. вероятностью, определяемой “после события”. Для обозначения апостериорной вероятности используется также термин **эмпирическая вероятность**. В основе определения апостериорной вероятности лежит измерение частоты, с которой возникает некоторое событие во время проведения большого количества испытаний, и последующее вычисление экспериментальной вероятности.

Например, чтобы определить экспериментальную вероятность отказа жесткого диска, можно получить результаты опроса других пользователей, имеющих опыт работы с жестким диском такого же типа. Результаты подобного гипотетического опроса приведены в табл. 4.4.

Таблица 4.4. Гипотетические данные о продолжительности работы жесткого диска до наступления отказа

Общее процентное соотношение отказавших жестких дисков	Продолжительность эксплуатации в часах
10	100
25	250
50	500
75	750
99	1000

Допустим, что жесткий диск проработал 750 часов. На основании этого можно логическим путем вывести, что существует 75%-ная вероятность того, что завтра произойдет его отказ. Следует отметить, что это значение, равное 75%, получено с помощью индукции, а не дедукции. В идеальной игре любые ситуации полностью повторяются, а жесткие диски не идеальны, поэтому не могут точно совпадать друг с другом. Различия между жесткими дисками обнаруживаются в составе используемых материалов, в результатах контроля качества, в условиях окружающей среды и эксплуатации, которые влияют на жесткий диск и его долговечность. Гораздо проще изготовить с меньшими допусками такие простые предметы, как игральные кости или игральные карты, чем такое сложное оборудование, как жесткие диски.

К категории экспериментальных вероятностей относится еще один тип данных, которые могут быть получены из статистических таблиц смертности, применяемых компаниями страхования жизни, в которых показаны вероятности смерти людей в зависимости от возраста и пола. Вычисление шансов наступления смерти конкретного человека на основании данных этих таблиц относится к области индукции, поскольку каждый человек уникален. Аналогичная ситуация возникает в области страхования жилья. Допустим, что дома какого-то типа строятся из готовых конструкций и часто сгорают, поэтому для этого типа домов можно индивидуально рассчитать экспериментальную вероятность. Но такая возможность складывается редко, и поэтому применяются экспериментальные вероятности, в которых учитываются аналогичные типы домов. В действительности любой, кто живет в трейлере, может показать эту книгу представителю страховой компании и потребовать скидку, поскольку вероятность возникновения пожара в жилье подобного типа подчиняется определению классической вероятности и может быть

точно вычислена. (Это — еще один пример применимости нашей книги, в данном случае как для получения страховой скидки, так и для уклонения от штрафа за превышение скорости, о чем было сказано в одной из предыдущих глав.)

Применяется также еще один тип вероятности, называемый **субъективной вероятностью**. Предположим, что вас попросили оценить вероятность того, что автомобили со сверхпроводящими электрическими двигателями будут стоить в 2020 году 10 тысяч долларов. Безусловно, в настоящее время отсутствуют данные о стоимости таких автомобилей, поэтому нет возможности экстраполировать подобные значения стоимости, чтобы узнать, является ли цена в 10 тысяч долларов оправданной. С другой стороны, некоторые данные уже позволяют получить такие модели гибридных бензиново-электрических автомобилей, как Toyota Prius, и эти данные можно экстраполировать на все электромобили. А поскольку речь идет о сверхпроводящем двигателе, который является на порядок более эффективным, то появляется возможность применить хоть какую-то разумную экстраполяцию.

Понятие субъективной вероятности распространяется на события, которые не являются воспроизводимыми и не имеют исторической основы, с помощью которой можно было бы осуществлять экстраполяцию. Такую ситуацию можно сравнить с бурением нефтяной скважины на новой площадке. Но оценка субъективной вероятности, сделанная экспертом, лучше по сравнению с полным отсутствием оценки и обычно является очень точной (так как в противном случае эксперт недолго оставался бы экспертом).

Субъективная вероятность — это фактически убеждение, или мнение, выраженное как вероятность, а не объективное значение вероятности, основанное на аксиомах или эмпирических измерениях. Убеждения и мнения экспертов играют важную роль в экспертных системах, о чем будет сказано ниже в данной главе. Итоговые сведения о различных типах вероятностей приведены в табл. 4.5.

4.7 Сложные вероятности

Вероятности сложных событий могут быть вычислены путем анализа соответствующих им выборочных пространств. В качестве очень простого примера рассмотрим вероятность такого броска игральной кости, в котором выпадает четное количество очков, делимое на три без остатка. Эти условия могут быть представлены с помощью диаграмм Венна для следующих множеств в выборочном пространстве результатов бросков игральной кости, как показано на рис. 4.5:

$$A = \{2, 4, 6\}$$
$$B = \{3, 6\}$$

Таблица 4.5. Типы вероятностей

Обозначение	Формула	Характерные особенности
Априорная (классическая, теоретическая, математическая, симметричная, равновозможная, равноправдоподобная)	$P(E) = \frac{W}{N}$, где W — количество результатов события E из общего количества возможных результатов N	Повторяющиеся события; равновероятные результаты; известна точная математическая форма. Не основана на опыте; известны все возможные события и результаты
Апостериорная (экспериментальная, эмпирическая, научная, основанная на определении относительной частоты, статистическая)	$P(E) = \lim\limits_{N\to\infty} \frac{f(E)}{N}$, где $f(E)$ — частота f, с которой событие E наблюдалось в общем количестве результатов N, $P(E) \approx \frac{f(E)}{N}$	Повторяющиеся события, изучение которых осуществляется на основе опыта. Приближенное вычисление по результатам конечного количества экспериментов; точная математическая форма неизвестна
Субъективная (индивидуальная)	См. раздел 4.12	Неповторяющиеся события; точная математическая форма неизвестна; применение метода определения относительной частоты невозможно; определяется на основе мнения, опыта, суждения, убеждения эксперта

Обратите внимание на то, что пересечение множеств A и B представляет собой следующее:

$$A \cap B = \{6\}$$

Сложная вероятность выпадения четного количества очков, делимого на три, определяется приведенным ниже выражением, в котором n — количество элементов в множествах; S — выборочное пространство.

$$P(A \cap B) = \frac{n(A \cap B)}{n(S)} = \frac{1}{6}$$

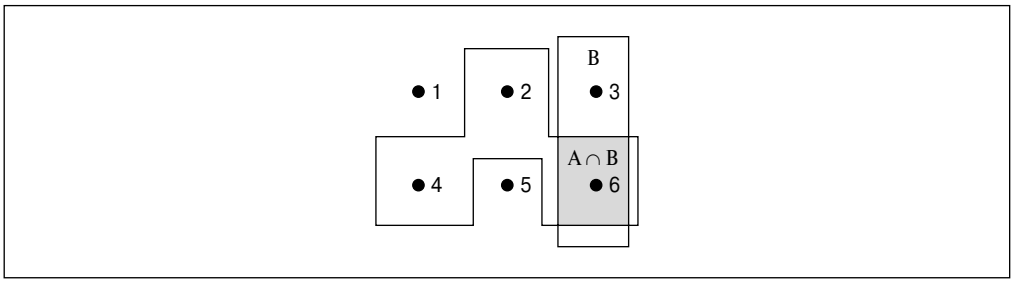


Рис. 4.5. Сложная вероятность броска одной игральной кости, который приводит к получению четного количества очков, делимого на три

События, никоим образом не влияющие друг на друга, называются **независимыми событиями**. Для двух независимых событий A и B вероятность представляет собой произведение отдельных вероятностей. Такие события A и B называются **попарно независимыми**:

$$P(A \cap B) = P(A)P(B)$$

Два события называются **стохастически независимыми** событиями тогда и только тогда, когда для них верна приведенная выше формула. Термин *стохастический* происходит от греческого слова, означающего "предположение". Этот термин обычно используется как синоним термина *вероятностный*. Таким образом, стохастический эксперимент имеет вероятностный результат, в отличие от результата детерминированного эксперимента, не являющегося случайным. В качестве стохастического эксперимента можно попытаться последовательно вступить в брак и развестись с 10 людьми, оказываясь при этом либо в счастливом, либо в несчастливом браке. С другой стороны, попытка вступить в брак одновременно с 10 людьми приводит в детерминированное состояние тюремного заключения.

Можно было бы предположить, что для трех независимых событий формула вычисления вероятности должна быть такой:

$$P(A \cap B \cap C) = P(A)P(B)P(C)$$

Но, к сожалению, законы жизни и вероятности не столь просты. Формула вычисления вероятностей N **взаимно независимых** событий требует решения 2^N уравнений. Это требование подытоживается в следующем уравнении, звездочки в котором означают, что должна быть учтена каждая комбинация из всех событий и их дополнений:

$$P(A_1^* \cap A_2^* \cdots \cap A_N^*) = P(A_1^*)P(A_2^*) \ldots P(A_N^*)$$

При наличии трех событий приведенное выше уравнение для вероятности взаимно независимых событий требует решения всех следующих уравнений:

$$P(A \cap B \cap C) = P(A)P(B)P(C)$$
$$P(A \cap B \cap C') = P(A)P(B)P(C')$$
$$P(A \cap B' \cap C) = P(A)P(B')P(C)$$
$$P(A \cap B' \cap C') = P(A)P(B')P(C')$$
$$P(A' \cap B \cap C) = P(A')P(B)P(C)$$
$$P(A' \cap B \cap C') = P(A')P(B)P(C')$$
$$P(A' \cap B' \cap C) = P(A')P(B')P(C)$$
$$P(A' \cap B' \cap C') = P(A')P(B')P(C')$$

Как показано в условиях задачи 4.6, недостаточно обеспечить попарную независимость каждого из двух событий, чтобы гарантировать взаимную независимость всех событий.

Возвращаясь к примеру с игральной костью, отметим, что события, связанные с выпадением четного количества очков и количества очков, делимого на три, определенно влияют друг на друга, поэтому данный эксперимент не является стохастическим. С другой стороны, как показано ниже, вероятность выпадения четного количества очков на одной игральной кости и количества очков, делимого на три, на другой, является стохастической.

$$P(A \cup B) = P(A)P(B) = \frac{3}{6} \cdot \frac{2}{6} = \frac{1}{6}$$

Теперь рассмотрим случай с объединением событий, $P(A \cup B)$. Определим n как функцию, возвращающую количество элементов в множестве. Если мы сложим количество элементов в множестве A с количеством элементов в множестве B и разделим на общее количество элементов в выборочном пространстве, $n(S)$, то полученный результат, если множества пересекаются, будет слишком большим.

$$P(A \cup B) = \frac{n(A) + n(B)}{n(S)} = P(A) + P(B) \quad (1)$$

Как показано на рис. 4.5, применение этой формулы приводит к получению следующего результата:

$$P(A \cup B) = \frac{3+2}{6} = \frac{5}{6}$$

Тем не менее можно легко убедиться в том, что в объединении содержатся только четыре элемента, поскольку имеет место такое выражение:

$$A \cup B = \{2, 3, 4, 6\}$$

Проблема состоит в том, что при сложении чисел $n(A)$ и $n(B)$ количество элементов в пересечении множеств $\{6\}$ засчитывается дважды, поскольку это пересечение принадлежит и к A, и к B.

Для получения правильной формулы достаточно вычесть лишнее значение, определяющее вероятность пересечения множеств:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) \tag{2}$$

Эта формула применительно к рассматриваемому примеру с игральной костью сводится к следующему выражению:

$$P(A \cup B) = \frac{3}{6} + \frac{2}{6} - \frac{1}{6} = \frac{4}{6} = \frac{2}{3}$$

Формула (1) является правильной, если множества не пересекаются, поэтому не имеют общих элементов. Таким образом, формула (1) представляет собой частный случай формулы (2), называемой **аддитивным законом**. Аддитивный закон можно также вывести как теорему с использованием трех аксиом вероятностей, описанных выше в данной главе. Аддитивный закон для трех событий принимает следующий вид:

$$\begin{aligned} P(A \cup B \cup C) = {} & P(A) + P(B) + P(C) \\ & - P(A \cap B) - P(A \cap C) - P(B \cap C) \\ & + P(A \cap B \cap C) \end{aligned}$$

Из указанных аксиом можно также вывести другие законы для таких случаев, в которых к событиям A и B применяется операция NEITHER NOR (НЕ ИЛИ) или XOR (исключительное ИЛИ). Значения вероятностей, которые могут быть определены с помощью этих законов, соответствуют сложным вероятностям, характеризующимся тем, что, во-первых, нет необходимости проводить эксперименты для определения каждой возможной комбинации вероятностей, и, во-вторых, не требуется отдельно подсчитывать элементы крупных выборочных пространств.

4.8 Условные вероятности

События, не являющиеся взаимоисключающими, влияют друг на друга. Узнав о том, что произошло одно событие, мы можем оказаться вынужденными пересмотреть оценку вероятности возникновения другого события.

Мультипликативный закон

Вероятность возникновения события A, определяемая с учетом того, что произошло событие B, называется **условной вероятностью** и обозначается $P(A \mid B)$.

Условная вероятность определяется следующим образом:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \text{ для } P(B) \neq 0$$

Вероятность $P(B)$ — это априорная вероятность, определяемая до того, как станет известной какая-либо дополнительная информация. Априорную вероятность, применяемую в связи с использованием условной вероятности, иногда называют **безусловной вероятностью**, или **абсолютной вероятностью**.

Этому определению условной вероятности можно дать интуитивное объяснение, обратившись к примеру на рис. 4.6, где показано выборочное пространство из восьми событий. Согласно рис. 4.6, вероятности могут быть вычислены как отношения количества событий, $n(A)$ или $n(B)$, к общему количеству событий в выборочном пространстве, $n(S)$, следующим образом:

$$P(A) = \frac{n(A)}{n(S)} = \frac{4}{8}$$
$$P(B) = \frac{n(B)}{n(S)} = \frac{6}{8}$$

Если станет известно, что произошло событие B, то выборочное пространство сократится и будет включать только элементы, относящиеся к B:

$$n(S) = 6$$

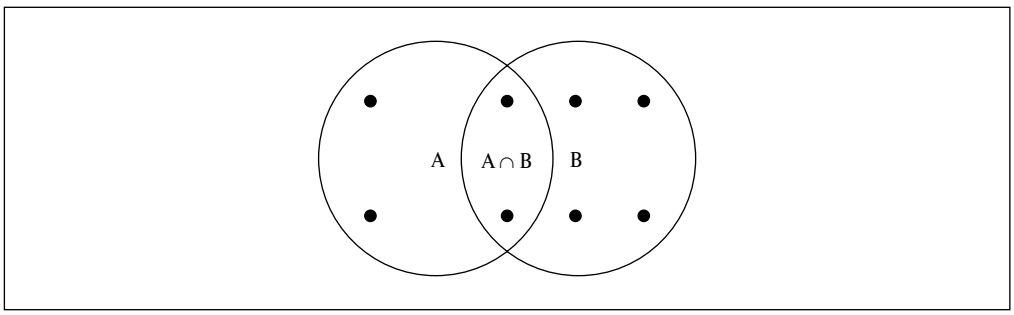


Рис. 4.6. Выборочное пространство для двух пересекающихся событий

Поскольку произошло событие B, то должны рассматриваться только те события, относящиеся к A, которые связаны с B:

$$P(A \mid B) = \frac{n(A \cap B)}{n(B)} = \frac{2}{6}$$

Чтобы выразить этот результат в терминах вероятностей, достаточно разделить числитель и знаменатель приведенного выше выражения на $n(S)$:

$$P(A \mid B) = \frac{\dfrac{n(A \cap B)}{n(S)}}{\dfrac{n(B)}{n(S)}} = \frac{P(A \cap B)}{P(B)}, \text{ для } P(B) \neq 0$$

Таким образом, **мультипликативный закон** для вероятностей двух событий определяется следующей формулой:

$$P(A \cap B) = P(A \mid B)P(B)$$

А эта формула эквивалентна формуле

$$P(A \cap B) = P(B \mid A)P(A)$$

Мультипликативный закон для трех событий выражается следующим образом:

$$P(A \cap B \cap C) = P(A \mid B \cap C)P(B \mid C)P(C)$$

С другой стороны, обобщенный мультипликативный закон можно представить так:

$$\begin{aligned} P(A_1 \cap A_2 \cap \cdots \cap A_N) = {} & P(A_1 \mid A_2 \cap \cdots \cap A_N)\cdot \\ & P(A_2 \mid A_3 \cap \cdots \cap A_N)\cdot \\ & \ldots P(A_{N-1} \mid A_N)P(A_N) \end{aligned}$$

В качестве примера применения условных вероятностей рассмотрим приведенные в табл. 4.6 гипотетические вероятности отказа жесткого диска при эксплуатации диска модели X в течение одного года.

Таблица 4.6. Гипотетические вероятности отказа жесткого диска в течение одного года

	Относящиеся к модели X (обозначаются X)	**Не относящиеся к модели X (обозначаются X')**	**Итоговые данные по строке**
Отказ, C	0,6	0,1	0,7
Отсутствие отказа C'	0,2	0,1	0,3
Итоговые данные по столбцу	0,8	0,2	1,0

Вероятности 0,6, 0,1, 0,2 и 0,1, приведенные в средней части таблицы, называются **внутренними вероятностями** и представляют пересечения событий.

Суммы вероятностей по строкам и столбцам обозначаются как “Итоговые данные...” и называются **маргинальными вероятностями**, поскольку лежат на полях (margin) таблицы.

В табл. 4.7 и 4.8 приведены более подробные сведения о комбинациях множеств элементарных событий и вероятностей, а на рис. 4.7 показана диаграмма Венна пересечения выборочных пространств. Как показано в табл. 4.8, сумма итоговых вероятностей по строкам и столбцам равна 1.

Таблица 4.7. Интерпретация условной вероятности с учетом множеств элементарных событий

	X	X'	**Итоговые данные по строкам**
C	$C' \cap X$	$C \cap X'$	$C = (C \cap X) \cup (C \cap X')$
C'	$C' \cap X$	$C' \cap X'$	$C' = (C' \cap X) \cup (C' \cap X')$
Итоговые данные по столбцам	$X = (C' \cap X) \cup (C \cap X)$	$X' = (C' \cap X') \cup (C \cap X')$	S (Выборочное пространство)

Таблица 4.8. Вероятностная интерпретация двух множеств элементарных событий

	X	X	**Итоговые данные по строкам**
C	$P(C \cap X)$	$P(C \cap X')$	$P(C)$
C'	$P(C' \cap X)$	$P(C' \cap X')$	$P(C')$
Итоговые данные по столбцам	$P(X)$	$P(X')$	1.0

С помощью табл. 4.8 могут быть рассчитаны вероятности всех событий. Некоторые из полученных при этом значений вероятности описаны ниже.

(1) Вероятность аварийного отказа как для модели X, так и для модели, отличной от X (выборочное пространство):

$$P(C) = 0.7$$

(2) Вероятность отсутствия отказа для данного выборочного пространства:

$$P(C') = 0.3$$

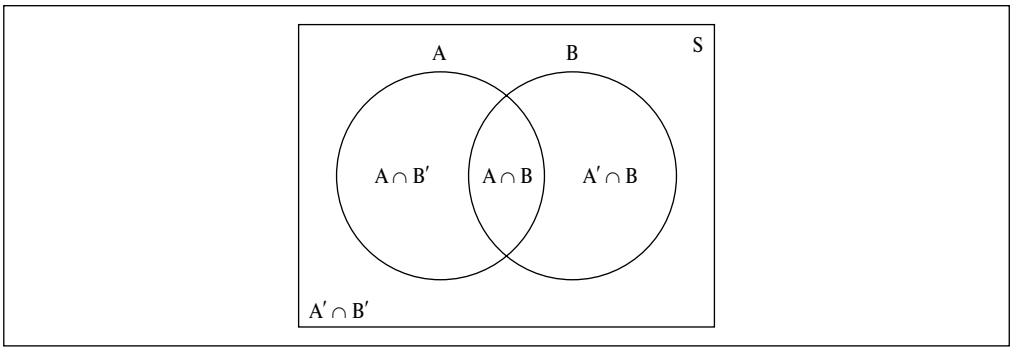


Рис. 4.7. Интерпретация условных вероятностей двух множеств элементарных событий в выборочном пространстве

(3) Вероятность использования модели X:

$$P(X) = 0.8$$

(4) Вероятность использования модели, отличной от X:

$$P(X') = 0.2$$

(5) Вероятность отказа и использования модели X:

$$P(C \cap X) = 0.6$$

(6) Вероятность отказа при условии, что используется модель X:

$$P(C \mid X) = \frac{P(C \cap X)}{P(X)} = \frac{0.6}{0.8} = 0.75$$

(7) Вероятность отказа при условии, что не используется модель X:

$$P(C \mid X') = \frac{P(C \cap X')}{P(X')} = \frac{0.1}{0.2} = 0.50$$

При чтении описаний вероятностей (5) и (6) может сложиться впечатление, что они имеют одинаковый смысл. Но случай (5) соответствует просто пересечению двух событий, а случай (6) определяет условную вероятность. Смысл вероятности пересечения двух событий, (5), состоит в следующем:

```
Если бы выбор жесткого диска осуществлялся случайным
образом, то в 0,6 случая это был бы жесткий диск модели
X и этот жесткий диск был бы отказавшим
```

Иными словами, в случае (5) используются выборки из совокупности событий, относящихся к жестким дискам. Одни из этих событий соответствуют тому, что эксплуатировался жесткий диск модели X и произошел отказ (0.6), другие — что эксплуатировался жесткий диск, не относящийся к модели X, и произошел отказ (0.1), третьи — что эксплуатировался жесткий диск модели X и не произошел отказ (0.2), и четвертые — что эксплуатировался жесткий диск, не относящийся к модели X, и не произошел отказ (0.1).

В отличие от этого, условная вероятность, рассматриваемая в случае (6), имеет совсем другой смысл:

```
Если бы был выбран жесткий диск модели X, то в 0,75 случая
этот жесткий диск был бы отказавшим
```

Обратите внимание на то, что при вычислении условной вероятности мы выбираем только интересующие нас элементы (относящиеся к модели X) и рассматриваем множество этих элементов как новое выборочное пространство.

Если любое из приведенных ниже уравнений является справедливым, то события A и B называются **независимыми**.

$$P(A \mid B) = P(A) \text{ или}$$
$$P(B \mid A) = P(B) \text{ или}$$
$$P(A \cap B) = P(A)P(B)$$

При этом, если справедливо одно из этих уравнений, таковыми являются и другие.

Теорема Байеса

Условная вероятность $P(A \mid B)$ определяет вероятность события A с учетом того, что произошло событие B. Задача, противоположная задаче вычисления условной вероятности, состоит в определении **обратной вероятности**, которая показывает вероятность предыдущего события с учетом того, что произошло последующее. На практике с вероятностью такого типа приходится встречаться довольно часто, например, при проведении медицинской диагностики или диагностики оборудования, в которой обнаруживаются симптомы, а задача заключается в том, чтобы найти наиболее вероятную причину. Для решения этой задачи применяется **теорема Байеса**, которую иногда называют формулой Байеса, правилом Байеса или законом Байеса в честь британского священнослужителя и математика XVIII века Томаса Байеса. Байесовская теория в наши дни широко используется во многих приложениях. В действительности байесовская логика применяется в знакомых многим приложениях Office Assistant и Technical Troubleshooter Help, которые по умолчанию предоставляются в комплекте с программой Microsoft Office и операционной системой Windows; эти приложения обеспечивают более

качественный поиск причин неисправности и лучшую помощь, чем при использовании простых деревьев решений [32].

В качестве примера применения теоремы Байеса рассмотрим, как с ее помощью рассчитать вероятности отказа жестких дисков. Согласно определению условной вероятности, показанному в случае (6), вероятность того, что отказ жесткого диска модели X произойдет в течение одного года, равна 75%, а согласно данным, относящимся к случаю (7), вероятность отказа в течение одного года жесткого диска, не относящегося к модели X, равна 50%. Противоположная задача состоит в следующем: предположим, что имеется жесткий диск и неизвестна его модель. Какова вероятность того, что он относится к модели X или к модели, отличной от X, если он отказал?

Подобная ситуация, в которой пользователь фактически не знает, какая модель жесткого диска установлена на его компьютере, встречается постоянно, поскольку изготовители компьютеров очень редко занимаются изготовлением жестких дисков. Вместо этого многие изготовители компьютеров покупают жесткие диски у изготовителей комплектного оборудования (Original Equipment Manufacturer — OEM), устанавливают эти жесткие диски в корпуса своих компьютеров и продают под собственной маркой. Один и тот же изготовитель компьютеров может время от времени переходить на другие модели жестких дисков из-за того, что предложение другого изготовителя комплектного оборудования становится более выгодным, но его собственная модель остается неизменной.

Если принято предположение, что произошел отказ жесткого диска, вероятность того, что он относится к модели X, можно вычислить с использованием формулы условной вероятности и результатов, полученных в случаях (1) и (5), следующим образом:

$$P(X \mid C) = \frac{P(C \cap X)}{P(C)} = \frac{0.6}{0.7} = \frac{6}{7}$$

Еще один вариант состоит в том, что можно применить мультипликативный закон к числителю и взять данные, полученные в случаях (1), (3) и (6), как показано ниже.

$$P(X \mid C) = \frac{P(C \mid X)P(X)}{P(C)} = \frac{(0.75)(0.8)}{0.7} = \frac{0.6}{0.7} = \frac{6}{7}$$

Вероятность типа $P(X \mid C)$ называется **обратной**, или апостериорной вероятностью, определяющей вероятность того, что отказавший диск относится к модели X. Дерево решений для задачи определения вероятностей отказов дисков приведено на рис. 4.8. Узлы, обозначенные прямоугольниками, соответствуют **действиям**, или решениям, а узлы, отмеченные кружками, показывают **события**, или проявления. Априорные вероятности представляют собой вероятности, полученные до проведения экспериментов по определению количества отказов.

С другой стороны, апостериорные (или обратные) вероятности представляют собой вероятности, определяемые после завершения экспериментов. Апостериорные вероятности позволяют пересматривать значения априорных вероятностей для получения более точных результатов.

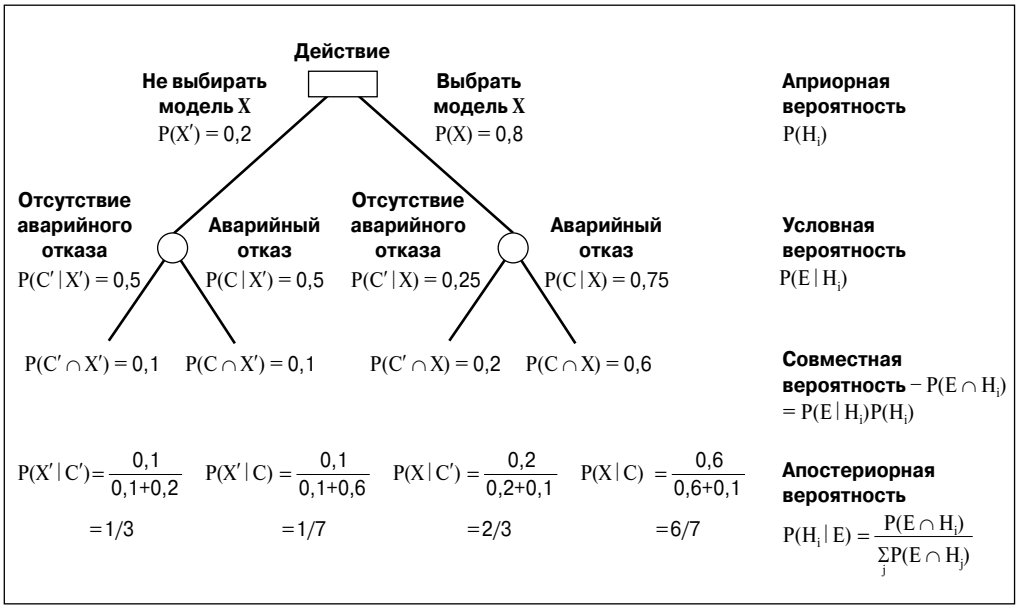


Рис. 4.8. Дерево решений для задачи с отказами жестких дисков

Общая форма теоремы Байеса может быть записана в терминах событий, E, и гипотез (предположений), H, в виде следующих альтернативных вариантов:

$$P(H_i \mid E) = \frac{P(E \cap H_i)}{\sum_j P(E \cap H_j)} = \\ = \frac{P(E \mid H_i)P(H_i)}{\sum_j P(E \mid H_j)P(H_j)} = \\ = \frac{P(E \mid H_i)P(H_i)}{P(E)}$$

4.9 Гипотетические рассуждения и обратная индукция

Теорема Байеса широко используется для анализа деревьев решений в экономике и общественных науках. Метод **байесовского принятия решений** применя-

ется также в экспертной системе PROSPECTOR при определении перспективных площадок для разведки полезных ископаемых. Система PROSPECTOR приобрела широкую известность как первая экспертная система, с помощью которой было открыто ценное месторождение молибдена, имеющее стоимость 100 миллионов долларов.

В качестве примера байесовского принятия решений в условиях неопределенности рассмотрим задачу разведки месторождений нефти. Первоначально геологоразведывательная компания должна решить, каковы шансы успешного обнаружения нефти. Если отсутствуют какие-либо свидетельства за или против наличия нефти, то геологоразведывательная компания может присвоить такие субъективные априорные вероятности наличия и отсутствия нефти (O):

$$P(O) = P(O') = 0.5$$

Принято использовать такое выражение, что присваивание вероятностей, равномерно распределенных между возможными результатами в условиях отсутствия свидетельств, осуществляется **в безнадежном положении** (in desperation). Термин *в безнадежном положении* не обязательно означает, что геологоразведывательная компания (обязательно) находится в положении, из которого нет выхода. Это — просто технический термин для обозначения непредубежденного априорного присваивания вероятностей. Безусловно, геологоразведывательная компания может считать, что шансы обнаружения нефти лучше чем 50 на 50, и в связи с этим руководствоваться следующим предположением:

$$P(O) = 0.6$$
$$P(O') = 0.4$$

Очень важным средством разведки месторождений нефти и полезных ископаемых являются **сейсмические исследования**. При использовании этого метода с помощью взрывчатых веществ или механических средств создаются звуковые импульсы, движущиеся в глубины Земли. Отраженные звуковые волны обнаруживаются микрофонами, расположенными в разных местах. Регистрация времени прибытия импульсов и наблюдение за искажениями формы звуковой волны позволяют определить наличие перспективных геологических структур и возможность залегания месторождений нефти и полезных ископаемых. Но, к сожалению, сейсмические исследования не обеспечивают 100%-ную точность. На распространение звуковых волн могут повлиять геологические структуры некоторых типов, поэтому отчеты о проведенных испытаниях будут свидетельствовать о наличии нефти, в то время как фактически месторождение отсутствует (ложно положительный результат). Аналогичным образом, результаты испытания могут указывать на отсутствие нефти, тогда как действительно месторождение нефти имеется

(ложно отрицательный результат). Предположим, что последние результаты сейсмических исследований привели к получению показанных ниже условных вероятностей, в которых + обозначает положительный результат, а − соответствует отрицательному результату. Обратите внимание на то, что в данном случае рассматриваются условные вероятности, поскольку причина (наличие или отсутствие нефти) должна была возникнуть до того, как был получен результат (имеются в виду данные испытаний). Апостериорная вероятность соответствует цепи событий от результата (данные испытаний) обратно к причине (наличие или отсутствие нефти). Вообще говоря, условная вероятность охватывает события, рассматриваемые во времени в прямом направлении, а апостериорная вероятность — события, происходящие во времени в обратном направлении.

$$P(+ \mid O) = 0.8$$
$$P(- \mid O) = 0.2 \qquad \text{(ложно отрицательный результат)}$$
$$P(+ \mid O') = 0.1 \qquad \text{(ложно положительный результат)}$$
$$P(- \mid O') = 0.9$$

Как показано на рис. 4.9, с применением априорных и условных вероятностей может быть построено дерево начальных вероятностей. Кроме того, показаны совместные вероятности, вычисленные на основании априорных вероятностей и условных вероятностей.

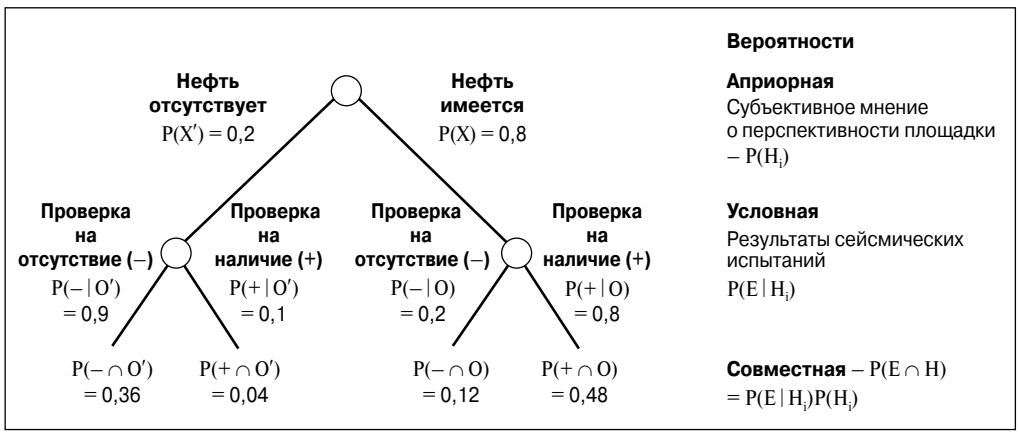


Рис. 4.9. Дерево начальных вероятностей для задачи поиска нефти

После этого для вычисления суммарной вероятности результатов испытаний + и − может использоваться аддитивный закон:

$$P(+) = P(+ \cap O) + P(+ \cap O') = 0.48 + 0.04 = 0.52$$
$$P(-) = P(- \cap O) + P(- \cap O') = 0.12 + 0.36 = 0.48$$

Как показано на рис. 4.10, $P(+)$ и $P(-)$ — это безусловные вероятности, которые теперь могут использоваться для вычисления апостериорных вероятностей наличия месторождения на рассматриваемой площадке. Например, как $P(O' \mid -)$ обозначается апостериорная вероятность отсутствия нефти в исследуемом районе, вычисленная на основании отрицательных результатов исследований. После этого вычисляются значения совместной вероятности. Обратите внимание на то, что значения совместной вероятности, показанные на рис. 4.10, являются такими же, как и на рис. 4.9. Такой пересмотр вероятностей необходим для получения качественных результатов, если вслед за выдвижением первоначальных оценок вероятностей (или предположений) поступает экспериментальная информация, такая как данные сейсмических исследований.

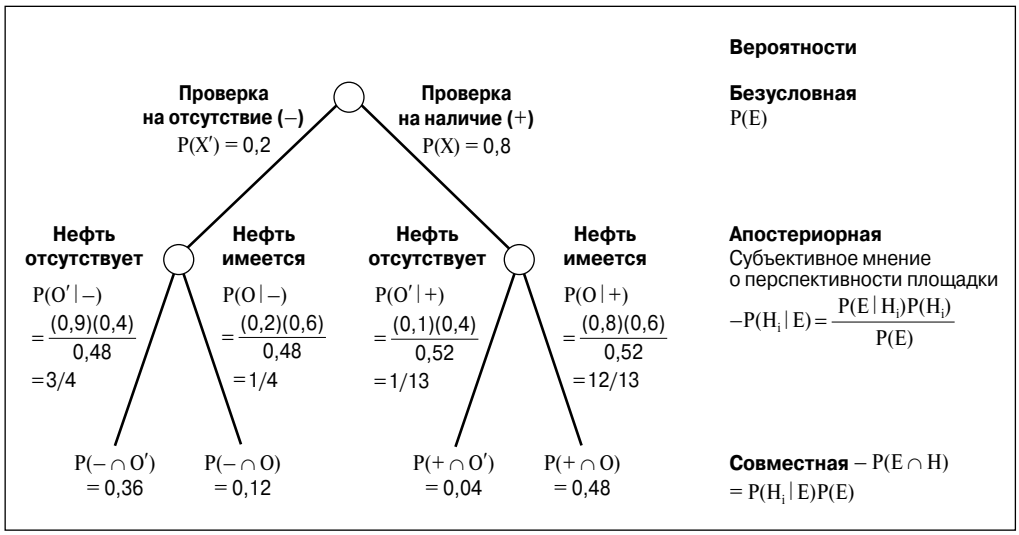


Рис. 4.10. Пересмотренное дерево вероятностей для задачи поиска нефти

На рис. 4.11 показан первоначальный вариант байесовского дерева принятия решений, в котором используются данные, приведенные на рис. 4.10. **Выигрыш**, показанный в нижней части дерева, является положительным, если получена прибыль, и отрицательным, если получен убыток. Предполагаемые денежные суммы показаны в табл. 4.9.

Таблица 4.9. Таблица выигрышей для задачи разведки нефти

Выигрыш	Сумма
В случае успеха сдача месторождения нефти в аренду	1 миллион долларов
Расходы на разведывательное бурение	−200 тысяч долларов
Расходы на сейсмическую разведку	−50 тысяч долларов

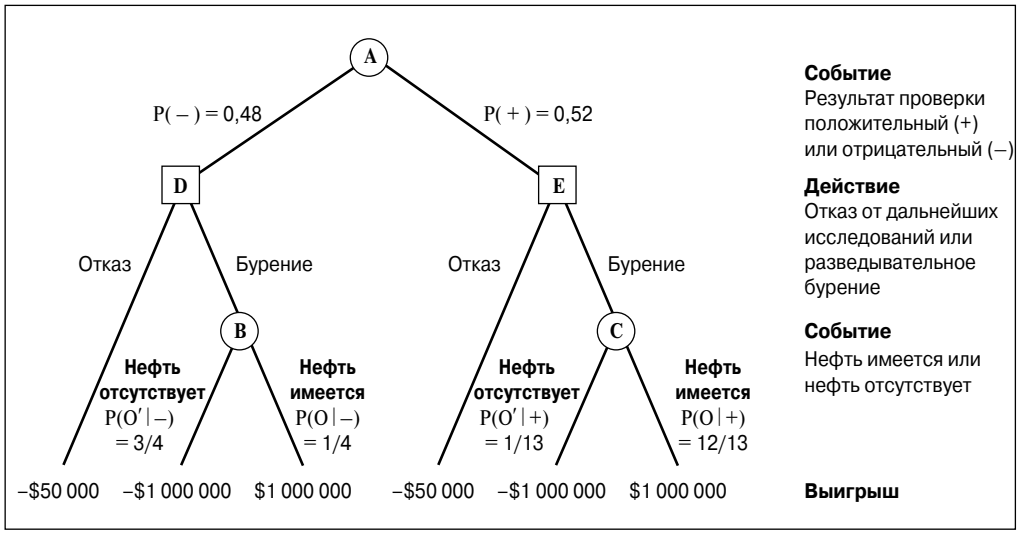


Рис. 4.11. Первоначальное байесовское дерево решений для задачи поиска нефти

Таким образом, если месторождение нефти будет найдено, то выигрыш в долларах составит $1\,000\,000 - 200\,000 - 50\,000 = 750\,000$; если будет решено отказаться от поиска после получения результатов сейсмических исследований, то выигрыш составит $-50\,000$ долларов; если же будут проведены сейсмические исследования и бурение, но месторождение нефти не будет найдено, то выигрыш в долларах составит $-200\,000 - 50\,000 = -250\,000$. Обратите внимание на то, что этот выигрыш в долларах не может составить $-1\,000\,000 - 200\,000 - 50\,000 = -1\,250\,000$, если вы не сможете воспользоваться лазейкой в налоговом законодательстве (а это не исключено, поскольку при наличии в Конгрессе хорошего лоббиста можно найти очень много удобных дополнений к законам о налогах).

Для того чтобы геологоразведывательная компания могла принять наилучшее решение, необходимо рассчитать **ожидаемый выигрыш** в узле события A. Ожидаемый выигрыш представляет собой сумму, которую может заработать геологоразведывательная компания, придерживаясь наилучшего способа действий. Для вычисления ожидаемого выигрыша в начальном узле, A, необходимо пройти по дереву решений в обратном направлении, от листовых узлов. В терминах теории вероятностей такой процесс называется **обратной индукцией**. Это означает, что для вычисления ожидаемого выигрыша или достижения желаемой цели необходимо проводить рассуждения в обратном направлении для поиска причин, которые приведут нас к цели.

Ожидаемый выигрыш в узле события определяется как сумма выигрышей в дочерних узлах, умноженная на вероятности, ведущие к этим выигрышам, как показано ниже.

Ожидаемый выигрыш в узле C

673 077 долларов $= (750\,000 \text{ долларов})(12/13) - (250\,000 \text{ долларов})(1/13)$

Ожидаемый выигрыш в узле B

0 долларов $= (750\,000 \text{ долларов})(1/4) - (250\,000 \text{ долларов})(3/4)$

В узле действия E необходимо сделать выбор между ожидаемым выигрышем для того случая, когда поиск нефти прекращается ($-50\,000$ долларов), и ожидаемым выигрышем, достигаемым в случае проведения бурения ($846\,153$ доллара). Поскольку значение $846\,153$ долларов больше чем $-50\,000$ долларов, можно прийти к логическому выводу, что это — лучший способ действий, и записать полученный выигрыш рядом с узлом E. Путь, ведущий к узлу, в котором происходит прекращение дальнейших действий, **отсекается**, или **исключается**. Для этого поперек этого пути проставляется символ $=$, указывающий, что дальнейшее его изучение проводиться не будет (рис. 4.12). Аналогичным образом, в узле действия D выбор наилучшего способа действий сводится к сравнению значений ожидаемого выигрыша, равных $-50\,000$ долларов и $-500\,000$ долларов, что влечет за собой выбор значения $-50\,000$ долларов, и это значение записывается рядом с узлом D.

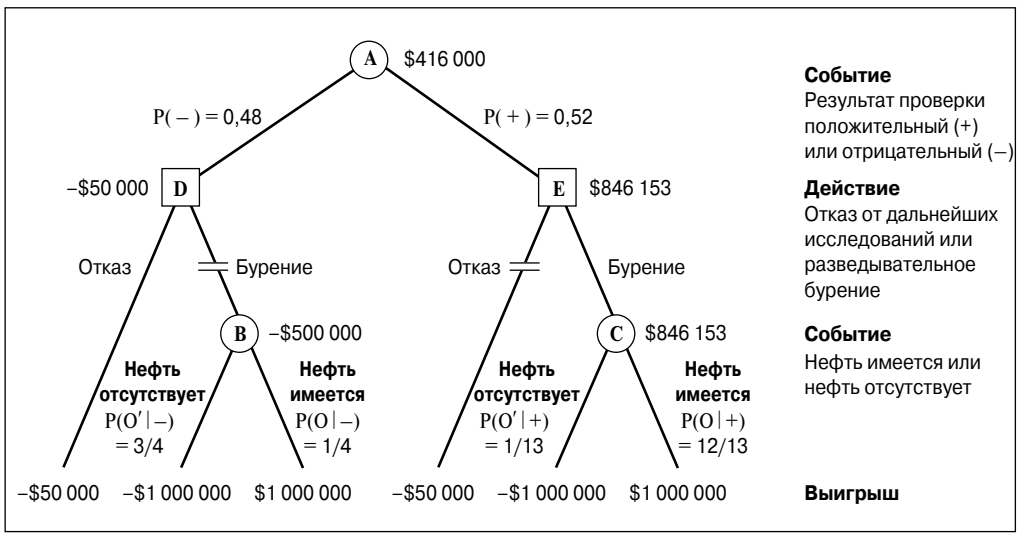


Рис. 4.12. Полное байесовское дерево решений для задачи разведки нефти, в которой используется обратная индукция

Наконец, ожидаемый выигрыш в начальном узле определяется, как показано ниже, и полученное значение записывается рядом с узлом A.

Ожидаемый выигрыш в узле A

350 000 долларов $= (673\,077 \text{ долларов})(0.52) - (0 \text{ долларов})(0.48)$

Ни один из путей, ведущих из узла A, не отсекается, поскольку уже было решено выполнить сейсмическое исследование. Чтобы принять решение по проведению сейсмического исследования, потребовалось бы развернуть дерево решений в обратном направлении перед узлом A. Следует учитывать, что при проведении обратной индукции рассуждения проводятся против хода времени, для того чтобы действия, намеченные на будущее, были оптимальными.

В таблице выигрышей (см. табл. 4.9) данным конкретным вариантам соответствует тривиальное решение, которое должна была бы принять геологоразведывательная компания в отношении узла действия D, поскольку ожидаемый выигрыш составляет 0 долларов. Но на этом этапе в игру вступают отвага, удача и опыт. Специалисты геологоразведывательной компании должны просто доверять своему чутью и принять в узле D решение о разведывательном бурении, несмотря на отрицательные результаты сейсмических исследований и на значительный шанс отсутствия нефти. Иногда в подобных случаях компания выигрывает, а иногда проигрывает. Но в таких ситуациях важно помнить одну мудрую рекомендацию — всегда ставьте на кон не свои деньги, а чужие! По мере повышения шансов на получение ожидаемого выигрыша в результате успешного бурения нефтяной скважины возрастают и значения выигрышей в узлах B и D. Например, если перспективная нефтяная скважина принесет выигрыш в $1\,250\,000$ долларов вместо $1\,000\,000$ долларов, ожидаемый выигрыш в узлах B и D составит $62\,500$ долларов, а это все еще меньше стоимости осуществления решения пробурить скважину, равной $200\,000$ долларов, но гораздо лучше по сравнению с предыдущим ожидаемым выигрышем, который составлял 0 долларов. Тем самым подтверждается правило, основанное на здравом смысле, согласно которому, чем больше приходится за что-то платить, тем больше выигрыш, который может быть получен в обмен на эти расходы. Но следует отметить, что даже если бы цена на нефть достигла миллиардов долларов, риски, связанные с существующими вероятностями, оставались бы теми же. (Изменяется только выигрыш, но, как знают многие "великие комбинаторы", кто не рискует, тот не пьет шампанское.)

Рассматриваемое дерево решений представляет собой пример гипотетических рассуждений, или ситуаций, в которых важную роль играют вопросы "что, если". Исследуя альтернативные способы действий, мы можем отсекать пути, не ведущие к оптимальным выигрышам. В инструментальных средствах экспертных систем и в байесовском программном обеспечении некоторых типов предусмотрены развитые механизмы проведения гипотетических рассуждений и исключения бесперспективных путей. Дополнительные сведения приведены в разделе "Программные ресурсы" приложения Ж. В частности, в настоящее время можно легко найти бесплатные или коммерческие байесовские инструментальные средства, позволяющие без особого труда строить в графической форме байесовские и вероятностные деревья и проводить с ними эксперименты в различных сценариях поиска ответов на вопросы "что, если". Очень развитые способы, обеспечиваю-

щие проведение расчетов в условиях неопределенности, предусмотрены в таких электронных таблицах, как Microsoft Excel, а проведение поиска в Web позволяет найти много макрокоманд, еще более превосходящих возможности встроенных функций для работы с классическими вероятностями и статистическими показателями, которые входят в состав Excel. Кроме того, удобные графические средства электронных таблиц позволяют легко представлять визуально сложные данные.

Дерево решений, приведенное на рис. 4.12, показывает оптимальную стратегию для геологоразведывательной компании. Если результаты сейсмических исследований являются положительными, то в рассматриваемом районе должно быть проведено бурение, а если результаты сейсмических исследований отрицательны, то район необходимо покинуть. Безусловно, этот пример байесовского принятия решений является очень простым, но он показывает, какого типа рассуждения применяются в условиях неопределенности. В более сложных случаях, например, касающихся принятия решения о проведении сейсмических исследований, деревья решений могут намного увеличиваться.

4.10 Временные рассуждения и марковские цепи

Рассуждения о событиях, зависящих от времени, называются **временны́ми рассуждениями**. Люди проводят такие рассуждения довольно легко. Но **временны́е события** трудно формализовать так, чтобы временны́е логические выводы мог выполнять компьютер. Вместе с тем экспертные системы, способные рассуждать о таких временны́х событиях, которые, например, связаны с управлением воздушным движением, могли бы принести большую пользу. Экспертные системы, проводящие свои рассуждения во времени, были разработаны в медицине. К ним относится система VM, предназначенная для управления вентиляторами, устанавливаемыми в масках пациентов, чтобы им было легче дышать. К числу других систем относится система CASNET, предназначенная для лечения глаукомы, и система, предоставляющая консультации по применению терапевтических средств на основе наперстянки при лечении больных с сердечными заболеваниями.

Все медицинские системы, кроме VM, перечисленные выше, предназначены для решения гораздо более легкой задачи проведения временны́х рассуждений по сравнению с системой управления воздушным движением, которая должна действовать в **реальном времени**. Большинство экспертных систем не могут функционировать в реальном времени, поскольку этого не позволяет проект машины логического вывода, к тому же требуется большой объем обрабатывающих мощностей. Задача создания экспертной системы, выполняющей значительное количество этапов временны́х логических рассуждений для исследования много-

численных гипотез в реальном времени, является слишком сложной. Тем не менее на основе различных множеств аксиом разработано много вариантов временны́х логик. Различия между разными теориями определяются тем, какой способ применяется для поиска ответов на определенные вопросы. Имеет ли время начало и конец? Протекает ли время непрерывно или дискретно? Можно ли считать, что прошлое только одно, а возможных вариантов будущего много?

Формулировка различных ответов на эти вопросы приводит к разработке различных видов логики. Временна́я логика применяется также в обычных программах, например, для порождения и синхронизации параллельных процессов в программах.

Еще один подход к организации рассуждений во времени состоит в использовании вероятностей. Система, переходящая из одного состояния в другое, может рассматриваться как развивающаяся во времени. Такая система может представлять любые вероятностные явления, такие как изменение цен на акции, распределение голосов избирателей, погодные явления, процессы, происходящие в деловом мире, развитие заболеваний, функционирование оборудования, генетические изменения и т.д. Если процесс перехода системы через последовательность состояний является вероятностным, то он рассматривается как **стохастический процесс**.

Стохастический процесс удобно представлять в форме матрицы переходов. В простом случае с двумя состояниями, S_1 и S_2, матрица переходов может быть представлена в следующей форме, где P_{mn} — вероятность перехода из состояния m в состояние n:

$$\begin{array}{c} \\ \text{Настоящее} \end{array} \begin{array}{c} \text{Будущее} \\ \begin{array}{cc} S_1 & S_2 \end{array} \\ \begin{array}{c} S_1 \\ S_2 \end{array} \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \end{array}$$

Например, предположим, что только 10% из тех, кто в настоящее время использует жесткие диски модели X, купят еще один жесткий диск модели X, когда это потребуется. Кроме того, 60% из тех, кто в настоящее время не использует модели X, купят жесткий диск модели X, когда им потребуется новый жесткий диск (единственным достоинством модели X является хорошая реклама). Какое количество людей купят жесткий диск модели X в течение продолжительного периода времени?

Ниже показана матрица переходов, T, в которой сумма чисел в каждой строке должна быть равна 1.

$$T = \begin{array}{c} \\ X \\ X' \end{array} \begin{array}{c} \begin{array}{cc} X & X' \end{array} \\ \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix} \end{array}$$

Вектор, компоненты которого не являются отрицательными и в сумме составляют 1, называется **вектором вероятностей**. Определению вектора вероятностей соответствует каждая строка матрицы T. Один из способов интерпретации матрицы переходов состоит в использовании диаграммы состояний, как показано на рис. 4.13. Обратите внимание на то, что вероятность системы остаться в состоянии X равна 0.1, вероятность остаться в состоянии X' равна 0.4, вероятность перейти из состояния X в состояние X' равна 0.9, а вероятность перейти из состояния X' в состояние X равна 0.6.

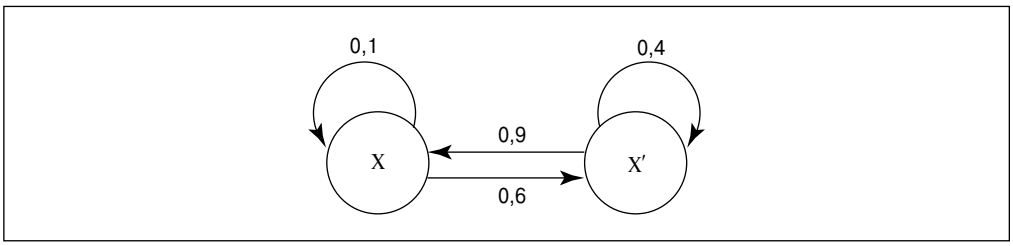


Рис. 4.13. Интерпретация матрицы переходов с помощью диаграммы состояний

Вначале примем предположение, что жесткий диск модели X установлен в компьютерах 80% пользователей. На рис. 4.14, *а* показано дерево вероятностей для нескольких переходов между состояниями, в котором состояния обозначены номером состояния и приведена используемая модель жесткого диска. Обратите внимание на то, как быстро начинает расти дерево. А если бы количество переходов достигло 10, то количество ветвей в дереве увеличилось бы до $2^{10} = 1024$. Альтернативный способ изображения этого дерева состоит в том, что оно может быть представлено в виде решетки, как показано на рис. 4.14, *б*. Преимущество представления в виде решетки состоит в том, что в нем не требуется так много связей, соединяющих состояния.

Вероятность пребывания системы в определенном состоянии может быть выражена с помощью следующей однострочной матрицы, называемой **матрицей состояний**, в которой $P_1 + P_2 + \cdots + P_N = 1$:

$$S = \begin{bmatrix} P_1 & P_2 & \ldots & P_N \end{bmatrix}$$

Первоначально, когда жесткие диски модели X установлены в компьютерах 80% пользователей, матрица состояний имеет такой вид:

$$S_1 = \begin{bmatrix} 0.8 & 0.2 \end{bmatrix}$$

А с течением времени числа в этой матрице изменяются в зависимости от того, какие модели жестких дисков покупают пользователи.

Для того чтобы рассчитать в состоянии 2 количество пользователей, имеющих жесткий диск модели X и не имеющих жесткий диск модели X, достаточно умножить матрицу состояний на матрицу переходов с использованием обычных правил умножения матриц, следующим образом:

$$S_2 = S_1 T$$

Выполнение этой операции приводит к получению таких результатов:

$$\begin{aligned} S_2 &= \begin{bmatrix} 0.8 & 0.2 \end{bmatrix} \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix} = \\ &= \begin{bmatrix} (0.8)(0.1) + (0.2)(0.6) & (0.8)(0.9) + (0.2)(0.4) \end{bmatrix} = \\ &= \begin{bmatrix} 0.2 & 0.8 \end{bmatrix} \end{aligned}$$

Умножение матрицы, соответствующей второму состоянию, на матрицу переходов позволяет получить следующие результаты:

$$\begin{aligned} S_3 &= S_2 T = \\ &= \begin{bmatrix} 0.2 & 0.8 \end{bmatrix} \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix} \\ S_3 &= \begin{bmatrix} 0.5 & 0.5 \end{bmatrix} \end{aligned}$$

А умножение полученной матрицы, соответствующей третьему состоянию, на матрицу переходов дает такие результаты:

$$\begin{aligned} S_4 &= S_3 T = \\ &= \begin{bmatrix} 0.57 & 0.5 \end{bmatrix} \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix} \\ S_4 &= \begin{bmatrix} 0.35 & 0.65 \end{bmatrix} \end{aligned}$$

Матрицы, соответствующие следующим состояниям, приведены ниже.

$$\begin{aligned} S_5 &= \begin{bmatrix} 0.425 & 0.575 \end{bmatrix} \\ S_6 &= \begin{bmatrix} 0.3875 & 0.6125 \end{bmatrix} \\ S_7 &= \begin{bmatrix} 0.40625 & 0.59375 \end{bmatrix} \\ S_8 &= \begin{bmatrix} 0.396875 & 0.602125 \end{bmatrix} \end{aligned}$$

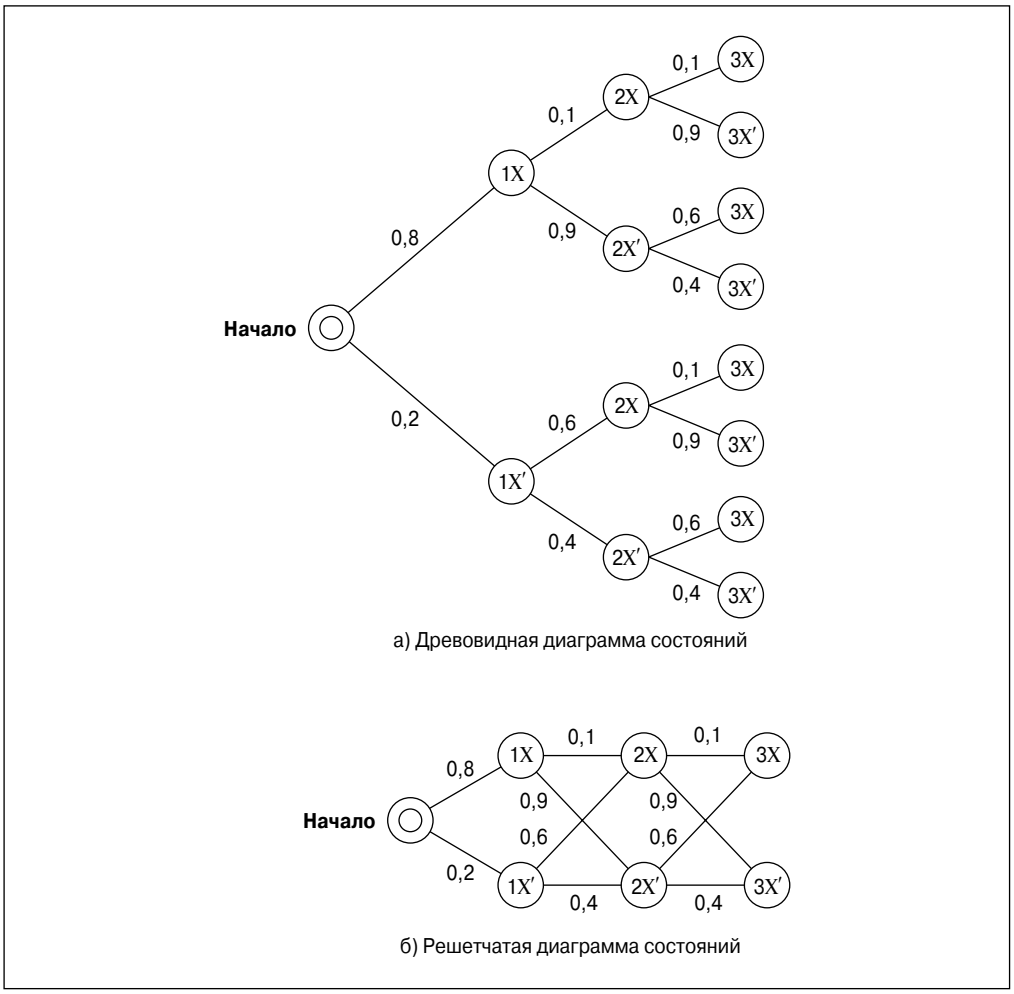


Рис. 4.14. Диаграммы переходов между состояниями во времени, имеющие вид дерева и решетки

Следует отметить, что эти матрицы состояний сходятся к следующей матрице, называемой **матрицей установившихся состояний**:

$$\begin{bmatrix} 0.4 & 0.6 \end{bmatrix}$$

Систему, находящуюся в установившемся состоянии, принято называть находящейся в равновесии, поскольку система не изменяется. Любопытно отметить, что значения в матрице установившихся состояний не зависят от начального состояния. Даже если бы использовался какой-то другой начальный вектор вероятностей, значения в установившемся состоянии были бы теми же самыми.

Вектор вероятностей S представляет собой матрицу установившихся состояний для матрицы переходов T, если справедливо следующее соотношение:

$$(1) \quad S = ST$$

Если T — **обычная матрица переходов** (таковой называется матрица, имеющая определенную степень и содержащая только положительные элементы), то существует единственное установившееся состояние S. Тот факт, что элементы матрицы переходов являются положительными, означает, что в какой-то момент времени возможен переход системы в определенное состояние, независимо от того, каковым является начальное состояние. Таким образом, потенциально достижимо любое состояние.

Процесс в марковской цепи определен как имеющий перечисленные ниже характеристики.

1. Процесс имеет конечное количество возможных состояний.
2. Процесс может одновременно находиться в одном и только одном состоянии.
3. Процесс со временем последовательно переходит, или **трансформируется** из одного состояния в другое.
4. Вероятность перехода в некоторое состояние зависит только от непосредственно предшествующего состояния.

Например, предположим, что дано конечное множество состояний $\{A, B, C, D, E, F, G, H, I\}$. В таком случае, если следующим состоянием, в которое переходит процесс из состояния H, является I, то условная вероятность определяется таким выражением:

$$P(I \mid H) = P(I \mid H \cap G \cap F \cap E \cap D \cap C \cap B \cap A)$$

Обратите внимание на то, насколько диаграмма в виде решетки, представленная на рис. 4.14, *б*, напоминает цепь.

Рассматриваемый случай с покупкой жестких дисков представляет собой процесс в марковской цепи, поэтому матрица установившихся состояний может быть определена путем применения уравнения (1). Примем за основу некоторый произвольный вектор S с компонентами X и Y и применим к нему уравнение (1) следующим образом:

$$\begin{bmatrix} X & Y \end{bmatrix} \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix} = \begin{bmatrix} X & Y \end{bmatrix}$$

Выполнение операции умножения в левой части и приравнивание полученных элементов соответствующим элементам в правой части приводит к получению

следующей зависимой системы уравнений:

$$0.1X + 0.6Y = X$$
$$0.9X + 0.4Y = Y$$

Решение этой системы уравнений для X в зависимости от Y приводит к получению таких результатов:

$$X = \frac{0.6}{0.9}Y = \frac{2}{3}Y$$

Чтобы полностью найти решение для X и Y, воспользуемся тем фактом, что сумма вероятностей равна 1. Таким образом,

$$X + Y = 1$$

Это означает, что:

$$X = 1 - Y = \frac{2}{3}Y$$

и поэтому:

$$X = \frac{2}{5}Y = \frac{3}{5}$$

Таким образом, матрица установившихся состояний принимает следующий вид, т.е. действительно совпадает с матрицей, полученной в конечном итоге в результате сходящихся вычислений с использованием значений, принятых в качестве гипотетических:

$$\begin{bmatrix} 0.4 & 0.6 \end{bmatrix}$$

4.11 Анализ вероятностных систем на основе понятий шансов и убеждений

До сих пор вероятности рассматривались как показатели, применяемые для анализа повторяющихся событий в идеальных системах. Но люди проявляют необычайные способности при вычислении вероятностей многих неповторяющихся событий, например, происходящих при медицинской диагностике и разведке полезных ископаемых, когда любой пациент и любое месторождение являются уникальными. Для того чтобы обеспечить возможность использования экспертных систем в подобных областях, необходимо расширить понятие события и распространить его на **высказывания** (таковыми являются утверждения, которые могут быть истинными или ложными). Например, в качестве события может рассматриваться следующее высказывание:

```
"Кожа пациента покрыта красными пятнами"
```

а также такое высказывание:

```
"У пациента - корь"
```

Предположим, что A — высказывание. В таком случае приведенное ниже выражение для условной вероятности может не обязательно соответствовать определению вероятности в классическом смысле, если события и высказывания не могут повторяться или иметь математическое обоснование.

$$P(A \mid B)$$

Вместо этого выражение $P(A \mid B)$ может интерпретироваться как **степень доверия** к тому, что A истинно, если дано B.

Если $P(A \mid B) = 1$, мы уверены в том, что выражение A действительно истинно, если же $P(A \mid B) = 0$, то мы уверены в том, что A действительно ложно, а все другие значения, $0 < P(A \mid B) < 1$, означают, что мы не совсем уверены в истинности или ложности A. Для обозначения некоторых высказываний, истинность или ложность которых невозможно установить с полной уверенностью на основании некоторого **свидетельства**, применяется термин **гипотеза**, взятый из статистики. В таком случае условная вероятность рассматривается как **правдоподобие**. Например, значение $P(H \mid E)$ выражает правдоподобие некоторой гипотезы, H, определяемое на основании некоторого свидетельства, E.

Безусловно, выражение $P(H \mid E)$ имеет такую же форму, как и выражение для условной вероятности, но фактически обозначает нечто иное — правдоподобие, или степень доверия. Понятие вероятности распространяется на повторяющиеся события, а понятие правдоподобия выражает степень доверия к тому, что произойдут неповторяющиеся события. Экспертные системы моделируют работу экспертов-людей, поэтому выражение $P(H \mid E)$, вообще говоря, представляет степень доверия эксперта, согласно которой некоторая гипотеза, H, является истинной, если дано некоторое свидетельство, E. Безусловно, если бы события были повторяемыми, то выражение $P(H \mid E)$ представляло бы просто вероятность.

Если мы согласимся с тем, что выражение $P(H \mid E)$ обозначает правдоподобие, или степень доверия, то что показывает некоторое значение этого выражения, такое как 50% или 95%? Например, предположим, вы на 95% уверены в том, что двигатель вашего автомобиля в следующий раз запустится. Один из способов интерпретации этого правдоподобия состоит в использовании понятия **шансов** выигрыша ставки. Шансы odds выигрыша A против B, если дано некоторое событие C, определяются следующим образом:

$$\text{odds} = \frac{P(A \mid C)}{P(B \mid C)}$$

Если $B = A'$, то выражение

$$\text{odds} = \frac{P(A \mid C)}{P(A' \mid C)} = \frac{P(A \mid C)}{1 - P(A \mid C)}$$

определяет следующее:

$$P = P(A \mid C)$$

А это приводит к получению:

$$\text{odds} = \frac{P}{1 - P} \quad \text{и} \quad P = \frac{\text{odds}}{1 + \text{odds}}$$

В терминах шансов выигрыша в азартной игре можно интерпретировать P как выигрыш wins, а $1 - P$ как проигрыш losses, поэтому имеет место следующее выражение:

$$\text{odds} = \frac{\text{wins}}{\text{losses}}$$

Если известны шансы, odds, могут быть вычислены вероятность, или правдоподобие, и наоборот.

Таким образом, правдоподобие, определяемое значением $P = 95\%$, эквивалентно такой степени уверенности в том, что двигатель автомобиля запустится:

$$\text{odds} = \frac{.95}{1 - .95} = 19 \text{ к } 1$$

Это означает, что вы должны быть **безразличны** к ставке с шансами 19:1 на то, что двигатель автомобиля запустится. Если кто-то вам предложит ставку в 1 доллар на то, что двигатель автомобиля не запустится, то вы готовы будете заплатить 19 долларов, если он действительно не запустится. Применение приведенной выше формулы позволяет при наличии степени доверия, выраженной как вероятность, интерпретировать ее в терминах ставки с эквивалентными шансами. Иными словами, вы всегда сможете узнать в реальной ситуации, какая степень доверия или ставка с эквивалентными шансами должна быть для вас безразлична.

Вероятности, как правило, используются при решении дедуктивных задач, в которых дана некоторая гипотеза и может произойти целый ряд различных событий, E_i. Например, предположим, что рассматриваются броски игральной кости, в которых выпадает четное количество очков (событие even); в таком случае количество возможных событий равно трем:

$$P(2 \mid \text{even})$$
$$P(4 \mid \text{even})$$
$$P(6 \mid \text{even})$$

В тех задачах, в которых используются вероятности, обычно представляет интерес выражение $P(E_i \mid H)$, где E_i — возможные события, соответствующие общей гипотезе. А в задачах статистики и в задачах с индуктивными рассуждениями обычно известно, что произошло некоторое событие и требуется найти правдоподобие гипотезы, в соответствии с которой могло произойти событие E, что определяется выражением $P(E \mid H_i)$. Понятие вероятности естественным образом подходит для осуществления прямого логического вывода, или дедуктивного вывода, а понятие правдоподобия — для обратного логического вывода, или индуктивного вывода. Безусловно, для вероятности и правдоподобия используется одно и то же обозначение, $P(X \mid Y)$, но области применения этих понятий различны. Обычно в задаче речь идет либо о правдоподобии гипотезы, либо о вероятности события.

Одной из теорий, которая была разработана на основе понятия степени доверия, является теория **индивидуальной вероятности**. В теории индивидуальной вероятности как возможные гипотезы рассматриваются **состояния**, а как результаты действий, выполненных на основе убеждений, —**последствия**.

4.12 Достаточность и необходимость

Теорема Байеса имеет следующую формулировку:

$$P(H \mid E) = \frac{P(E \mid H)P(H)}{P(E)} \tag{1}$$

а выражение для отрицания гипотезы H имеет вид

$$P(H' \mid E) = \frac{P(E \mid H')P(H')}{P(E)} \tag{2}$$

Разделив уравнение (1) на уравнение (2), получим следующее:

$$\frac{P(H \mid E)}{P(H' \mid E)} = \frac{P(E \mid H)P(H)}{P(E \mid H')P(H')} \tag{3}$$

Введем такое определение априорных шансов для гипотезы H:

$$O(H) = \frac{P(H)}{P(H')}$$

а определение апостериорных шансов определим так:

$$O(H \mid E) = \frac{P(H \mid E)}{P(H' \mid E)}$$

Наконец, получим следующее определение коэффициента правдоподобия:

$$LS = \frac{P(E \mid H)}{P(E \mid H')} \tag{4}$$

В таком случае уравнение (3) принимает следующий вид:

$$O(H \mid E) = LS\, O(H) \tag{5}$$

Уравнение (5) известно как **форма с шансами и правдоподобием** теоремы Байеса. Это уравнение представляет собой форму теоремы Байеса, более удобную для работы по сравнению с уравнением (1).

Коэффициент LS называется также **правдоподобием достаточности**. Это связано с тем, что если $LS = \infty$, то свидетельство E становится логически достаточным для принятия заключения об истинности гипотезы H. А если свидетельство E логически достаточно для принятия заключения об истинности гипотезы H, то $P(H \mid E) = 1$ и $P(H \mid E') = 0$. Уравнение (5) может использоваться для вычисления значения LS следующим образом:

$$LS = \frac{O(H \mid E)}{O(H)} = \frac{\dfrac{P(H \mid E)}{P(H' \mid E)}}{\dfrac{P(H)}{P(H')}} \tag{6}$$

Отметив, что $P(H)/P(H')$ представляет собой некоторую константу, C, преобразуем уравнение (6) следующим образом:

$$LS = \frac{\frac{1}{0}}{C} = \infty$$

В данном случае уравнение (4) также показывает, что гипотеза H является достаточной для свидетельства E. В табл. 4.10 приведены итоговые сведения о том, какой смысл имеют другие компоненты формулы вычисления коэффициента LS.

Правдоподобие необходимости, LN, определяется по такому же принципу, как и правдоподобие достаточности, LS:

$$LN = \frac{O(H \mid E')}{O(H)} = \frac{P(E' \mid H)}{P(E' \mid H')} = \frac{\dfrac{P(H \mid E')}{P(H' \mid E)}}{\dfrac{P(H)}{P(H')}} \tag{7}$$

$$O(H \mid E') = LNO(H) \tag{8}$$

Если $LN = 0$, то $P(H \mid E') = 0$. Это означает, что если значение E' является истинным, то значение H должно быть ложным. Таким образом, если событие E

Таблица 4.10. Отношения между коэффициентом правдоподобия, гипотезой и свидетельством

Коэффициент LS	**Влияние на гипотезу**
0	Гипотеза H ложна, если свидетельство E истинно, иными словами, для принятия гипотезы H необходимо, чтобы было истинным E'
Малые значения $(0 < LS << 1)$	Свидетельство E не способствует принятию гипотезы H
1	Свидетельство E не влияет на степень доверия к гипотезе H
Большие значения $(1 << LS)$	Свидетельство E способствует принятию гипотезы H
∞	Свидетельство E логически достаточно для принятия гипотезы H, или из наблюдения свидетельства E следует, что гипотеза H должна быть истинной

отсутствует, то гипотеза H ложна; иными словами, событие E является необходимым для принятия гипотезы H.

Отношения между LN, E и H показаны в табл. 4.11. Обратите внимание на то, что описания, приведенные в табл. 4.11, совпадают с описаниями в табл. 4.10, в которых термин “свидетельство” заменен термином “отсутствие свидетельства”.

Таблица 4.11. Отношения между правдоподобием необходимости, гипотезой и свидетельством

Коэффициент LN	**Влияние на гипотезу**
0	Гипотеза H ложна, если свидетельство E отсутствует, иными словами, свидетельство E необходимо для принятия гипотезыH
Малые значения $(0 < LS << 1)$	Отсутствие свидетельства E не способствует принятию гипотезы H
1	Отсутствие свидетельства E не влияет на степень доверия к гипотезе H
Большие значения $(1 << LS)$	Отсутствие свидетельства E способствует принятию гипотезы H
∞	Отсутствие свидетельства E логически достаточно для принятия гипотезы H

Значения правдоподобия LS и LN, необходимые для вычисления апостериорных шансов, должны быть предоставлены экспертом-человеком. Уравнения (5) и (8) имеют простую форму, доступную для понимания людей. Коэффициент

LS показывает, насколько изменяются априорные шансы, если обнаруживается, что свидетельство имеется, а коэффициент LN показывает, насколько изменяются априорные шансы, если обнаруживается, что свидетельство отсутствует. Эти формы позволяют затем эксперту-человеку проще задавать значения коэффициентов LS и LN.

В качестве примера укажем, что в экспертной системе PROSPECTOR имеется правило, определяющее, как с помощью свидетельства о наличии некоторого минерала обосновывается гипотеза:

```
IF обнаруживаются кварцево-сульфидные прожилки
THEN имеет место изменение пород по сложению и составу,
     перспективное с точки зрения обнаружения калиевых пород
```

Данная конкретная промежуточная гипотеза служит обоснованием для других гипотез, ведущих к формированию гипотезы верхнего уровня, говорящей о наличии месторождения меди. Значения коэффициентов LS и LN для этого правила являются следующими:

$$LS = 300$$
$$LN = 0.2$$

Это означает, что обнаружение кварцево-сульфидных прожилок является довольно обнадеживающим, а отсутствие наблюдений таких прожилок свидетельствует не в пользу гипотезы, но это влияние не так уж неблагоприятно. А если бы значение коэффициента LN было намного меньше 1, то отсутствие прожилок сульфида кварца строго свидетельствовало бы в пользу того, что гипотеза ложна. В качестве подобного примера можно привести следующее правило:

```
IF обнаруживаются стекловидные включения бурого железняка
THEN имеют место наилучшие перспективы минерализации
```

для которого приняты такие коэффициенты:

$$LS = 1000000$$
$$LN = 0.01$$

4.13 Применение неопределенности при формировании цепей логического вывода

Неопределенность может присутствовать в правилах, в свидетельствах, используемых в правилах, или в тех и в других. В данном разделе рассматриваются некоторые практические задачи, связанные с неопределенностью, и показано, как найти решение этих задач с помощью вероятностей.

Несовместимость коэффициентов, заданных экспертом

Согласно уравнению (4), приведенному в предыдущем разделе,

$$\text{if } LS > 1 \text{ then } P(E \mid H') < P(E \mid H)$$

Вычитая выражение в каждой части этого уравнения из 1, получим обратное неравенство:

$$1 - P(E \mid H') > 1 - P(E \mid H)$$

Поскольку имеют место соотношения $P(E' \mid H) = 1 - P(E \mid H)$ и $P(E' \mid H') = 1 - P(E \mid H')$, уравнение (7) можно преобразовать в такой вид:

$$LN = \frac{1 - P(E \mid H)}{1 - P(E \mid H')} < 1$$

На значения коэффициентов LS и LN распространяются определенные ограничения, общие сведения о которых приведены ниже.

Случай 1. $LS > 1$ и $LN < 1$

Согласно уравнениям (4) и (7), другими случаями являются:

Случай 2. $LS < 1$ и $LN > 1$

Случай 3. $LS = LN = 1$

Безусловно, в этих трех случаях рассматриваются математически строгие ограничения, которые распространяются на значения коэффициентов LS и LN, но в реальном мире такие ограничения не всегда оправданы. Практика применения экспертной системы PROSPECTOR для разведки полезных ископаемых показала, что эксперты нередко задают значения $LS > 1$ и $LN = 1$, которые не соответствуют трем рассматриваемым случаям. Иными словами, эксперт, задавая такие значения, указывает, что важно наличие свидетельства, но отсутствие свидетельства не является важным.

Этот последний случай показывает, что теория правдоподобия, основанная на байесовской теории вероятностей, является неполной с точки зрения задачи разведки полезных ископаемых. Это означает, что байесовская теория правдоподобия представляет собой всего лишь приближение для той теории, которая позволяла бы также справляться с тем случаем, когда $LS > 1$ и $LN = 1$. С другой стороны, для тех проблемных областей, в которых мнения экспертов всегда соответствуют одному из первых трех случаев, байесовская теория правдоподобия остается удовлетворительной.

Неопределенное свидетельство

В реальном мире почти ни в чем нельзя быть абсолютно уверенным, кроме наступления смерти и взыскания налогов, а после изобретения клонирования и крионики нельзя даже быть уверенным в неизбежном наступлении смерти. Безусловно, до сих пор речь шла в основном о неопределенных гипотезах, но более общей и реалистичной является такая ситуация, в которой неопределенными являются и гипотезы, и свидетельства.

В качестве общего случая предположим, что степень доверия к **полному свидетельству**, E, зависит от **частичного свидетельства**, e, которое определяется следующим выражением:

$$P(E \mid e)$$

Полное свидетельство — это суммарное свидетельство, представляющее все возможные свидетельства и гипотезы, из которых состоит E. Частичное свидетельство, e, представляет собой известную нам часть E. А если известно все свидетельство, то $E = e$ и справедливо следующее выражение, в котором $P(E)$ — априорное правдоподобие свидетельства E:

$$P(E \mid e) = P(E)$$

Правдоподобие $P(E \mid e)$ представляет собой степень доверия к свидетельству E, с учетом неполноты знаний, e, о полном свидетельстве E.

Например, предположим, что люди, живущие по соседству, которые пробурили нефтяные скважины в своих кухнях, разбогатели. Рассмотрим следующие гипотезу и свидетельство:

```
H = я разбогатею
E = под моей кухней находится месторождение нефти
```

выраженные в виде такого правила:

```
IF под моей кухней находится месторождение нефти
THEN я разбогатею
P(H | E) = 1
```

На этом начальном этапе вы еще не знаете с полной уверенностью, находится ли месторождение нефти и под вашей кухней. Окончательным свидетельством стало бы бурение контрольной скважины, но такой эксперимент обойдется достаточно дорого. Поэтому вы рассматриваете описанное ниже частичное свидетельство, e, которое могло бы служить обоснованием полного свидетельства E.

- Другие люди, живущие по соседству, смогли разбогатеть.
- Вокруг кухонной печи всегда просачивается какое-то черное вещество (которое ваша вторая половина всегда считала результатом вашей деятельности по приготовлению пищи и до последнего времени просто вытирала).

- К вашей двери подходил незнакомец и предлагал купить ваш дом за 20 миллионов долларов, объяснив такое невероятное предложение тем, что ему нравится, как этот дом красиво расположен.

На основании этого частичного свидетельства можно определить, что правдоподобие полного свидетельства о наличии месторождения нефти под вашей кухней является довольно высоким, т.е. $P(E \mid e) = 0.98$.

При формировании цепи логического вывода, основанной на использовании вероятности или правдоподобия, принято считать, что если гипотеза H зависит от полного свидетельства E, а E основано на некотором частичном свидетельстве, e, то $P(H \mid e)$ — правдоподобие того, что H зависит от e. Согласно правилу для условной вероятности:

$$P(H \mid e) = \frac{P(H \cap e)}{P(e)}$$

На рис. 4.15 показано, как выражается идея, состоящая в том, что гипотеза H обосновывается свидетельствами E и e. Таким образом, приведенное выше уравнение принимает вид

$$P(H \mid e) = \frac{P(H \cap E \cap e) + P(H \cap E' \cap e)}{P(e)}$$

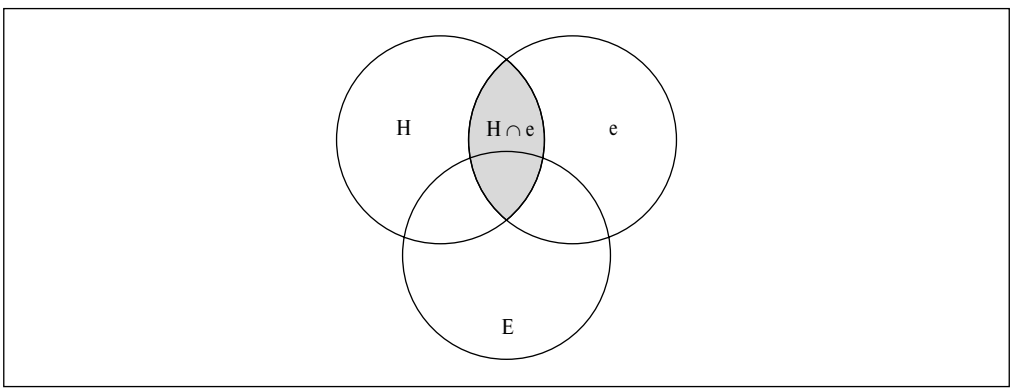


Рис. 4.15. Пересечение множеств, соответствующих гипотезе H и частичному свидетельству e

Используя правило условной вероятности, получим следующее:

$$P(H \mid e) = \frac{P(H \cap E \mid e)P(e) + P(H \cap E' \mid e)P(e)}{P(e)}$$

$$P(H \mid e) = P(H \cap E \mid e) + P(H \cap E' \mid e) \quad (1)$$

Из этого следует еще один способ представления уравнения (1), поскольку справедливо следующее уравнение:

$$P(H \cap E \mid e) = \frac{P(H \cap E \cap e)}{P(e)}$$

$$P(H \cap E \mid e) = \frac{P(H \mid E \cap e)P(E \cap e)}{P(e)} \tag{2}$$

а также, поскольку имеет место следующее:

$$P(E \mid e) = \frac{P(E \cap e)}{P(e)}$$

Таким образом, уравнение (2) принимает вид

$$P(H \cap E \mid e) = P(H \mid E \cap e)P(E \mid e)$$

Аналогичным образом, справедливо следующее уравнение:

$$P(H \cap E' \mid e) = P(H \mid E' \cap e)P(E' \mid e)$$

и уравнение (1) принимает вид

$$P(H \mid e) = P(H \mid E \cap e)P(E \mid e) + P(H \mid E' \cap e)P(E' \mid e) \tag{3}$$

Как правило, вероятности $P(H \mid E \cap e)$ и $P(H \mid E' \cap e)$ неизвестны. Но если принять предположение, что эти вероятности могут быть приближенно представлены с помощью выражений $P(H \mid E)$ и $P(H \mid E')$, то уравнение (3) упрощается до следующего уравнения:

$$P(H \mid e) = P(H \mid E)P(E \mid e) + P(H \mid E')P(E' \mid e) \tag{4}$$

Уравнение (4) по существу представляет собой линейную интерполяцию $P(H \mid e)$ по отношению к $P(E \mid e)$. Конечными точками этой интерполяции являются следующие:

(i) Свидетельство E истинно, поэтому $P(H \mid e) = P(H \mid E)$.

(ii) Свидетельство E ложно, поэтому $P(H \mid e) = P(H \mid E')$.

Но если значение $P(E \mid e)$ равно априорной вероятности $P(E)$, то при использовании уравнения (4) возникает проблема. С другой стороны, если система подчиняется чисто байесовской вероятности, то имеют место следующие уравнения, которые являются правильными:

$$P(H \mid e) = P(H \mid E)P(E) + P(H \mid E')P(E') \tag{5}$$

$$P(H \mid e) = P(H) \tag{6}$$

Но опыт решения реальных задач показал, что эксперты-люди присваивают такие значения субъективных вероятностей, которые почти наверняка оказываются несовместимыми. Например, если эксперт использует несовместимый случай с $LS > 1$ и $LN = 1$, то имеет место следующее:

$$O(H \mid E') = LNO(H) = O(H)$$

поэтому:

$$O = \frac{P}{1 - P}$$

В таком случае становится справедливой зависимость:

$$P(H \mid E') = P(H) \tag{7}$$

Подстановка уравнения (7) в уравнение (5) позволяет получить следующее уравнение:

$$\begin{aligned} P(H \mid e) &= P(H \mid E)P(E) + P(H)P(E') = \\ &= P(H \mid E)P(E) + P(H)(1 - P(E)) \end{aligned}$$

$$P(H \mid e) = P(H) + P(H \mid E)P(E) - P(H)P(E) \tag{8}$$

Теперь отметим, что:

$$O(H \mid E) = LSO(H)$$

Если $LS = 1$, то:

$$O(H \mid E) = O(H)$$
$$P(H \mid E) = P(H)$$

Эксперт-человек задал значение $LS > 1$. Таким образом, $P(H \mid E) > P(H)$, поэтому следующий терм в уравнении (8) будет иметь значение больше 0, где 0 — нижняя граница, если $LS = 1$:

$$P(H \mid E)P(E) - P(H)P(E)$$

Поэтому из уравнения (8) при $LS > 1$ и $LN = 1$ следует:

$$P(H \mid e) > P(H)$$

Это соотношение противоречит тому факту, что значение $P(H \mid e)$ должно равняться $P(H)$, если $P(E \mid e) = P(E)$. Начиная с соотношения $P(H \mid e) > P(H)$, вероятность становится больше, чем должна быть, и может продолжать увеличиваться, поскольку вывод из одного правила используется в другом правиле в цепи логического вывода.

Исправление недостатков, связанных с неопределенностью

Один из способ устранения описанных выше проблем состоит в принятии предположения, что $P(H \mid e)$ выражается кусочно-линейной функцией. Это — произвольное предположение, оправдавшее себя в системе PROSPECTOR, но не основанное на традиционной теории вероятностей. А линейная функция выбрана для упрощения вычислений.

Эта функция согласуется с ограничениями в трех обозначенных зачерненными кружками точках, которые показаны на рис. 4.16. Формула для $P(H \mid e)$ вычисляется с использованием линейной интерполяции следующим образом:

$$P(H \mid e) = \begin{cases} P(H \mid E') + \dfrac{P(H) - P(H \mid E')}{P(E)} P(E \mid e) \\ \quad \text{для } 0 \leqslant P(E \mid e) < P(E) \\ P(H) + \dfrac{P(H \mid E) - P(H)}{1 - P(E)} [P(E \mid e) - P(E)] \\ \quad \text{для } P(E) \leqslant P(E \mid e) \leqslant 1 \end{cases}$$

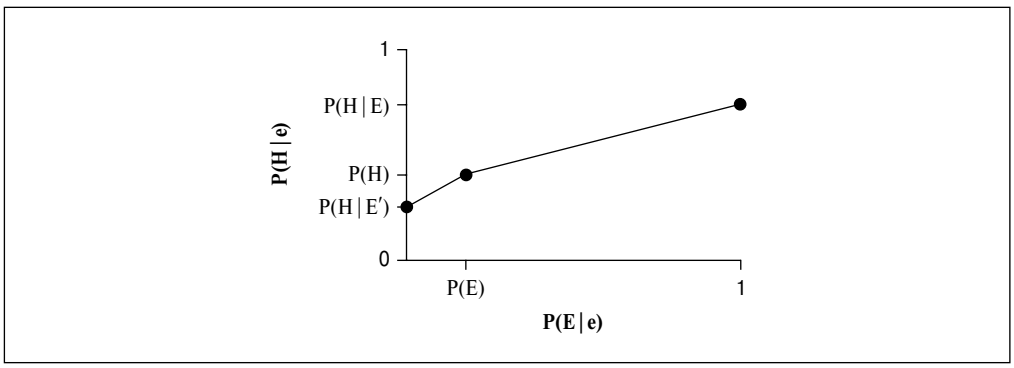


Рис. 4.16. Функция кусочно-линейной интерполяции для частичного свидетельства в системе PROSPECTOR

С помощью этой формулы можно выполнить требования, соответствующие описанному выше случаю несовместимости, в котором $LS > 1$ и $LN = 1$. Теперь значение $P(H \mid e)$ остается одинаковым, если $P(E \mid e) < P(E)$, и возрастает, если $P(E \mid e) = P(E)$. Причина, по которой была выбрана такая кусочно-линейная форма, а не просто одна прямая линия, состоит в том, что нужно обеспечить выполнение соотношения $P(H \mid e) = P(H)$ при $P(E \mid e) = P(E)$.

4.14 Комбинация свидетельств

В простейшем случае правило экспертной системы имеет приведенную ниже форму, в которой E — единственная часть известного свидетельства, на основание которого можно прийти к заключению, что гипотеза H истинна.

```
IF E THEN H
```

К сожалению, не все правила могут быть такими простыми, поскольку необходимо учитывать наличие неопределенности.

Варианты классификации неопределенных свидетельств

Если имеется какая-либо неопределенность в отношении правила, то возникают более сложные ситуации. *Неопределенность* может быть выражена как вероятность или правдоподобие, в зависимости от того, рассматриваются ли воспроизводимые события или субъективные вероятности. Но для упрощения мы будем использовать для обозначения неопределенности термин *вероятность*.

Различные ситуации можно классифицировать с учетом того, является ли свидетельство определенным или неопределенным, а также имеет ли место **простое свидетельство** или **сложное свидетельство**. Простое свидетельство состоит из единственной части свидетельства, такой как:

```
IF неисправна коробка передач
THEN сдайте автомобиль в ремонт
```

А сложное свидетельство состоит из нескольких частей свидетельства, обычно объединяемых с помощью связок AND, как в следующем примере:

```
IF неисправна коробка передач AND
   неисправен двигатель
THEN продайте автомобиль
```

или, в более формальном виде:

IF E_1 and E_2 then H

Этому правилу можно присвоить вероятность примерно так: $P(H \mid E_1 \cap E_2) = 0.80$. Это означает, что, согласно данному свидетельству, мы на 80% уверены в том, что автомобиль должен быть продан.

Дальнейшее уточнение свидетельства состоит в определении его вероятности. Например, вероятность того, что коробка передач неисправна, может зависеть от наличия следующих двух симптомов:

а) наблюдается утечка трансмиссионного масла;

б) автомобиль дергается при переключении скоростей.

Эти наблюдения помогают проще определить значение вероятности свидетельства, касающегося коробки передач. Например, на основании симптомов

а) и б) может быть принято следующее значение:

$$P(E \mid e) = 0.95$$

В этой формуле E — свидетельство, согласно которому неисправна коробка передач; e — описанные выше симптомы а) и б).

Теперь рассмотрим более подробно вероятности, связанные с различными ситуациями.

Ситуация 1. Заключение об истинности гипотезы H можно вывести из одной части известного свидетельства.

Это — простейший случай, в котором используются правила в следующей форме:

```
IF E THEN H
```

Априорная вероятность гипотезы H, определяемая до того, как становятся известными какие-либо свидетельства, равна $P(H)$. А после того как становится известным свидетельство, вероятность гипотезы H, согласно теореме Байеса, изменяется, как показано ниже.

$$P(H \mid E) = \frac{P(H \cap E)}{P(E)} = \frac{P(H \cap E)}{P(E \cap H) + P(E \cap H')}$$

$$P(H \mid E) = \frac{P(H \mid E)P(H)}{P(E \mid H)P(H) + P(E \mid H')P(H')}$$

Здесь $P(E)$ — вероятность наблюдения свидетельства E.

Ситуация 2. Заключение об истинности гипотезы H следует из двух частей известного свидетельства.

Эта ситуация соответствует более сложному случаю, чем ситуация 1, и отражается в правилах, имеющих такую форму:

```
IF E1 and E2 then H
```

После наблюдения свидетельств E_1 и E_2 вероятность гипотезы H изменяется и вместо априорной вероятности $P(H)$ принимает следующую форму:

$$P(H \mid E_1 \cap E_2) = \frac{P(H \cap E_1 \cap E_2)}{P(E_1 \cap E_2)} =$$

$$= \frac{P(H \cap E_1 \cap E_2)}{P(E_1 \cap E_2 \cap H) + P(E_1 \cap E_2 \cap H')}$$

$$P(H \mid E_1 \cap E_2) = \frac{P(E_1 \cap E_2 \mid H)P(H)}{P(E_1 \cap E_2 \mid H)P(H) + P(E_1 \cap E_2 \mid H')P(H')} \quad (1)$$

Дальнейшее сокращение этой формулы становится невозможным, если не будут приняты некоторые упрощающие допущения. В частности, если принято предположение, что свидетельства E_1 и E_2 условно независимы друг от друга, то имеет место следующее соотношение:

$$P(E_1 \cap E_2 \mid H) = P(E_1 \mid H)P(E_2 \mid H)$$
$$P(E_1 \cap E_2 \mid H') = P(E_1 \mid H')P(E_2 \mid H')$$

и поэтому справедливо такое уравнение (см. задачу 4.12, касающуюся определения условной независимости):

$$(2) \qquad P(H \mid E_1 \cap E_2) = \frac{P(E_1 \mid H)P(E_2 \mid H)}{P(E_1 \mid H)P(E_2 \mid H) + O(H')P(E_1 \mid H')P(E_2 \mid H')}$$

Уравнение (2) представляет собой результат значительного упрощения по сравнению с уравнением (1), поскольку теперь в нем все вероятности выражены в терминах индивидуальных вероятностей, а не совместных вероятностей, таких как $P(E_1 \cap E_2 \mid H')$. Но, как будет описано ниже, при использовании этого предположения об условной независимости возникают некоторые проблемы.

Очевидно, что уравнение (2) является значительно более упрощенным, но для его использования все равно требуется знать априорные вероятности $P(E_1)$ и $P(E_2)$. Обычно для экспертов задача определения априорных вероятностей является сложной, поскольку эксперты не проводят свои рассуждения по такому принципу. Например, если к врачу приходит пациент, то врач не предполагает априорно, что у этого пациента простуда, лишь на основании того, что простуда — наиболее распространенное заболевание, поэтому ее вероятность высока.

Основная проблема при присваивании значений априорным вероятностям заключается в том, что при использовании подхода, основанного на учете правдоподобия, значения таких вероятностей сложно определить. Если речь идет о вероятностях, касающихся таких воспроизводимых событий, как бросок игральной кости, то задача определения указанных вероятностей на основании эмпирических или теоретических исследований является несложной. С другой стороны, если дело касается таких событий, как поиск крупного месторождения минерального сырья, то действительно невозможно точно определить априорные значения правдоподобия. Например, каково априорное правдоподобие такой гипотезы, что ваш дом стоит на участке, под которым находится крупное месторождение золота? Измеряется ли оно значением 1,0, 0,01, 0,00000001 или каким-то другим числом?

Это — уникальные ситуации, в которых невозможно определить априорные вероятности.

Ситуация 3. Заключение об истинности гипотезы H следует из N частей неопределенного свидетельства.

Это — общий случай наличия неопределенности в свидетельстве и в правиле, зависящем от свидетельства. С какого-то момента возрастание количества частей свидетельства приводит к тому, что определение всех значений совместных и априорных вероятностей или правдоподобий становится невозможным. Поэтому для учета общего случая с N частями свидетельства используются различные приближенные методы.

Комбинирование свидетельств с использованием нечеткой логики

Конъюнкция свидетельств

Предположим, что следующее правило:

```
IF E THEN H
```

включает антецедент E, представляющий собой конъюнкцию свидетельств, как показано ниже.

```
IF E1 AND E2 AND ... EN THEN H
```

Для того чтобы антецедент принял истинное значение, необходимо, чтобы с определенной вероятностью все термы E_i были истинными. В общем случае каждая часть свидетельства основана на частичном свидетельстве e. Вероятность этого свидетельства определяется следующей формулой:

$$P(E \mid e) = P(E_1 \cap E_2 \cap \cdots \cap E_N \mid e) = \\ = \frac{P(E_1 \cap E_2 \cap \cdots \cap E_N \cap e)}{P(e)}$$

Если все свидетельства E_i являются условно независимыми, то их совместная вероятность вычисляется как произведение отдельных вероятностей. Это следует из обобщенного мультипликативного закона. Например, для двух частей свидетельства справедливо следующее соотношение:

$$P(E_1 \cap E_2 \mid e) = \frac{P(E_1 \cap E_2 \cap e)}{P(e)} = \\ = \frac{P(E_2 \mid E_1 \cap e)P(E_1 \mid e)P(e)}{P(e)}$$

На основании предположения о независимости можно вывести следующую формулу, поскольку свидетельство E_1 не способствует получению каких-либо

знаний об истинности свидетельства E_2.

$$P(E_2 \mid e) = P(E_2 \mid E_1 \cap e)$$

Таким образом, справедливо следующее соотношение:

$$P(E_2 \cap E_1 \mid e) = P(E_2 \mid e)P(E_1 \mid e)$$

а также, вообще говоря, следующее:

$$P(E_1 \cap E_2 \cap \cdots \cap E_N \mid e) = \prod_{i=1}^{N} P(E_i \mid e)$$

Безусловно, эта формула является теоретически обоснованной, но при ее применении для решения практических задач возникают сложности. Во-первых, обычно в реальном мире отдельные вероятности $P(E_i \mid e)$ не являются независимыми. Во-вторых, в результате умножения многочисленных факторов, полученных на основании первого предположения, формируется произведение, которое обычно слишком мало для $P(E \mid e)$.

Приближенное решение этой задачи состоит в использовании для вычисления $P(E \mid e)$ **нечеткой логики**, например, в виде следующей формулы, в которой **функция min** возвращает минимальное значение среди всех значений $P(E_i \mid e)$:

$$P(E \mid e) = \min[P(E_i \mid e)]$$

В системе PROSPECTOR эта формула вполне себя оправдала. После определения $P(E \mid e)$ это значение может использоваться в формуле кусочно-линейной функции для $P(H \mid e)$.

Основным недостатком этой формулы, полученной с помощью нечеткой логики, является то, что при ее использовании значение $P(E \mid e)$ становится нечувствительным ко всем значениям $P(E_i \mid e)$, кроме минимального. Это означает, что даже если все остальные вероятности будут возрастать, а минимальное значение вероятности останется одинаковым, подобные изменения не отразятся на значении $P(E \mid e)$. Таким образом, минимальное значение $P(E_i \mid e)$ блокирует распространение информации о других изменениях вероятностей по цепи логического вывода. А преимуществом формулы, полученной с помощью нечеткой логики, является то, что она позволяет проводить несложные вычисления.

Дизъюнкция свидетельств

Если правило представляет собой дизъюнкцию свидетельств, т.е. имеет такой вид:

IF E_1 OR E_2 OR ... E_N THEN H

то можно показать (см. задачу 4.13), что в условиях использования предположения о независимости свидетельств может быть получена следующая формула:

$$P(E \mid e) = 1 - \prod_{i=1}^{N}[1 - P(E_i \mid e)]$$

Недостаток этой формулы состоит в том, что рассчитанные с ее помощью значения вероятности слишком велики. Вместо этой формулы в системе PROSPECTOR применяется следующая формула, полученная с помощью нечеткой логики, в которой **функция max** возвращает максимальное значение среди всех значений $P(E_i \mid e)$:

$$P(E \mid e) = \max[P(E_i \mid e)]$$

Логическая комбинация свидетельств

Если антецедент представляет собой логическую комбинацию свидетельств, то для комбинирования свидетельств можно использовать нечеткую логику и правила отрицания. Например, допустим, что дано следующее правило:

```
IF E1 AND (E2 OR E3') THEN H
```

В таком случае можно применить следующие формулы:

$$E = E_1 \text{ AND } (E_2 \text{ OR } E_3')$$
$$E = \min\{P(E_1 \mid e), \max[P(E_2 \mid e), 1 - P(E_3 \mid e)]\}$$

Безусловно, подобные формулы, полученные на основе нечеткой логики, успешно использовались во многих системах, но могут быть определены и другие функции, обеспечивающие комбинирование свидетельств. Например, ниже приведена формула, альтернативная по отношению к той формуле, в которой для вычисления вероятности дизъюнкции применялась функция $\max$.

$$P(E_1 \cap E_2 \mid H) = \min[1, P(E_1 \mid H) + P(E_2 \mid H)]$$

Эффективные значения правдоподобия

В общем случае конкретная гипотеза может стать следствием многочисленных правил с неопределенными свидетельствами и несовместимыми априорными вероятностями. Если принято предположение об условной независимости свидетельств и все значения E_i, вносящие свой вклад в подтверждение гипотезы H, являются истинными, то справедлива такая формула:

$$O(H \mid E_1 \cap E_2 \cap \ldots E_N) = \left[\prod_{i=1}^{N} LS_i\right] O(H)$$

В этой формуле значения LS_i определены следующим образом:

$$LS_i = \frac{P(E_i \mid H)}{P(E_i \mid H')}$$

А если все свидетельства, подтверждающие гипотезу H, являются ложными, то справедлива аналогичная формула:

$$O(H \mid E'_1 \cap E'_2 \cap \cdots \cap E'_N) = \left[\prod_{i=1}^{N} LN_i\right] O(H)$$

в которой значения LN_i определяются таким образом:

$$LN_i = \frac{P(E'_i \mid H)}{P(E'_i \mid H')}$$

В общем случае, когда имеют место несовместимые априорные вероятности и неопределенные свидетельства, **эффективный коэффициент правдоподобия** LE определяется следующим образом:

$$LE_i = \frac{O(H \mid e_i)}{O(H)}$$

В этой формуле e_i представляет собой i-е частичное свидетельство, вносящее свой вклад в подтверждение гипотезы H. Обновление значения H на основании неопределенного и несовместимого свидетельства осуществляется по аналогии с предыдущим случаем, как показано ниже.

$$O(H \mid e_1 \cap e_2 \cap \cdots \cap e_N) = \left[\prod_{i=1}^{N} LE_i\right] O(H)$$

Эта формула может использоваться в экспертной системе, основанной на применении неопределенных свидетельств и несовместимых априорных вероятностей, как описано ниже.

а) Сохранять априорные шансы для каждого правила и значения LE после каждого вычисления вклада свидетельства в подтверждение гипотезы правила.

б) После каждого обновления значений $P(E_i \mid e_i)$ вычислять новые значения LE_i и апостериорные шансы.

Сложности, связанные с использованием условной независимости

Разумеется, предположение о независимости условных вероятностей позволяет упростить теорему Байеса, но при использовании этого предположения возникает целый ряд проблем. Предположение о независимости условных вероятностей в основном может оказать помощь на начальных этапах создания экспертной системы, когда гораздо важнее правильно представить общее поведение системы, чем получить правильные числовые результаты. На первых порах может оказаться, что инженер по знаниям более заинтересован в разработке адекватных цепей логического вывода, чем в получении точных числовых результатов. Это означает, что, допустим, в некоторой системе промежуточная гипотеза 10 должна активизироваться свидетельствами 23 и 34. Затем гипотеза 10 и свидетельство 8 должны активизировать гипотезы 52 и 96. Гипотеза 15 не должна активизироваться свидетельствами 23 и 24, и т.д. Как показывает уравнение (2), приведенное в этом разделе, если принято предположение о независимости условных вероятностей, то для вычисления значения $P(H \mid E_1 \cap E_2)$ требуется только пять значений вероятности. Эти значения приведены ниже.

$$P(E_1 \mid H), P(E_2 \mid H), P(E_1 \mid H'), P(E_2 \mid H'), P(H)$$

Еще одно значение вероятности, $P(H')$, необходимое для вычисления $O(H')$, не является независимым от других, поскольку справедливо следующее соотношение:

$$P(H') = 1 - P(H)$$

В задаче 4.12, *б* показана еще одна формула для вычисления $P(H \mid E_1 \cap E_2)$. В этой формуле предусматривается использование четырех других значений вероятностей:

$$P(H \mid E_1), P(H \mid E_2), P(H' \mid E_1), P(H' \mid E_2)$$

Таким образом, для вычисления значения $P(H \mid E_1 \cap E_2)$ может использоваться общее количество вероятностей, равное девяти, но фактически требуется задать только пять значений, поскольку остальные четыре можно вычислить при наличии этих пяти значений.

По мере того как экспертная система становится все более зрелой, это ограничение на количество независимых вероятностей превращается в реальную проблему. После того как цепи логического вывода начинают действовать правильно, инженер по знаниям должен заняться обеспечением того, чтобы экспертная система выводила правильные числовые значения. Но в связи с тем, что принято предположение о независимости условных вероятностей, инженер по знаниям не обладает полной свободой в настройке всех девяти вероятностей, пользуясь ко-

торой он мог бы легко настраивать значения этих вероятностей для получения желаемых результатов.

Применение предположения о независимости условных вероятностей позволяет откорректировать значение совместной вероятности $P(E_1 \cap E_2)$, поскольку справедлива следующая формула:

$$P(E_1 \cap E_2) = P(E_1)P(E_2)\left[\frac{P(H \mid E_1)P(H \mid E_2)}{P(H)} + \frac{P(H' \mid E_1)P(H' \mid E_2)}{P(H')}\right]$$

Это означает, что теперь на значение $P(E_1 \cap E_2)$ налагают ограничения априорные вероятности $P(E_1)$, $P(E_2)$ и $P(H)$. Такое положение дел противоречит структуре знаний эксперта-человека, который может знать только отдельно взятые значения $P(E_1)$, $P(E_2)$ и $P(E_1 \cap E_2)$. Таким образом, принятие предположения о независимости условных вероятностей препятствует применению всех знаний эксперта-человека.

Безусловно, решение руководствоваться предположением о независимости условных вероятностей на первый взгляд кажется оправданным, но фактически то, будет ли благодаря этому достигнут успех, зависит от реальной ситуации, моделируемой с помощью экспертной системы, поэтому надежды оправдываются не во всех случаях. В одной теории утверждается, что предположение о независимости условных вероятностей обычно оказывается ложным. А в другой теории говорится о том, что из предположения о независимости условных вероятностей следует наличие строгой независимости, а это служит доказательством того, что данное свидетельство неприменимо для обновления гипотез! Но такие утверждения так и остались утверждениями, которые опровергаются другой теорией. (Вывод из всей этой истории состоит в том, что утверждения, основанные на нечеткой логике, не вполне безупречны.)

Еще в одном методе применения субъективных байесовских вероятностей демонстрируется, как можно ослабить предположение о независимости условных вероятностей для того, чтобы из такого предположения больше не следовал вывод о строгой независимости [77].

4.15 Сети логического вывода

Вплоть до этого момента рассматривались лишь такие примеры формирования цепей прямого и обратного логического вывода, которые были очень малы и состояли всего лишь из нескольких правил. Но в практических задачах количество этапов логического вывода, требуемых для обоснования гипотезы или для достижения заключения, намного больше. Кроме того, основная часть или даже все эти этапы логического вывода осуществляются с учетом неопределенности свидетельств и самих правил. В подобных практических системах успешно использовались вероятностные рассуждения и теорема Байеса.

Подходящей архитектурой для экспертных систем, основанных на таксономии знаний, является сеть логического вывода. **Таксономия** представляет собой один из способов классификации и широко используется в таких естественных науках, как геология и биология. Простой пример таксономии уже рассматривался в этой книге в виде рисунка, приведенного в главе 3, на котором показана классификация различных видов кустарников малины (рис. 3.4).

Применение таксономии может принести пользу по двум причинам. Во-первых, этот метод позволяет организовать знания, относящиеся к некоторой теме, по принципу классификации объектов и описания отношений этих объектов с другими объектами. Таксономии позволяют выявлять такие важные особенности, как наследование свойств.

Во-вторых, причина значительной пользы таксономий состоит в том, что они способны направлять поиск доказательства гипотезы, такой как “В этом районе есть залежи медной руды”. Подобная помощь при доказательстве или опровержении гипотез очень важна в таких областях, как разведка полезных ископаемых, поскольку прямые свидетельства о наличии ценной руды редко лежат на поверхности. А прежде чем принять решение о вложении времени и денег в бурение разведывательной скважины, геолог пытается собрать свидетельства, обосновывающие эту гипотезу.

Система PROSPECTOR

Классической экспертной системой, в которой используются вероятностные рассуждения, является PROSPECTOR. Эта система была разработана для оказания помощи геологоразведывательным компаниям при определении того, является ли некоторый район перспективным с точки зрения наличия в нем месторождений полезных ископаемых определенных типов. Основная идея системы PROSPECTOR состоит в том, что в базе знаний экспертной системы PROSPECTOR закодированы знания опытных геологов, знакомых с принципами экономики, о различных **моделях** залегания полезных ископаемых. Геологическая модель — это группа свидетельств и гипотез, на основании которых можно судить о наличии в некотором районе минерального сырья определенного типа. Система PROSPECTOR не только помогает выявлять наличие полезных ископаемых, но и обладает способностью давать рекомендации по выбору наилучшего места для проведения разведывательного бурения в данном районе. По мере создания все большего и большего количества моделей возможности системы PROSPECTOR продолжают расширяться.

Данные, применяемые в каждой модели, организованы в виде **сети логического вывода**. На рис. 4.17 приведены итоговые данные о типах сетей, которые рассматривались до сих пор в настоящей книге, а на рис. 4.17, *г* показана очень простая сеть логического вывода. Узлы сети логического вывода могут представлять свидетельства, предназначенные для обоснования гипотез, например, касающихся наличия полезных ископаемых определенного типа (сами гипоте-

зы представлены другими узлами сети). В табл. 4.12 приведены общие сведения о некоторых из 22 моделей месторождений полезных ископаемых, представленных в системе PROSPECTOR.

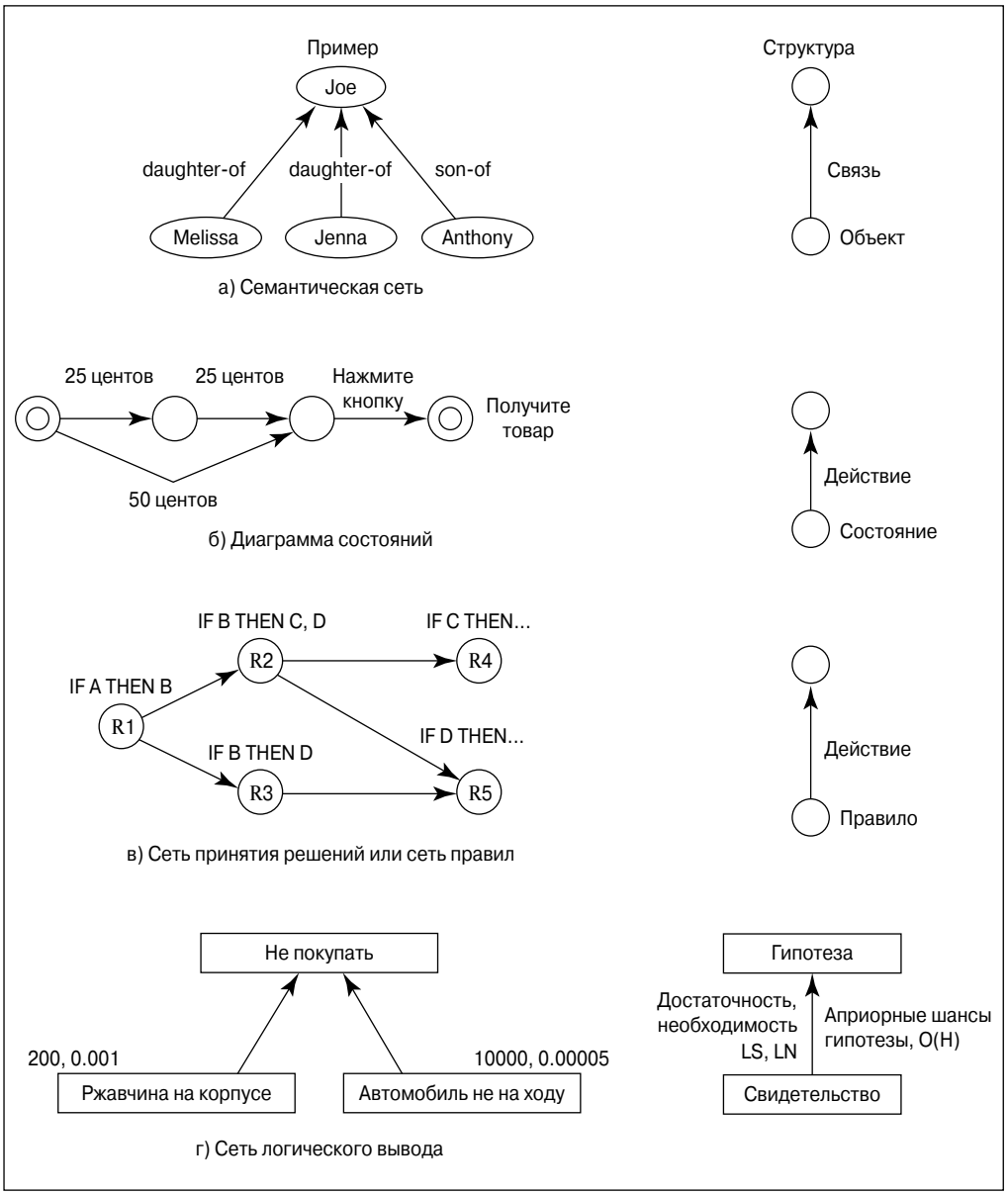


Рис. 4.17. Некоторые типы сетей

Таблица 4.12. Некоторые модели из системы PROSPECTOR

Наименование модели	Описание	Количество узлов	Количество запрашивающих узлов	Количество правил
PCD	Медно-порфировая руда (porphyry copper)	200	97	135
RFU	Месторождение урана в виде закругленного рудного тела (roll-front uranium)	185	147	169
SPB	Месторождение свинца в свите песчаников (sandstone-hosted lead)	35	21	32

В столбце табл. 4.12 “Количество узлов” показано общее количество узлов модели. **Запрашивающими узлами** называются узлы, с помощью которых формулируются вопросы к пользователю для получения наблюдаемых свидетельств. В столбце таблицы “Количество правил” показано количество вероятностных правил вывода. Вполне очевидно, что в показанных здесь моделях много неопределенности, поэтому при обосновании гипотез важно учитывать вероятность.

Сети логического вывода

Каждая модель, предназначенная для использования в системе PROSPECTOR, представлена в виде сети со связями (или отношениями), соединяющими свидетельства и гипотезы. Таким образом, сеть логического вывода является разновидностью семантической сети. Наблюдаемые факты, например, касающиеся типа горных пород, полученные в ходе геологического поиска, составляют свидетельство, применяемое для обоснования промежуточных гипотез. Затем группы промежуточных гипотез используются для обоснования **гипотезы верхнего уровня**. Таковой является гипотеза, которую требуется доказать. Если различие между свидетельством и гипотезой не важно, то для обозначения того и другого применяется термин **утверждение**. На рис. 4.18 показана небольшая часть сети логического вывода для гипотезы верхнего уровня из модели разведки медно-порфировой руды.

В системе PROSPECTOR используются показанные на рис. 4.18 **коэффициенты достоверности** CHE и CHNE, поскольку практика показала, что эксперты сталкиваются с трудностями, когда от них требуют задать значения апостериорных вероятностей или коэффициентов правдоподобия. Аналогичные практические результаты были обнаружены во время разработки системы MYCIN, предназначенной для диагностирования заболеваний крови. В ходе этой разработки

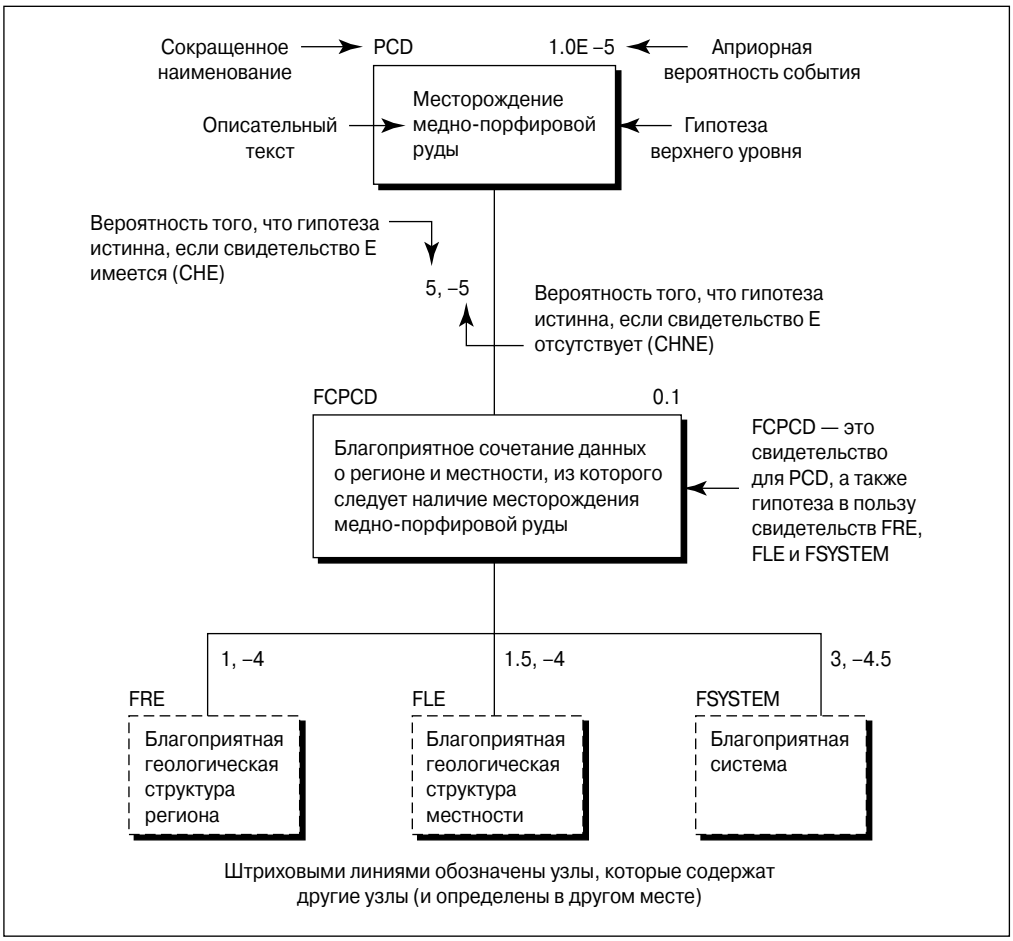


Рис. 4.18. Гипотеза верхнего уровня из модели разведки медно-порфировой руды системы PROSPECTOR, выраженная с использованием коэффициентов достоверности

врачи не испытывали большого желания задавать значения вероятностей, поэтому использовались коэффициенты достоверности. Что касается системы MYCIN, то коэффициенты достоверности ранжировались по 11-точечной шкале со значениями от -5 до $+5$, где -5 означало “определенно нет”, а $+5$ — “определенно да”.

PROSPECTOR не относится к категории чисто вероятностных систем, поскольку в ней для комбинирования свидетельств используются нечеткая логика и коэффициенты достоверности. Более подробное описание коэффициентов достоверности и нечеткой логики приведено в главе 5.

На рис. 4.19 более подробно показан узел FRE, приведенный на рис. 4.18, в развернутом виде.

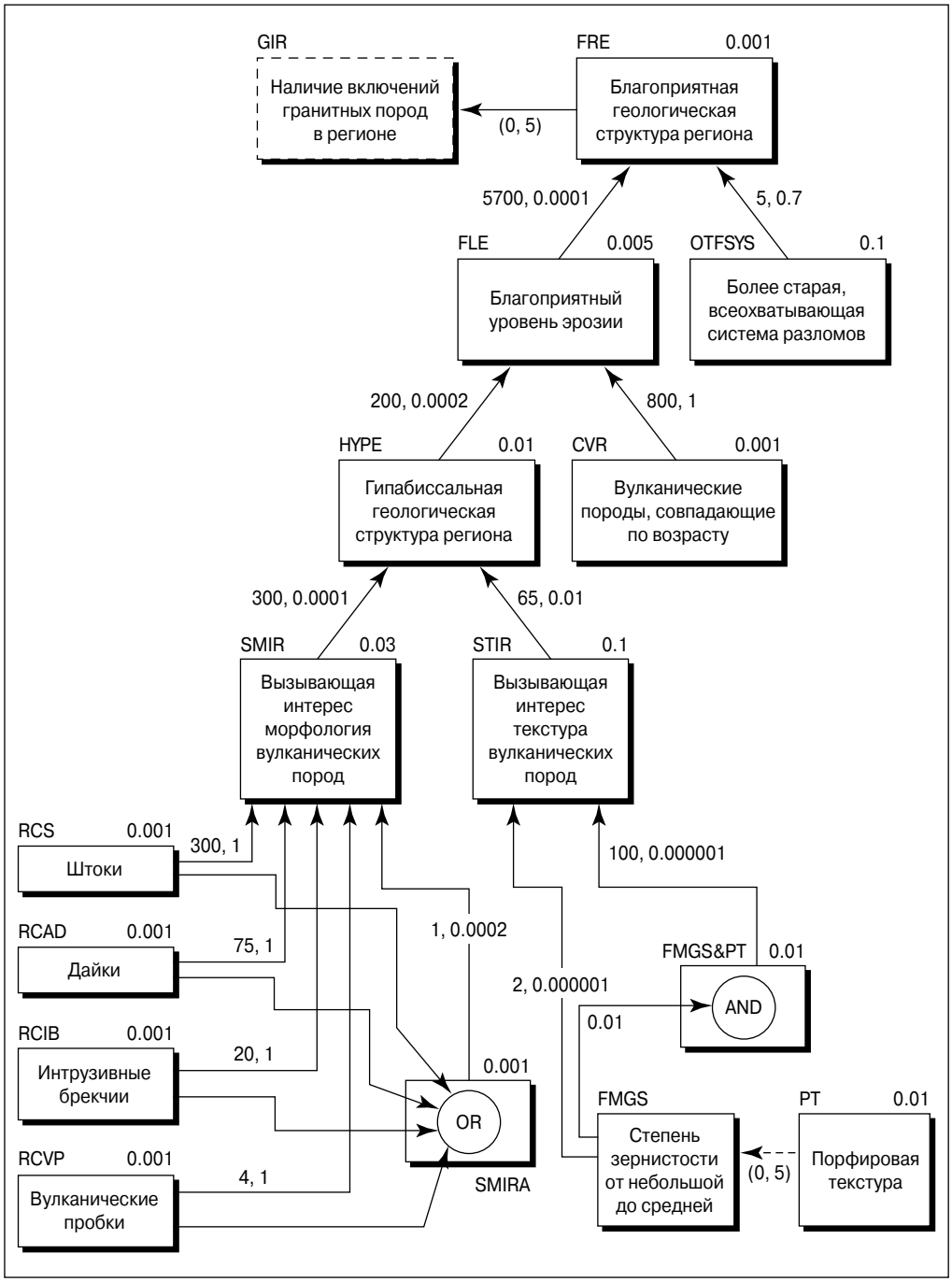


Рис. 4.19. Небольшая часть сети логического вывода системы PROSPECTOR, относящаяся к модели разведки медно-порфировой руды

На данной диаграмме над каждым узлом показаны два числа, которые представляют собой коэффициент правдоподобия, LS, и коэффициент необходимости, LN, разделенные запятой. Например, значения LS, LN для узла, находящегося внизу слева и обозначенного как RCIB, составляют 20.1. Имя каждого узла представляет собой аббревиатуру его описания, например, RCIB — сокращение от "the region contains intrusive breccias" ("в данном районе имеются интрузивные брекчии"). В верхней части каждого узла показано также отдельное число, представляющее собой априорную вероятность; например, для узла RCIB показано значение 0.001.

Отношения логического вывода

Свидетельства, доказывающие или опровергающие гипотезу, направлены вверх по сети логического вывода. Например, все свидетельства RCS, RCAD, RCIB, RCVP и SMIRA доказывают или опровергают промежуточную гипотезу SMIR. Промежуточная гипотеза SMIR, в свою очередь, является свидетельством для своей гипотезы HYPE, которая служит свидетельством для FLE, и т.д. Свидетельства могут комбинироваться двумя описанными ниже основными способами для получения отношений, необходимых для геолога, создающего определение модели.

- **Логические комбинации**, такие как комбинации, создаваемые с помощью узлов AND и OR. Как было указано в предыдущем разделе, для вычисления результата в этих логических узлах может применяться нечеткая логика.
- **Взвешенные комбинации**, в которых используются коэффициенты правдоподобия LS и коэффициенты необходимости LN. Апостериорные шансы вычисляются с помощью следующей формулы, если известно, что свидетельство E истинно:

  $$O(H \mid E) = LSO(H)$$

  и с помощью следующей формулы, если известно, что свидетельство E ложно:

  $$O(H \mid E') = LNO(H)$$

  Если же точное значение E неизвестно, то, как описано в предыдущем разделе, с помощью линейной интерполяции может быть вычислено значение $P(H \mid E)$.

Термин **взвешенная комбинация** сложился с учетом общего случая, в котором свой вклад в обоснование гипотезы вносят многочисленные части свидетельства. Как показано в предыдущем разделе, имеет место следующее соотношение:

$$O(H \mid E_1 \cap E_2 \cap \ldots E_N) = \left[\prod_{i=1}^{N} LS_i\right] O(H)$$

Логарифмирование указанного соотношения приводит к получению такой формулы:

$$\log O(H \mid E_1 \cap E_2 \cap \cdots \cap E_N) = \log O(H) + \left[\prod_{i=1}^{N} \log LS_i\right]$$

Эту формулу можно интерпретировать так, что каждое значение LS_i, преобразующееся в значение $\log LS_i$, "голосует" в пользу гипотезы. Каждое значение $\log LS_i$ представляет собой весовой коэффициент, от которого зависит гипотеза.

Безусловно, систему PROSPECTOR иногда называют системой, основанной на правилах, поскольку взвешенные комбинации соответствуют правилам, подобным приведенному ниже, но эта система не является столь же гибкой, как настоящая продукционная система, основанная на правилах.

```
IF E1 AND E2 AND ... EN THEN H
```

Одним из ее ограничений является отсутствие полноценного механизма связывания переменных. PROSPECTOR — это специализированная система, при разработке которой основной упор был сделан на обеспечение эффективности и контроля над приложениями геологического назначения, а не на обеспечение общности продукционной системы.

Рассматриваемые взвешенные комбинации могут также служить примером **правдоподобных отношений**. Термин *правдоподобный* означает, что есть некоторое свидетельство в пользу сложившегося доверия. PROSPECTOR — пример системы, в которой используется правдоподобный логический вывод для доказательства или опровержения гипотезы. Этапы правдоподобного логического вывода в системе PROSPECTOR основаны на использовании байесовских вероятностей, в которых значения LS и LN задают эксперты-люди.

На рис. 4.20 показан нечеткий граф, в котором рассматриваются правдоподобные и другие отношения, основанные на использовании различных степеней доверия. Применяемые здесь термины имеют такой общий смысл, как показано в табл. 4.13.

Таблица 4.13. Некоторые термины, используемые при описании свидетельств

Термин	Свидетельство, относящееся к гипотезе
Невозможное	Определенно известное как противоречащее гипотезе
Возможное	Не может быть опровергнуто со всей определенностью
Правдоподобное	Существует некоторое свидетельство
Вероятное	Некоторое свидетельство обосновывает гипотезу
Достоверное	Определенно известное как обосновывающее гипотезу

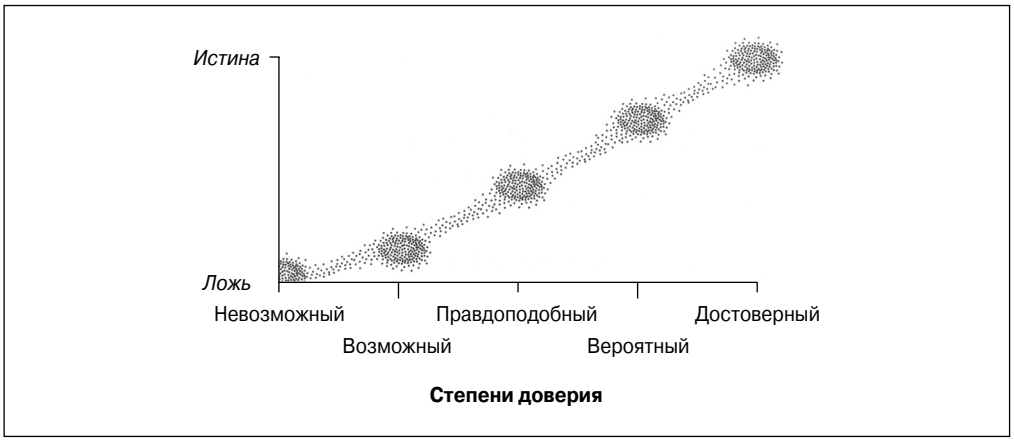


Рис. 4.20. Относительный смысл некоторых терминов, используемых для описания свидетельств

График, приведенный на рис. 4.20, намеренно изображен как нечеткий, чтобы проще было показать неопределенный характер этих терминов и подчеркнуть некоторую степень неосведомленности, наблюдающейся при переходе от одного понятия к другому. Обратите внимание на то, как на рис. 4.20 возрастает степень доверия к гипотезе от степени "невозможно" до степени "несомненно". В данном случае **несомненная степень доверия** (certain belief) означает, что доверие полностью оправдано, а **невозможная степень доверия** (impossible belief) означает, что оно полностью не оправдано. С несомненными и невозможными степенями доверия не связано никакой неопределенности. Эти степени доверия эквивалентны логически истинным и логически ложным значениям. С другой стороны, термин **определенное свидетельство** (certain evidence) иногда используется неоднозначно. Определенное свидетельство является либо логически истинным, либо логически ложным. Это означает, что с определенным свидетельством не ассоциируется какая-либо неопределенность. Таким образом, определенное свидетельство соответствует либо несомненной степени доверия (логически истинному значению), либо невозможной степени доверия (логически ложному значению).

Возможная степень доверия (possible belief) означает, что гипотеза, независимо от того, насколько далеко она от истины, не может быть исключена. Например, до того как стал возможным научный анализ состава поверхности Луны, нельзя было исключить возможность, хотя и очень, очень и очень отдаленную, что Луна состоит из белого сыра. Такая возможность существовала потому, что не было доступно определенное доказательство.

Термин **правдоподобная степень доверия** (plausible belief) означает, что существует несколько возможностей. Термин *правдоподобный* часто используется в судебном разбирательстве как синоним термина *обоснованный*, но лишь при отсутствии надежных свидетельств, которые оправдывали бы доверие к рассмат-

риваемой гипотезе. Это означает, что даже до получения первых результатов научных свидетельств такая гипотеза, что Луна сделана из зеленого, а не белого сыра, не была бы правдоподобной.

Термин **вероятная степень доверия** (probable belief) означает, что имеются некоторые свидетельства в пользу гипотезы, но этих свидетельств недостаточно, чтобы доказать гипотезу со всей определенностью. Например, если вы все время побеждали, бросая игральные кости, и вдруг начали терпеть поражение, после того как ваш друг предложил вам испытать его "удачливые игральные кости", вы можете приобрести вероятную степень доверия к гипотезе, что теперь удача перешла к вашему другу.

Без свидетельств, полученных в результате геологического поиска, таких как наблюдение RCIB ("район содержит интрузивные брекчии"), отношения, показанные на рис. 4.19 и относящиеся к конкретному району, являются просто возможными. По мере накопления свидетельств возможные отношения могут стать правдоподобными, затем вероятными и, наконец, несомненными, если некоторая выборка подтвердит выдвинутую гипотезу. А в случае игры с чужими игральными костями, если вы потеряете все свои победные очки, то определенно можете считать правдоподобным предположение, что вы попали в полосу неудач. Если же вы потеряете не только все очки, но и все деньги, с которых начинали игру, то можете определенно быть уверенными в том, что вам не везет. Но независимо от того, сколько раз вы бросили кости (даже если количество таких бросков приближается к миллиону), всегда остается шанс, что наблюдаемое явление диктуется законами статистики, поскольку для достижения 100%-й вероятности может потребоваться сделать бесконечное количество бросков. Выражения "0%-ная вероятность" и "100%-ная вероятность" фактически являются **парадоксальными**, т.е. содержащими в себе внутреннее противоречие. Дело в том, что случаи, в которых наблюдаются вероятности 0% и 100%, соответствуют полной **достоверности**, а достоверность не имеет ничего общего с вероятностью, поскольку в реальном мире достоверность равносильна убежденности. Если кто-то говорит: "Я наблюдал те же самые свидетельства, что и другие, но все еще уверен, что где-нибудь обязательно можно найти нечерного ворона", то он выражает убеждение, а истинные убеждения не могут измениться под действием фактов.

Эти соображения становятся очень важными, если вы создаете экспертную систему и пытаетесь формально выразить знания эксперта в правилах и фактах. На этапе **приобретения знаний**, когда проводятся собеседования с экспертом, вы можете услышать некоторые утверждения, которые внешне выглядят как знания, но фактически представляют собой просто убеждения (проявления полного доверия к определенным взглядам). Но необходимо соблюдать исключительную осторожность при попытке представить в экспертных системах убеждения, поскольку если убеждения не соответствуют действительности, то могут привести к недействительным заключениям. С другой стороны, если эксперт платит вам

именно за то, чтобы вы формально отразили его убеждения, то ему не понравится, если система не будет действовать в соответствии с ожиданиями.

Еще один парадокс связан с тем, что всегда существует вероятность услышать прогноз погоды, согласно которому "имеются 50%-ные шансы на то, что" пойдет дождь. Дело в том, что 50%-ные шансы означают **общее незнание** (total ignorance). Дождь либо будет, либо нет. Но для предсказания того, что дождь пойдет или не пойдет, не нужны теория вероятностей или бюро погоды. Тот факт, что дождь может идти или не идти, диктуется здравым смыслом. Аналогичным образом, в сводке бюро погоды иногда приходится слышать, что шансы на то, что пойдет сильный дождь, равны 100%, в то время как за окном действительно бушует гроза. Эта оценка шансов соответствует не вероятности, а достоверности.

Сети логического вывода должны не только обеспечивать представление отношений между узлами принятия решений и соответствующим им вероятностными действиями, но и обладать еще одной желаемой характеристикой — представлять **контексты**, которые блокируют распространение информации до тех пор, пока это распространение остается нежелательным. Использование контекстов позволяет разрешать или запрещать функционирование определенных частей сети логического вывода до того времени, как станет известно, что другие определенные части присутствуют, отсутствуют или неизвестны. Одно из назначений контекстов состоит в предотвращении необходимости формировать в системе вопросы к пользователю, касающиеся определенного свидетельства, до тех пор, пока не будет установлено, что это свидетельство действительно требуется. Это очень важно, поскольку люди раздражаются, если им задают вопросы, которые кажутся не относящимися к делу. Целью любой системы должно быть получение минимально необходимого объема информации для достижения действительного или по меньшей мере приемлемого заключения. Например, при посещении врача первым контекстом становится вопрос: "У вас есть медицинская страховка?" До получения ответа на этот вопрос могут не оправдаться затраты времени и денег на получение другой необходимой информации, например, такой: "Вы плохо себя чувствуете? Что у вас болит?" А если у вас нет медицинской страховки, то ваша судьба никого не интересует.

Основная идея контекстов состоит в том, что они позволяют управлять тем порядком, в котором система переходит от одного утверждения (напомним, что так именуются в целом гипотезы и свидетельства) к другому. Контексты устанавливают необходимые условия, которые должны быть доказаны прежде, чем появится возможность использовать некоторое утверждение. Контексты обозначаются штриховой линией со стрелкой, под которой указан диапазон коэффициентов достоверности как в случае, касающемся узлов FMGS и PT, показанных на рис. 4.19. Узел PT блокируется, если нет свидетельства со значениями достоверности в диапазоне от 0 до $+5$, согласно которому имеются **порфировые** породы с зернистостью от мелкой до средней. *Порфировыми породами* называются вулканические породы, **текстура** которых (определяемая по внешнему виду) состоит

из мелких кристаллов, внедренных в содержащую их породу, так называемую **матрицу**. Вулканические породы представляют собой породы, сформированные в результате затвердевания расплавленного твердого вещества, называемого магмой и поступающего из глубин земли. Вулканические породы некоторых типов, выступающие над поверхностью земли (называемые **интрузивными брекчиями**), являются свидетельством таких залежей полезных ископаемых, как медно-порфировые руды.

Итак, медно-порфировая руда состоит из мелких кристаллов меди, вкрапленных в скальную матрицу, содержащую эти кристаллы. Медно-порфировые руды — наиболее часто встречающая разновидность залежей меди. Если отсутствуют по меньшей мере небольшие кристаллы размерами от мелких до средних, то нет смысла задавать вопрос о том, наблюдается ли порфировая текстура, и поэтому экспертная система должна быть достаточно интеллектуальной, чтобы не задавать этот вопрос. Экспертная система, которая формирует ненужные вопросы, является неэффективной и вскоре начинает раздражать пользователя.

Диапазон значений достоверности от 0 до $+5$ для данного контекста означает, что работа с узлом PT не будет проводиться системой, если пользователь не покажет отсутствие свидетельства, соответствующего коэффициенту достоверности 0 или некоторому положительному значению, которое может достигать значения $+5$ (это значение показывает, что свидетельство определенно существует).

Все три описанных метода, позволяющих комбинировать или разрешать использование свидетельств, фактически представляют собой отношения между узлами. Все эти методы позволяют показать, как изменение вероятности одного утверждения влияет на другие утверждения.

Архитектура сети логического вывода

Формально **сеть логического вывода** может быть определена как ориентированный, ациклический граф, узлами которого являются утверждения, а дугами — меры неопределенности, такие как LS и LN. Сети, приведенные на рис. 4.21 *а–в*, могут рассматриваться как примеры, в которых правильно соблюдаются требования к архитектуре сети логического вывода, поскольку в этих сетях отсутствуют циклы. Обратите внимание на то, что в дереве логического вывода стрелки направлены к гипотезам, тогда как в отличие от этого в дереве структуры данных стрелки направлены от корня. Как было описано в главе 3, в ациклическом графе отсутствует возможность вернуться в начальную точку, следуя по стрелке. А на рис. 4.21, *г* имеется цикл, охватывающий четыре узла. Причина, по которой введен запрет на формирование циклов, обусловлена необходимостью предотвращения циклических рассуждений при установлении некоторой гипотезы. Но в системах, основанных на правилах, допустимым исключением является установление цикла путем обеспечения того, чтобы два правила активизировали друг друга до тех пор, пока не будет выполнено условие завершения. Еще одной широкой областью применения циклов является логика, которой руководствуются политики.

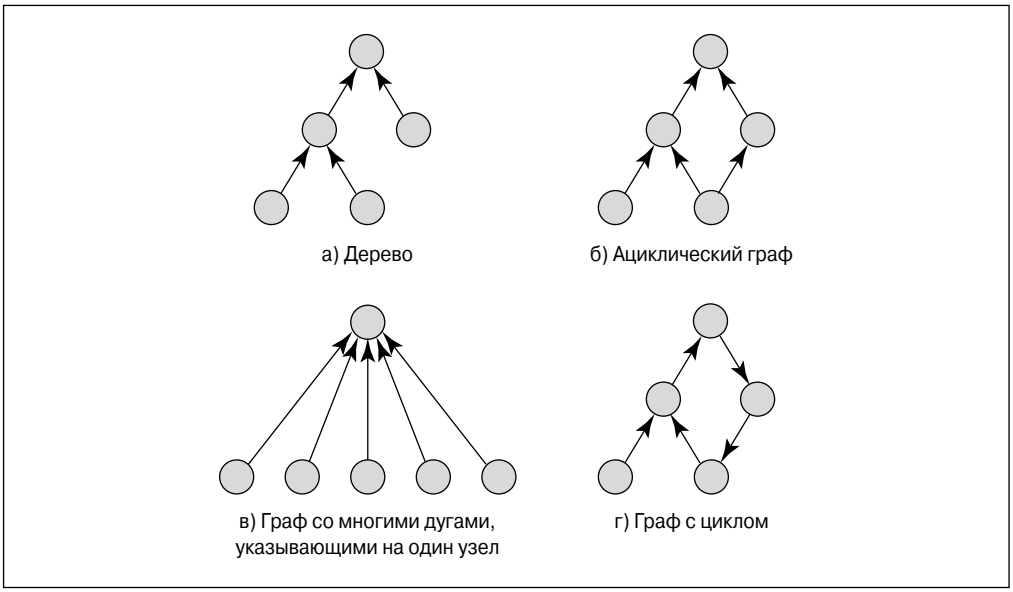


Рис. 4.21. Некоторые типы графов

На рис. 4.21, *в* показан граф такого типа, применение которого в сети логического вывода является нежелательным. Недостаток этого графа состоит в том, что в нем многочисленные части свидетельства вносят свой вклад в единственную гипотезу, а это способствует повышению шансов нежелательного взаимодействия частей свидетельства. Кроме того, решение задачи сбора слишком большого количества частей свидетельства может оказаться дорогостоящим. Как правило, лучше преобразовать структуру такого типа в структуру, более напоминающую дерево, с промежуточными гипотезами.

Сеть логического вывода системы PROSPECTOR можно также описать как **секционированную семантическую сеть**, в которой части сети сгруппированы в осмысленные блоки. Секционированные семантические сети были впервые разработаны Хендриксом (Hendrix), который стремился применить в семантических сетях такие мощные средства исчисления предикатов, как квантификация, импликация, отрицание, дизъюнкция и конъюнкция. Как было указано в главе 2, обычные семантические сети фактически предназначены для представления описательных знаний с помощью отношений. Безусловно, фреймы в большей степени приспособлены для представления причинных знаний, но, с другой стороны, фреймы нелегко использовать для представления логических отношений.

Основная идея секционированной семантической сети состоит в том, что такая сеть позволяет группировать множества узлов и дуг в абстрактные **пространства**, которые задают область определения представленных в них отношений. Знакомой аналогией пространства может служить область определения модуля

или пакета в языке структурированного программирования. Например, узел FRE (см. рис. 4.18) может рассматриваться как пространство со структурой, показанной на рис. 4.19.

Наибольшие возможности семантические сети проявляют при моделировании высказываний. И действительно, Хендрикс в своей докторской диссертации впервые разработал такие сети для представления естественного языка. Высказывания подобны формулировкам свидетельств и гипотез, представляющих собой узлы сетей логического вывода. В качестве простого примера секционированной семантической сети, представляющей высказывание, рассмотрим сеть для высказывания "существует компьютер с цветным экраном". Обратите внимание на то, что данное высказывание — экзистенциальное, поскольку в нем имеется квантор "существует".

На рис. 4.22 показано представление этого высказывания с помощью секционированной семантической сети, в которой используются три пространства. Пространство SPACE-1 состоит из некоторых общих взаимосвязанных концепций, касающихся компьютеров. Пространство SPACE-2 включает сведения о том конкретном компьютере, о котором идет речь и которому присвоен идентификатор COMPUTER-1. Пространство SPACE-3 включает конкретное отношение COMPONENTS-OF-1, объектом которого является конкретный цветной экран COLOR-SCREEN-1, рассматриваемый в данном высказывании. Каждая дуга с надписью "элемент" показывает, что одно состояние является элементом другого. Дуга с надписью "сущность" показывает, какой конкретный компьютер представляет собой сущность с данным конкретным цветным экраном.

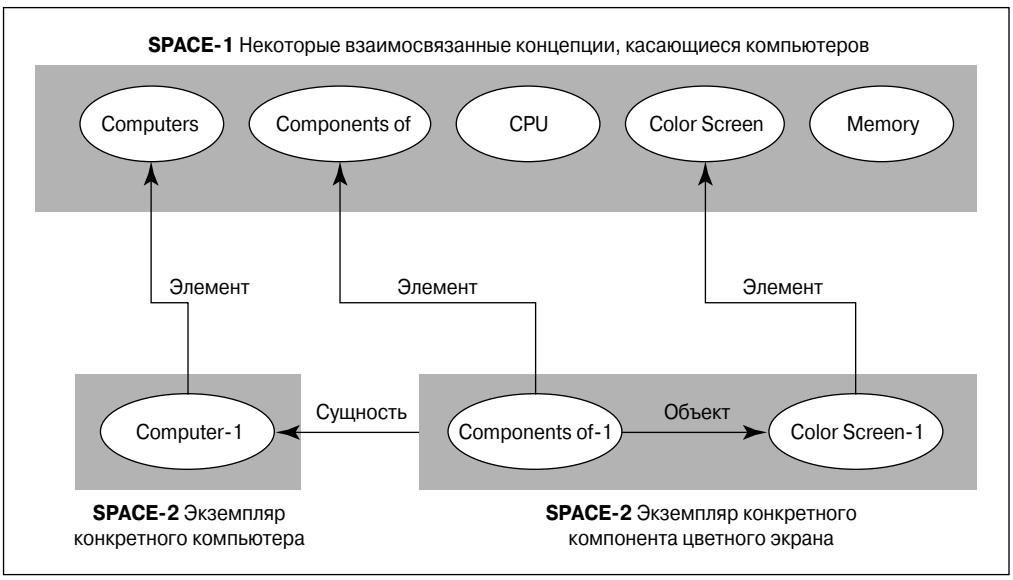


Рис. 4.22. Простая секционированная семантическая сеть с информацией о компьютере

Преимущество секционированной семантической сети демонстрируется на примере части правила системы PROSPECTOR, относящегося к узлу STIR, в котором используется свидетельство FMGS (рис. 4.23). Это правило можно сформулировать следующим образом: “Если имеется сущность E-1, кардинальность которой определяется как ABUNDANT (представленный в изобилии) и которая состоит из интрузивных пород, INTRUSIVE ROCKS, то консеквентом является правило STIR (Suggestive Texture of Igneous Rocks — перспективная текстура вулканических пород)”. Обратите внимание на то, что эта диаграмма упрощена, поскольку в нее не включено требование, что антецедент также должен указывать на наличие кристаллов с размерами от мелких до средних.

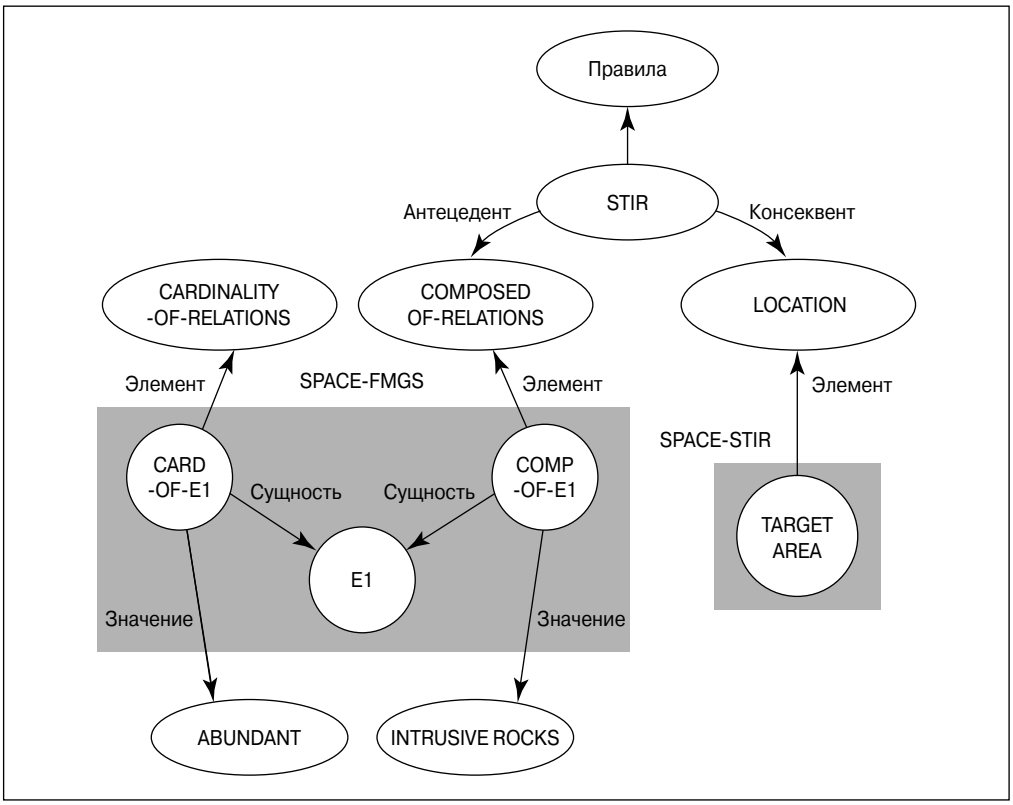


Рис. 4.23. Упрощенная часть секционированной семантической сети системы PROSPECTOR, относящаяся к правилу STIR, в котором используется свидетельство FMGS

Следует отметить, что на рис. 4.23 одна секционированная семантическая сеть составляет антецедент правила, а другая — консеквент. Безусловно, консеквент этого правила является тривиальным, но другие правила могут иметь более сложную структуру. Преимуществом секционированных семантических сетей является то, что с их помощью система может прийти к логическому выводу о наличии

отношений между узлами, которые не являются непосредственно связанными. Поэтому данная система располагает более глубокими знаниями, чем просто поверхностные знания.

4.16 Распространение вероятностей

Сети логического вывода, подобные применяемым в системе PROSPECTOR, имеют **статическую структуру знаний**. Это означает, что узлы и связи между узлами остаются постоянными, для того, чтобы в структуре знаний сохранялись отношения между узлами. В этом состоит принципиальное отличие сетей логического вывода от систем, основанных на правилах, поскольку в таких системах правила, шаблоны которых согласуются с фактами, помещаются в рабочий список правил, после чего осуществляется разрешение конфликтов, а затем выполняется правило с наивысшим приоритетом. Таким образом, система, основанная на правилах, обычно имеет **динамическую структуру знаний**, поскольку в ней отсутствуют такие постоянные соединения между правилами, которые установлены между узлами в сети логического вывода.

С другой стороны, хотя структура сети логического вывода остается неизменной, вероятности, связанные с каждым узлом гипотезы, по мере получения свидетельств изменяются. Но в действительности единственным изменением в любой сети логического вывода является изменение вероятностей. Основной характерной особенностью сети логического вывода является то, что в них по мере накопления свидетельств происходит изменение вероятностей, начиная с априорных вероятностей и заканчивая апостериорными вероятностями. Указанное изменение вероятностей происходит в направлении к верхней части сети, что позволяет в конечном итоге обосновать или опровергнуть гипотезу верхнего уровня, например, касающуюся существования залежей медно-порфировой руды (PCDA).

Рассмотрим некоторые этапы распространения вероятностей в пользу гипотезы PCDA на основе некоторого свидетельства. Этот пример должен служить в качестве демонстрации конкретного расчета по многим из формул, представленных в этой главе. Как показано на рис. 4.19, проследим за распространением вероятностей от узла свидетельства RCIB (the Region Contains Intrusive Breccias — в этом районе имеются интрузивные брекчии).

Если пользователь укажет, что интрузивные брекчии определенно присутствуют, то

$$P(E \mid e) = P(RCIB \mid e) = 1$$

Отметим, что, как показано на рис. 4.19, все значения LN для других свидетельств SMIR равны 1. Таким образом, если пользователь не дает никаких комментариев в отношении свидетельств, касающихся куполов (stock), валов (dike) и вулканических пробок (volcanic plug), то указанные свидетельства не вносят

свой вклад в гипотезу SMIR и поэтому могут быть проигнорированы во время решения данной задачи, посвященной изучению распространения вероятностей.

Причина, по которой все указанные свидетельства могут быть проигнорированы, обусловлена наличием следующей формулы для сложного свидетельства, которая обсуждалась в разделе 4.14:

$$O(H \mid e) = \left[\prod_{i=1}^{N} L_i\right] O(H)$$

В этой формуле $L_i = LS_i$ для всех свидетельств, которые являются заведомо истинными, а для свидетельств, являющихся заведомо ложными, используются значения $L_i = LN_i$. А если отсутствуют известные свидетельства, то значение $P(E_i \mid e)$ сводится к значению априорных шансов свидетельства, $P(E_i)$, поэтому значение $P(H \mid E_i)$ сводится к значению априорных шансов $P(H)$. Таким образом, становится справедливым следующее соотношение и указанные значения L_i не вносят никакого вклада в значение $O(H \mid E)$:

$$L_i = \frac{O(H \mid e_i)}{O(H)} = \frac{O(H)}{O(H)} = 1$$

Например, если известно, что истинно только свидетельство RCIB, то шансы, соответствующие SMIR, становятся следующими:

$$\begin{aligned} O(H \mid E) &= LS_{\text{RCVP}} LS_{\text{RCAD}} LS_{\text{RCS}} LS_{\text{SMIRA}} LS_{\text{RCIB}} = \\ &= 1 \cdot 1 \cdot 1 \cdot 1 \cdot 20 \\ O(\text{SMIR} \mid \text{RCIB}) &= 20 \end{aligned}$$

Еще одной интересной особенностью свидетельства SMIR является наличие узла OR, называемого SMIRA. Заслуживает внимания то, что узел SMIRA связан со всеми теми же свидетельствами, с которыми связан узел SMIR. Если имеется какое-либо свидетельство, вносящее свой вклад в значение SMIR, такое как RCS, RCAD, RCIB или RCVP, то узел SMIRA типа OR не вносит никакого вклада в значение узла SMIR, поскольку коэффициент LS для узла SMIRA равен 1. Но если свидетельства RCS, RCAD, RCIB и RCVP отсутствуют, то благодаря значению LN узла SMIRA, равному 0,0002, значение вероятности SMIR по существу становится равным нулю. Фактически это означает, что отсутствие одного или нескольких свидетельств, касающихся SMIR, не имеет значения ($LS = 1$), но отсутствие всех свидетельств является очень важным, поскольку влечет за собой исключение узла SMIR из рассмотрения.

Проектировщик модели имеет возможность вводить узлы OR, узлы AND, а также значения LS и LN, тем самым приобретает способность регламентировать способ влияния свидетельства на гипотезу. Но в некоторых сложных случаях

введение большого количества узлов OR и AND для выполнения требований к свидетельствам приводит к загромождению структуры сети логического вывода до такой степени, что ее становится трудно понять. Кроме того, введение таких вспомогательных узлов связано с необходимостью предусматривать дополнительную проверку модели.

Согласно рис. 4.20, для интрузивных брекчий (RCIB) значение $LS = 20$, а значение $P(\text{SMIR}) = 0.03$. Таким образом:

$$\text{odds} = O = \frac{P}{1-P}$$

Это означает, что априорные шансы SMIR таковы:

$$O(\text{SMIR}) = \frac{0.03}{1-0.03} = 0.0309$$

Если свидетельство E является достоверным, то апостериорные шансы являются следующими:

$$O(H \mid E) = LS\, O(H)$$
$$O(\text{SMIR} \mid \text{RCIB}) = 20 \cdot 0.0309 = 0.618$$

апостериорная вероятность вычисляется с помощью следующей фундаментальной формулы шансов:

$$P = \frac{O}{1+O}$$
$$P(H \mid E) = \frac{O(H \mid E)}{1+O(H \mid E)}$$
$$P(H \mid E) = \frac{0.618}{1+0.618} = 0.382$$

поэтому апостериорная вероятность SMIR при наличии достоверного свидетельства RCIB равна:

$$P(\text{SMIR} \mid \text{RCIB}) = 0.382$$

Априорные шансы HYPE являются таковыми:

$$O(\text{HYPE}) = \frac{0.01}{1-0.01} = 0.0101$$

На этом этапе вычислений могло бы возникнуть стремление использовать следующие соотношения для вычисления шансов гипотезы HYPE, обусловленных

значением SMIR, но это было бы неправильным:

$$O(H \mid E) = LSO(H)$$
$$O(\text{HYPE} \mid \text{SMIR}) = \text{LSSMIR}O(\text{SMIR}) =$$
$$= 300 \cdot \frac{0.0101}{1 + 0.0101} = 3.00$$

Если свидетельство E является достоверным, то становится истинной следующая формула:

$$O(H \mid E) = LS\,O(H)$$

Эта формула выражает изменение вероятности гипотезы, после принятия предположения о том, что свидетельство является достоверным. Но значение SMIR достоверно не известно. В действительности вероятность гипотезы SMIR при наличии достоверного свидетельства об обнаружении интрузивных брекчий является таковой:

$$P(\text{SMIR} \mid \text{RCIB}) = 0.382$$

Это означает, что правдоподобие убеждения в том, что гипотеза SMIR истинна, составляет 38,2%.

В действительности нам требуется рассчитать вероятность гипотезы HYPE на основе недостоверного свидетельства SMIR, которая имеет значение $P(\text{HYPE} \mid \text{RCIB})$. Один из способов вычисления этого значения состоит в использовании формулы для $P(H \mid e)$, которая рассматривалась в разделе 4.14, имеющей такой вид:

$$P(H \mid e) = \begin{cases} P(H \mid E') + \dfrac{P(H) - P(H \mid E')}{P(E)} P(E \mid e) \\ \quad \text{для } 0 \leqslant P(E \mid e) < P(E) \\ P(H) + \dfrac{P(H \mid E) - P(H)}{1 - P(E)} [P(E \mid e) - P(E)] \\ \quad \text{для } P(E) \leqslant P(E \mid e) \leqslant 1 \end{cases}$$

В этой формуле неопределенность свидетельства SMIR выражается следующим образом:

$$P(E \mid e) = P(\text{SMIR} \mid \text{RCIB}) = 0.382$$

а требуемое значение вероятности — формулой

$$P(H \mid e) = P(\text{HYPE} \mid \text{SMIR})$$

В данной формуле известно только такое значение:

$$P(H \mid E) = \frac{O(H \mid E)}{1 + O(H \mid E)} = \frac{3.00}{1 + 3.00} = 0.75$$

Кроме того, согласно рис. 4.19, априорные шансы равны:

$$P(H) = P(\text{HYPE}) = 0.01$$
$$P(E) = P(\text{SMIR}) = 0.03$$

Безусловно, $P(E \mid e) > P(E)$, поэтому для вычисления значения $P(H \mid E)$ можно воспользоваться формулой для выражения $P(E) \leqslant P(E \mid e) \leqslant 1$ следующим образом:

$$P(H \mid e) = 0.01 + \frac{(0.75 - 0.01)(0.382 - 0.03)}{1 - 0.03}$$
$$P(\text{HYPE} \mid \text{RCIB}) = 0.279$$

Это значение представляет собой апостериорную вероятность HYPE.

Такое распространение вероятностей продолжается вверх по сети логического вывода. На данном этапе вероятность $P(\text{HYPE} \mid \text{RCIB})$ рассматривается как недостоверное свидетельство для узла FLE. Апостериорная вероятность $P(\text{FLE} \mid \text{RCIB})$ вычисляется с использованием той же формулы, в которой значения откорректированы для FLE:

$$P(E) = P(\text{HYPE}) = 0.01$$
$$P(E \mid e) = P(\text{HYPE} \mid \text{RCIB}) = 0.279$$
$$P(H) = P(\text{FLE}) = 0.005$$
$$O(H \mid E) = O(\text{FLE} \mid \text{HYPE}) = \text{LSHYPE}\ O(\text{FLE}) = 200 \cdot 0.005 = 1$$
$$P(H \mid E) = P(\text{FLE} \mid \text{HYPE})$$

А поскольку $P = \text{odds}/(1 + \text{odds})$, можно выполнить следующие расчеты:

$$P(H \mid E) = O(\text{FLE} \mid \text{HYPE})/(1 + O(\text{FLE} \mid \text{HYPE})) =$$
$$= 1/(1 + 1) =$$
$$= 0.5$$
$$P(H \mid e) = P(\text{FLE} \mid \text{RCIB}) =$$
$$= 0.005 + \frac{(0.5 - 0.005)(0.279 - 0.01)}{1 - 0.01}$$
$$P(\text{FLE} \mid \text{RCIB}) = 0.140$$

4.17 Резюме

В настоящей главе вначале рассматривались основные понятия формирования рассуждений в условиях неопределенности и возможные типы ошибок, вызванные неопределенностью. Приведена краткая сводка элементарных сведений по

классической теории вероятностей. Описаны различия между классической и другими теориями вероятностей, такими как теории экспериментальных и субъективных вероятностей. Представлены методы комбинирования вероятностей и теорема Байеса. Показана связь между доверием (убеждениями) и вероятностью, а также раскрыт смысл понятия правдоподобности.

Для иллюстрации того, как концепции теории вероятностей используются в реальных системах, была подробно проанализирована классическая экспертная система PROSPECTOR. Система PROSPECTOR использовалась также для вводного описания сетей логического вывода и секционированных семантических сетей.

Материал, представленный в данной главе, позволяет сделать такой основной вывод, что классическая теория вероятностей превосходно подходит для решения задач, касающихся идеальных систем, но не всегда применима при решении таких практических задач, как разведка полезных ископаемых с помощью методов, применяемых в системе PROSPECTOR. Безусловно, теория вероятностей используется как исходный подход, обеспечивающий решения задач, характеризующихся наличием неопределенности, но для обнаружения конкретных полезных ископаемых требуется так дополнить теорию вероятностей, чтобы она позволяла учитывать доверительные коэффициенты и формулы нечеткой логики, так как лишь после этого экспертная система приобретает возможность правильно предсказывать наличие месторождений минерального сырья с использованием известных моделей. После указанной модификации система PROSPECTOR использовалась для обработки заранее неизвестных ей данных и успешно предсказала наличие месторождения молибдена стоимостью 100 миллионов долларов.

Следует отметить, что для обнаружения месторождений полезных ископаемых других типов требуются другие модели геологических характеристик, коэффициенты достоверности и формулы нечеткой логики. В этой ситуации нет ничего обескураживающего, поскольку она просто отражает тот факт, что различные залежи полезных ископаемых формировались на протяжении естественной истории по-разному, поэтому для их обнаружения требуются другие свидетельства и теории. Именно поэтому необходимо иметь возможность привлекать знания экспертов, касающиеся различных предметных областей. Эксперт, который хорошо зарекомендовал себя при поиске залежей молибдена, не обязательно должен обладать способностью успешно находить месторождения нефти.

Следует учитывать одно важное соображение. Необходимо стремиться к тому, чтобы экспертная система соответствовала реальному миру, а не пытаться подогнать реальный мир к существующей экспертной системе, поскольку такая попытка неизбежно окончится неудачей. В конечном итоге все теории основаны на аксиомах, представляющих собой недоказанные истины, такие как аксиома Евклида, согласно которой параллельные прямые в бесконечности не пересекаются. Безусловно, это утверждение трудно оспаривать, если речь идет об евклидовой плоскости, но указанная аксиома не соблюдается в геометрии сферических тел

или тел другой формы, не являющихся плоскими. Теории неопределенности также основаны на аксиомах. Тот факт, что, рассуждая о полезных ископаемых, мы вынуждены вводить дополнительные факты, такие как доверительные коэффициенты и формулы нечеткой логики, просто означает, что мы не знаем правильных аксиом, поскольку не имеем представления о том, как действительно формировались залежи полезных ископаемых миллионы лет тому назад и какой конкретный химический состав имела в этом месте земная твердь.

Кроме того, в данной главе обсуждался смысл терминов *невозможный*, *возможный*, *правдоподобный*, *вероятный* и *достоверный*. Проводя собеседование с экспертом, важно помнить смысл этих терминов, чтобы суметь отделить вероятность от убежденности, о чем шла речь в примере с лево- и правокрылыми воронами. Особое внимание следует уделять высказываниям, рассматриваемым как достоверные, поскольку за ними могут скрываться обыденные суждения, внешне кажущиеся обоснованными. Например, согласно здравому смыслу, трудно оспаривать утверждение, что условная вероятность прихода к финишу лошади в скачках равна 100%, если лошадь жива, $P(\text{Finish} \mid \text{Live}) = 100\%$. Обратите внимание на то, что здесь не говорится о том, что лошадь обязательно победит, а лишь придет к финишу. И тем более здравый смысл говорит о том, что участник соревнований должен быть живым, чтобы выиграть гонку. Это несомненно, не правда ли? Но не рекомендуем держать пари на то, что этот закон жизни никогда не нарушается.

Как уже упоминалось ранее, политическая логика не подчиняется законам каких-либо других видов логики. В октябре 2000 года губернатор штата Миссури Мэл Карнахан погиб в авиационной катастрофе в одной из поездок, проводимых во время избирательной кампании. Тем не менее его противник, сенатор Джон Эшкрофт, потерпел поражение на выборах, проведенных в ноябре. Избиратели проголосовали за покойного сенатора, и его кресло заняла вдова, миссис Карнахан (`http://www.specialednews.com/washwatch/washnews/election111000.html`). Это — еще одно свидетельство в пользу того, что в наше время, когда доминируют рассуждения на основе “здравого смысла”, даже смерть перестала быть определяющим фактором.

Задачи

4.1. Выполните следующие задания.

а) Докажите аддитивный закон вероятности, используя только три аксиомы вероятности:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Подсказка. Любое событие $X \cup Y$ может быть записано в форме $X \cup (X' \cap Y)$, где X и $X' \cap Y$ являются взаимоисключающими. Кроме того, представьте B как объединение двух взаимоисключающих множеств.

б) Предположим, что имеются два компьютера, которые работают либо правильно, либо неправильно. Определите с помощью аддитивного закона вероятность того, что по меньшей мере один из них работает правильно.

4.2. При условии, что имеют место события A и B, которые могут перекрываться, определите с помощью аксиом вероятности следующие значения вероятности в терминах множеств:

а) Ни A, ни B.

б) Или A, или B, но не оба вместе (исключительное ИЛИ).

4.3. Как показано в табл. 4.14, в трех магазинах автоматического питателя содержатся и исправные, и неисправные компоненты.

Таблица 4.14. Распределение компонентов по магазинам

Магазин	Количество исправных компонентов	Количество неисправных компонентов
1	8	2
2	3	1
3	2	2

По истечении продолжительного периода времени из магазина 1 извлекаются 20% компонентов, из магазина 2 — 30% и из магазина 3 — 50%.

а) Постройте дерево вероятностей.

б) Если извлечен неисправный компонент, то каковы вероятности его извлечения из каждого магазина? Составьте таблицу вероятностей и покажите в ней числовые результаты, а также априорные, условные, совместные и апостериорные вероятности.

4.4. Программист намеревается купить жесткий диск и пытается выбрать наилучшую среди трех моделей. В табл. 4.15 показаны его предпочтения и вероятности отказов в течение одного года. (Предпочтения выражаются функцией от стоимости, быстродействия, емкости и надежности.)

Таблица 4.15. Характеристики жестких дисков

Модель	Вероятность выбора	Вероятность отказа
X	0,3	0,1
Y	0,5	0,3
Z	0,2	0,6

а) Составьте таблицу вероятностей и покажите в ней числовые результаты, а также априорные, условные, совместные и апостериорные вероятности.

б) Какова вероятность того, что в течение одного года произойдет отказ жесткого диска каждой модели?

в) Какова вероятность того, что в течение одного года произойдет отказ жесткого диска любой модели?

г) Если в течение одного года произошел отказ жесткого диска, то какова вероятность того, что это был жесткий диск модели X, Y или Z?

4.5. Массовая проверка — это недорогой способ проверить большие группы людей на наличие определенного заболевания. Более дорогостоящая, но вместе с тем более точная проверка показывает, что такое заболевание имеется у 1% всех людей. А массовая проверка показывает наличие заболевания у 90% из тех людей, кто действительно его имеет (положительный результат проверки), и у 20% из тех, кто его не имеет (ложно положительный результат проверки).

а) Составьте таблицу вероятностей и покажите в ней числовые результаты, а также априорные, условные, совместные и апостериорные вероятности.

б) Какова процентная доля людей, для которых проверка показала положительный результат, но они фактически не имеют заболевания (ложно положительный результат проверки)?

в) Какова процентная доля людей, для которых проверка показала отрицательный результат, но они фактически имеют заболевание (ложно отрицательный результат проверки)?

4.6. Чтобы показать, что наличие попарной независимости не обязательно означает наличие взаимной независимости, определите следующие события для броска двух игральных костей:

```
A - четное количество очков на первой игральной кости
B - четное количество очков на второй игральной кости
C - четная сумма очков
```

а) Нарисуйте выборочное пространство для двух игральных костей. Покажите в нем события A, B и $A \cap B$.

б) Определите состав элементов множеств $A \cap B$, C, $A \cap C$, и $B \cap C$.

в) Чему равна вероятность $P(C)$?

г) Покажите, что для следующих формул соблюдается условие попарной независимости:

$$P(A \cap B) = P(A)P(B)$$
$$P(A \cap C) = P(A)P(C)$$
$$P(B \cap C) = P(B)P(C)$$

д) Докажите справедливость следующей формулы, которая показывает, что наличие попарной независимости не означает наличие взаимной независимости:

$$P(A \cap B \cap C) = P(A \cap B) \neq P(A \mid B)P(C)$$

4.7. Даны два взаимоисключающих множества, A и B:

а) Что представляет собой вероятность $P(A \mid B')$ в терминах A и B (а не в терминах A и B')?

б) Каково числовое значение выражения $P(A' \mid B)$?

в) Каково числовое значение выражения $P(A \mid B)$?

г) Что представляет собой вероятность $P(A' \mid B')$ в терминах A и B (а не в терминах A' и B')?

д) Что означают следующие выражения:

1) $P(A \mid B) + P(A' \mid B)$
2) $P(A \mid B') + P(A' \mid B')$

4.8. Выполните следующие задания.

а) Докажите теорему:

$$P(A \cap B \cap C) = P(A \mid B \cap C)P(B \mid C)P(C)$$

б) Докажите теорему:

$$P(A \cap B \mid C) = P(A \mid B \cap C)P(B \mid C)$$

4.9. Жесткий диск может работать со сбоями, имея либо неисправность F_1, либо неисправность F_2, но не обе неисправности одновременно. Возможными симптомами являются перечисленные ниже, причем вероятность наличия неисправности F_2 в три раза выше по сравнению с F_1.

$$A = \{\text{ошибка записи, ошибка чтения}\}$$
$$B = \{\text{ошибка чтения}\}$$

Для жесткого диска этого типа:

$$P(A \mid F_1) = 0.4$$
$$P(B \mid F_1) = 0.6$$
$$P(A \mid F_2) = 0.2$$
$$P(B \mid F_2) = 0.8$$

Каковы вероятности того, что в жестком диске имеется неисправность F_1 или F_2?

4.10. Ниже приведена матрица переходов, которая показывает, как происходит смена моделей жестких дисков в течение года.

$$\begin{array}{cc} & \text{Следующий год} \\ & \begin{array}{ccc} X & Y & Z \end{array} \\ \text{Текущий год} \begin{array}{c} X \\ Y \\ Z \end{array} & \begin{bmatrix} .5 & .5 & 0 \\ .25 & .5 & .25 \\ 0 & .5 & .5 \end{bmatrix} \end{array}$$

а) Предположим, что первоначально количество жестких дисков модели X, находящихся в эксплуатации, составляет 50%, жестких дисков Y — 25% и жестких дисков Z — 25%. Чему будет равна процентная доля дисков каждой модели, находящихся в эксплуатации, через 1 год, 2 и 3 года?

б) Определите матрицу установившегося состояния.

4.11. Выполните следующие задания.

а) Предположим, что случайным образом выбраны N человек. Какова вероятность того, что среди этих людей никакие двое не имеют один и тот же день рождения?

б) Каково значение этой вероятности для 30 человек?

4.12. Исходя из предположения об условной независимости свидетельств E_1 и E_2, покажите, что с учетом правила, в котором применяется конъюнкция известных свидетельств:

```
IF E1 AND E2 THEN H
```

справедливы следующие формулы:

а) $$P(H \mid E_1 \cap E_2) = \frac{P(E_1 \mid H)P(E_2 \mid H)}{P(E_1 \mid H)P(E_2 \mid H') + O(H')P(E_1 \mid H')P(E_2 \mid H')}$$

б) $$P(H \mid E_1 \cap E_2) = \frac{P(H \mid E_1)P(H \mid E_2)}{P(H \mid E_1)P(H \mid E_2) + O(H)P(H' \mid E_1)P(H' \mid E_2)}$$

4.13. С учетом следующего правила дизъюнкции свидетельств:

```
IF E1 OR E2 OR ... EN THEN H
```

покажите, что на основе теории вероятностей и предположения об условной независимости свидетельств можно вывести следующую формулу:

$$P(E \mid e) = 1 - \prod_{i=1}^{N}(1 - P(E_i \mid e))$$

В этой формуле E — свидетельство; e — частичные свидетельства, относящиеся к E.

4.14. Предположим, что дано свидетельство E, являющееся антецедентом следующего правила:

```
IF E THEN H
```

Составьте перечисленные ниже выражения нечеткой логики для $P(E \mid e)$.

а) $E = E_1$ OR $(E_2$ AND $E_3')$

б) $E = (E_1$ AND $E_2)$ OR $(E_3$ AND $E_4)$

в) $E = (E_1'$ AND $E_2')$ OR E_3

г) $E = E_1'$ AND $(E_2'$ OR $E_3)$

д) $E = E_1'$ OR $(E_2 AND E_3)$

4.15. Предположим, что в сети логического вывода, относящейся к задаче поиска месторождений медно-порфировой руды, значение RCS заведомо истинно, а значение RCAD заведомо ложно. Чему равна модифицированная вероятность FRE?

4.16. Рассмотрите следующую задачу семантической индукции — определение очередных трех чисел в последовательности $2, 8, 8, \ldots$. При проведении семантической индукции требуется ключ, который позволил бы выявить подсказку и восстановить последовательность; в данном случае ключом является слово “грузовик”.

Глава 5

Нестрогие рассуждения

5.1 Введение

В настоящей главе продолжается обсуждение тематики формирования рассуждений в условиях неопределенности, начатое в главе 4. Изложенный в главе 4 подход к формированию рассуждений в условиях неопределенности был основан на теореме Байеса и вероятностных рассуждениях, а в данной главе рассматривается несколько других подходов, позволяющих действовать в условиях неопределенности. В частности, при создании многих приложений (таких как обработка изображений и управление; обучение машины правилам классификации на основе нечетких правил; кластеризация и функциональная аппроксимация) чрезвычайно успешными оказались методы, осуществляемые на базе нечеткой логики [15], [50].

Для разработки экспертных систем, в которых используется большой объем нечетких рассуждений, были созданы две версии языка CLIPS. Первой из них является версия fuzzyCLIPS компании NRC (`http://ai.iit.nrc.ca/IR_public/fuzzy/`). Компанией NRC были также созданы инструментарии FuzzyJToolkit для платформы Java и FuzzyJess (версия CLIPS, основанная на языке Java). Вторая версия — это язык Fuzzy CLIPS, разработанный компанией Togai InfraLogic. Разработчики этого языка предлагают воспользоваться многими дополнительными ссылками на другие информационные ресурсы, посвященные нечеткой логике (`http://www.ortech-engr.com/fuzzy/togai.html`). Как показывает даже самый краткий поиск в Web и книгах, в настоящее время экспертные системы на основе нечеткой логики нашли чрезвычайно широкое распространение [86].

Как описано в главе 4, теория вероятностей была первоначально разработана с учетом проблематики идеальных азартных игр, в которых один и тот же эксперимент можно воспроизводить бесконечное количество раз. И действительно,

в математике теорию вероятностей часто называют теорией **воспроизводимой неопределенности**. На первый взгляд такое сравнение выглядит парадоксальным, поскольку трудно понять, как можно снова и снова воспроизводить нечто неопределенное. (Безусловно, речь не идет о рождении детей.) Термин "воспроизводимый" в упомянутом определении имеет статистический смысл, согласно которому берется большая совокупность, в ходе испытаний из этой совокупности формируются выборки и результаты выборок усредняются. Безусловно, было бы великолепно иметь возможность точно предсказать, что́ произойдет при следующем броске игральных костей, но мы можем определить только шансы на получение тех или иных результатов, но не имеем возможности сформулировать какие-либо достоверные утверждения.

В главе 4 был приведен пример успешного применения теории субъективных вероятностей в системе PROSPECTOR, но потребности создания многих других приложений могут быть гораздо лучше удовлетворены с использованием других теорий. Такие альтернативные теории были специально разработаны так, чтобы с их помощью можно было учитывать числовые оценки степени убежденности человека в справедливости некоторой гипотезы, а не использовать классическую частотную интерпретацию вероятности. Все эти теории могут служить примерами **нестрогих рассуждений**, в которых антецеденты, заключения и даже смысл самих правил до некоторой степени всегда остаются неопределенными.

5.2 Неопределенность и правила

В данном разделе приведен общий обзор понятий, касающихся продукционных правил и неопределенности, а в следующих разделах рассматриваются другие методы учета неопределенности в правилах и другие **интеллектуальные системы**. В настоящее время термин "интеллектуальные системы" широко используется и, строго говоря, распространяется на любые приложения, в которых применяется искусственный интеллект. На практике этот термин часто служит для описания любой системы, способной оперировать с неопределенной информацией.

Источники неопределенности в правилах

На рис. 5.1 схематически показано, какую роль играет понятие неопределенности в системах, основанных на правилах. Неопределенность может проявляться в отдельных правилах, возникать в ходе разрешения конфликтов и обнаруживаться из-за несовместимости в консеквентах правил. Задача по минимизации или, по возможности, устранению таких неопределенностей стоит перед инженером по знаниям.

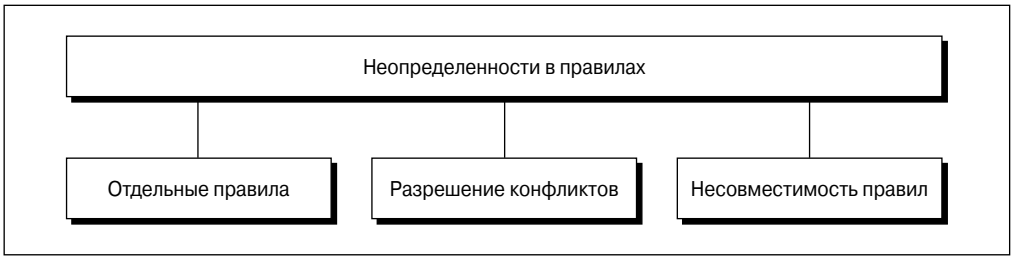


Рис. 5.1. Основные неопределенности, обнаруживаемые в экспертной системе, основанной на правилах

Задача минимизации неопределенности в отдельных правилах решается в процессе **верификации** правил. Как было указано в разделе 3.15, верификация — это процесс проверки правильности структурных блоков системы, а в системе, основанной на правилах, структурными блоками являются правила.

Но лишь достижение того, что отдельные правила будут действовать безукоризненно, не позволяет обеспечить получение правильного ответа от всей системы, поскольку формирование цепей логического вывода может нарушаться из-за несовместимости отдельных правил. В связи с этим возникает необходимость проведения аттестации системы. А что касается системы, основанной на правилах, то процесс аттестации отчасти предусматривает минимизацию неопределенности в цепях логического вывода. Итак, верификация может рассматриваться как минимизация локальных неопределенностей, тогда как аттестация скорее связана с минимизацией глобальной неопределенности во всей экспертной системе. Проведя грубую аналогию со строительством, можно отметить, что верификацией является проверка того, построен ли мост должным образом, с точки зрения применения качественных материалов и выполнения добросовестной сборки. С другой стороны, аттестация — это проверка того, может ли мост выдерживать требуемую дорожную нагрузку и, самое главное, построен ли мост действительно в нужном месте. Как описано более подробно в главе 6, для создания качественной экспертной системы необходимы и верификация, и аттестация.

На рис. 5.2 приведена подробная схема классификации неопределенностей отдельных правил, которые перед этим были показаны в виде общей схемы на рис. 5.1. Кроме возможных ошибок, связанных с созданием правил (см. раздел 4.3), некоторые неопределенности связаны с присваиванием значений правдоподобия. Как указано в главе 4, с точки зрения вероятностных рассуждений такие неопределенности связаны со значениями достаточности, LS, и необходимости, LN. Значения LS и LN основаны на оценках, предоставляемых людьми, поэтому с ними связана неопределенность. Кроме того, неопределенность связана с правдоподобием консеквента. Когда речь идет о вероятностных рассуждениях,

соответствующее выражение записывается как $P(H \mid E)$ для определенных свидетельств и как $P(H \mid e)$ — для неопределенных свидетельств.

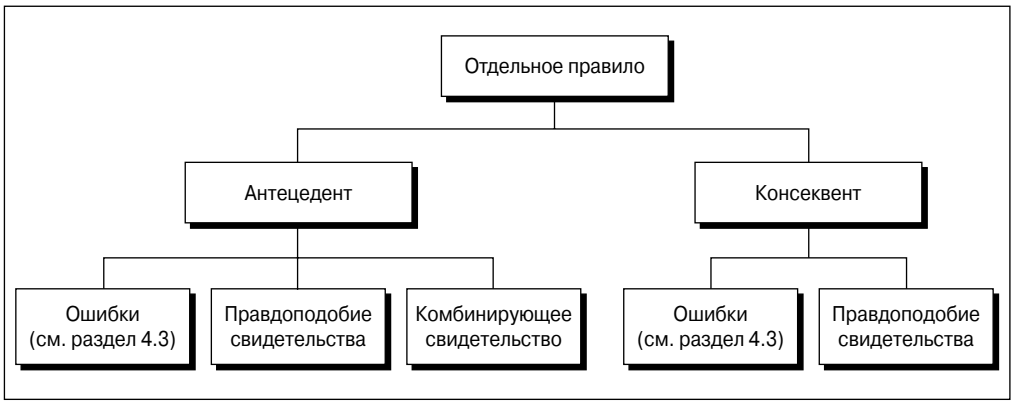


Рис. 5.2. Неопределенности в отдельных правилах

Еще одним источником неопределенности становится комбинирование свидетельств. В частности, приходится часто искать ответ на вопрос о том, следует ли применять способы комбинирования свидетельств, заданные с помощью приведенных ниже выражений, или использовать какие-то другие возможные способы создания логических комбинаций с использованием операций AND, OR или NOT.

как E_1 AND E_2 AND E_3
или как E_1 AND E_2 OR E_3
или как E_1 AND NOT E_2 OR E_3

Отсутствие надежных теоретических оснований

Как упоминалось в главе 4, при попытке вводить в вероятностную систему произвольные формулы, например, относящиеся к нечеткой логике, возникают проблемы. Дело в том, что в подобных случаях создаваемая экспертная система не имеет надежного теоретического фундамента, базирующегося на классической теории вероятностей, и поэтому становится реализацией произвольно взятых методов, которые действуют только в ограниченных ситуациях. Опасность применения произвольных методов состоит в том, что с ними не связана законченная теория, которая позволяла бы руководить созданием приложения или определять, действительно ли эти методы подходят в той или иной ситуации.

Одним из примеров использования произвольных методов, который рассматривался в настоящей книге, является применение коэффициентов LS и LN в сетях логического вывода. С точки зрения теории сеть логического вывода с N узлами представляет собой вероятностную систему с пространством событий, состоящим

из N возможных событий. Это означает, что количество возможных вероятностей равно 2^N. Но на практике, в ситуациях, складывающихся в реальном мире, известны лишь немногие из этих вероятностей. Коэффициенты LS и LN используются для наиболее значимых (по нашему мнению) дуг между узлами, что позволяет избавиться от необходимости вычислять все вероятности. Применение коэффициентов LS и LN для наиболее значимых дуг позволяет существенно уменьшить сложность сети.

Таким образом, коэффициенты LS и LN лежат в основе способа сокращения объема работы по представлению вероятностей, но этот способ сокращения обладает тем недостатком, что не гарантирует равенства единице суммы условных вероятностей для каждой гипотезы в сети. Этот недостаток может не иметь существенного значения для пользователя, если его целью является просто получение данных об относительном ранжировании всех гипотез в сети. Например, пользователь, заинтересованный в оценке гипотезы о наличии медно-порфировой руды, может быть вполне удовлетворен, узнав о том, что значение узла, показывающего правдоподобие гипотезы, благоприятной с точки зрения обнаружения меди, является наибольшим, даже если оно не равно 1. Но не менее важной является также абсолютная вероятность. Даже если гипотеза о наличии меди после ранжирования гипотез стала на первое место, стоит ли бурить разведывательную скважину, если вероятность обнаружения меди все-таки очень низка, скажем, равна 0,00000001?

Взаимодействия правил

Еще один источник неопределенности связан с необходимостью разрешения конфликтов между оценками неопределенности. В частности, потенциальный источник ошибок возникает, если инженер по знаниям задает **явный приоритет** правил, поскольку заданные им значения приоритета могут оказаться неоптимальными или неправильными. Еще одна проблема возникает при разрешении конфликтов из-за наличия **неявного приоритета** правил.

Разрешение конфликтов является лишь частью более крупного источника неопределенности, обусловленного взаимодействиями правил. Как показано на рис. 5.3, взаимодействие правил зависит от разрешения конфликтов и **совместимости правил**. Неопределенность, связанная с совместимостью правил, вызывается пятью основными причинами.

Одной из причин неопределенности является потенциальная **противоречивость правил**. При этом может происходить запуск правил с противоречивыми консеквентами. Такая ситуация возникает, если антецеденты не заданы должным образом. В качестве очень простого примера предположим, что в базе знаний заданы следующие два правила:

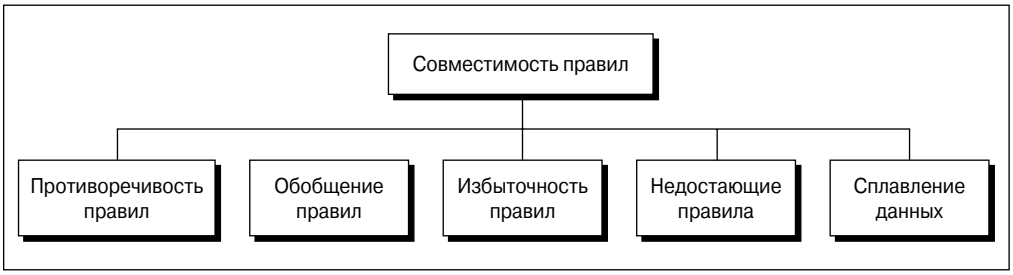


Рис. 5.3. Неопределенность, связанная с совместимостью правил

(1) IF происходит пожар THEN примените для его тушения воду

(2) IF происходит пожар THEN не применяйте для его тушения воду

Правило (1) подходит для тушения обычных пожаров, например, связанных с возгоранием деревянных конструкций, а правило (2) относится к тушению пожаров, в которых горит нефть или консистентная смазка. Проблема состоит в том, что не указаны достаточно точно антецеденты, соответствующие категории пожара. Если обнаруживается факт “происходит пожар”, то выполняются оба правила с противоречивыми консеквентами, в результате чего неопределенность увеличивается.

Вторым источником неопределенности становится **обобщение правил**. Одно правило обобщается другим, если часть его антецедента является подмножеством другого правила. Например, предположим, что в базе знаний представлены следующие два правила и оба эти правила имеют одинаковое заключение:

(3) IF E_1 THEN H

(4) IF E_1 AND E_2 THEN H

Если обнаруживается только свидетельство E_1, то проблема не возникает, поскольку активизируется только правило (3). Если же обнаруживаются оба свидетельства, E_1 и E_2, то активизируются оба правила, (3) и (4), поэтому возникает необходимость в разрешении конфликтов.

Разрешение конфликтов

В процессе разрешения конфликтов возникает неопределенность, относящаяся к приоритетам правил. Как показано на рис. 5.4, такая неопределенность зависит от целого ряда факторов.

Первым фактором является то, каковы особенности применяемого командного интерпретатора или инструментального средства. В инструментальных средствах экспертных систем, в которых используется rete-алгоритм, более высокий приоритет приобретает правило с более конкретными шаблонами. Это действительно

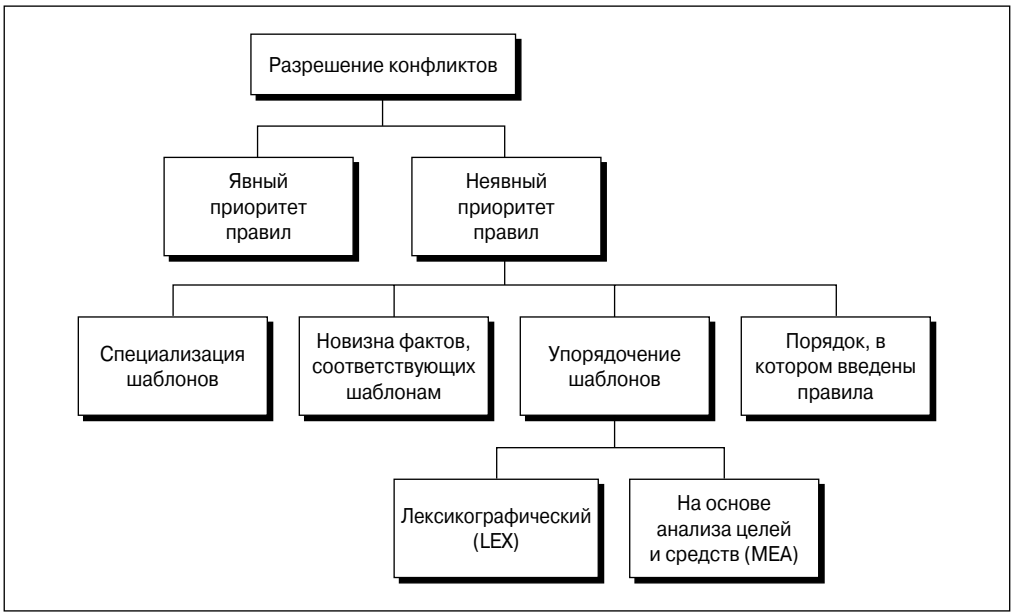


Рис. 5.4. Неопределенность, связанная с разрешением конфликтов

необходимо, поскольку при увеличении объема доступной информации возрастает доверие к более конкретному правилу, когда существует потенциальная возможность активизации для последующего запуска многочисленных правил.

Например, если экспертная система применяется для обеспечения комфорта и ее пользователь говорит: “Я чувствую жар”, то одним из правил может быть: “IF пользователь чувствует жар THEN он должен принять аспирин”. С другой стороны, рассматривается как более конкретное правило, в котором указано: “IF пользователь чувствует жар AND сидит в пальто летом на пляже THEN он должен снять пальто”, если доступна дополнительная информация, что данное лицо действительно сидит в пальто на горячем песке. Вообще говоря, мерой доверия к тому, что некоторое правило является более подходящим, может служить количество успешных согласований с шаблонами в левой части правила, т.е. в большей степени подходит правило с бóльшим, а не меньшим количеством согласующихся шаблонов (например, меньшим из-за того, что доступен меньший объем информации). **Конкретность** правила CLIPS зависит от количества шаблонов и от внутренней сложности каждого шаблона. Например, следующий шаблон:

(ball ellipsoidal)

является более конкретным, чем такой шаблон:

(ball)

В первом шаблоне речь идет об эллипсоидальном мяче, который, по-видимому, представляет собой мяч для американского футбола, а во втором шаблоне говорится просто о мяче, который может представлять собой теннисный мяч, мяч для крикета, баскетбольный, бейсбольный или мяч любого другого типа.

Возвращаясь к двум правилам, которые рассматривались выше, отметим, что правило (4) является более конкретным, поскольку имеет в антецеденте два шаблона, и поэтому с ним неявно связан более высокий приоритет. В системе CLIPS в первую очередь будет выполнено правило (4).

Но обнаруживаются и другие осложнения. Дело в том, что приходится учитывать не только конкретность, но и **новизну фактов**. После ввода факта в рабочую память ему присваивается уникальная **временна́я отметка**, позволяющая узнать, когда он был введен. Таким образом, правило со следующим шаблоном получило бы более высокий приоритет, чем правило (3), если бы факт E_3 был введен после факта E_1:

(5) IF E_3 THEN H

Свое влияние оказывает также порядок расположения шаблонов в правиле. В системе OPS5 разрешено использование двух других **стратегий управления**, называемых **лексикографической** (**lexicographic — LEX**) стратегией и **анализом целей и средств** (**means-ends analysis — MEA**). Эти стратегии определяют способ интерпретации шаблонов правил машиной логического вывода. Стратегия управления выбирается путем ввода команды `strategy` в интерпретатор экспертной системы. При использовании стратегии LEX порядок шаблонов не имеет никакого значения (если не считать того, что, возможно, влияет на эффективность). Таким образом, следующие два правила, согласно стратегии LEX, по сути рассматриваются как одинаковые:

IF E_1 AND E_2 THEN H
IF E_2 AND E_1 THEN H

С другой стороны, стратегия MEA используется в программе универсального решателя задач Ньюэлла и Саймона, которая рассматривалась в главе 1. Основная идея стратегии MEA состоит в том, что нужно стремиться уменьшить различие между начальным состоянием и состоянием успеха. В той реализации MEA, которая применяется в системе OPS5, очень важен выбор первого шаблона, поскольку именно он применяется для управления процессом согласования с шаблонами. Назначение стратегии MEA состоит в том, что она позволяет системе продолжать выполнение текущего задания, не подвергаясь влиянию того, что в рабочую память введены еще более свежие факты. В стратегии MEA приоритет правил определяется с учетом первого шаблона, который непосредственно следует за конструкцией IF. Если одно из правил приобретает более высокий приоритет по

сравнению с другими с учетом именно первого шаблона, то выбирается как раз такое правило. Для определения наиболее высокоприоритетного правила может применяться либо критерий конкретности, либо критерий новизны. Если же количество наиболее высокоприоритетных правил, классифицируемых по первому шаблону, становится больше единицы, то, согласно стратегии MEA, осуществляется переупорядочение правил с учетом остальных шаблонов. В конечном итоге, если не обнаруживается единственное наиболее высокоприоритетное правило, осуществляется произвольный выбор правила.

Важным фактором в разрешении конфликтов может также стать правильный порядок ввода правил в экспертную систему. Если машина логического вывода не имеет возможности выбирать правила с учетом приоритетов, то должен происходить произвольный выбор правила. Но проектировщики машины логического вывода должны соблюдать исключительную осторожность, поскольку в противном случае может оказаться, что машина осуществляет свой выбор в определенном смысле детерминированно, так как выбор происходит в зависимости от последовательности ввода правил. Такая ситуация может возникать из-за того, что проектировщики машины логического вывода используют стек или очередь для хранения правил в рабочем списке правил. Но для того чтобы выбор правил с равными приоритетами происходил произвольно, правила должны извлекаться из стека или очереди действительно случайно. Но вместо этого проектировщики могут пойти по более легкому пути и просто выталкивать очередное правило из стека или извлекать следующее правило из очереди. Такой метод выбора прост, но его применение приводит к внесению в систему известного *артефакта* (постороннего влияния), поскольку выбор одного из правил среди правил с равным приоритетом не является произвольным.

Обобщение и неопределенность

Проблема обобщения становится более трудноразрешимой, если с правилами связаны правдоподобия, как показано ниже.

(6) IF E_1 THEN H with LS_1

(7) IF E_1 AND E_2 THEN H with LS_2

Например, рассмотрим следующие правила:

```
IF the starter motor doesn't work
THEN check the battery with LS1 = 5
IF the starter motor doesn't work AND
   the lights don't work
THEN check the battery with LS2 = 10
```

В этих правилах значение LS_2 больше чем LS_1, поскольку заключение обосновывается большим количеством свидетельств.

Как было указано в главе 4, если наблюдаются оба свидетельства, E_1 и E_2, то, согласно следующей формуле апостериорных шансов, произведение становится равным $5 \cdot 10 = 50$:

$$O(H \mid E) = \left[\prod_{i=1}^{N} LS_i\right] O(H) \tag{8}$$

Но в действительности формула (8) предназначена для описания правил, свидетельства которых комбинируются независимо друг от друга. Тем не менее антецеденты правил (6) и (7) на самом деле не являются независимыми, поскольку в них имеется общее свидетельство E_1. Таким образом, выражение E_1 AND E_2 обобщает выражение E_1, поскольку выражение E_1, отдельно взятое, является частным случаем выражения E_1 AND E_2. Иначе говоря, возникает ошибка, которая заключается в том, что произведение отношений LS_1 и LS_2 слишком высоко, поскольку антецеденты правил не являются независимыми. Решение данной проблемы состоит в замене LS_2 отношением LS_2/LS_1, для того чтобы при наличии обоих свидетельств, E_1 и E_2, вычислялось правильное произведение LS_2. Но задача поиска вручную возможных обобщений в крупной базе знаний может оказаться слишком сложной.

Третьей причиной неопределенности становится наличие **избыточных правил**, которые имеют одинаковые консеквенты и свидетельства. Обычно появление таких правил вызвано тем, что их непреднамеренно вводит повторно инженер по знаниям; в некоторых случаях такие дубликаты случайно возникают после модификации правил в результате удаления шаблонов. Например, рассмотрим следующие правила:

IF E_1 AND E_2 THEN H
IF E_2 AND E_1 AND E_3 THEN H

Эти правила становятся избыточными после удаления свидетельства E_3, поскольку антецеденты после этого соответствуют одному и тому же шаблону. Но задача выбора для удаления одного из избыточных правил не всегда решается просто. Если в одном из избыточных правил более редко встречающийся шаблон стоит на первом месте, то эффективность системы увеличивается благодаря тому, что в результате неудачного завершения попытки согласования первого шаблона машиной логического вывода отпадает необходимость проверять второй шаблон. Дополнительные усложнения возникают при использовании таких командных интерпретаторов экспертных систем, как OPS5, поскольку в них порядок активизации зависит от стратегии управления, выбранной пользователем. Как и в случае обобщения, наличие избыточных правил также приводит к возникновению ошибки, связанной со слишком высоким значением произведения коэффициентов правдоподобия.

Четвертой причиной появления неопределенности в консеквентах правил является наличие **недостающих правил**. Такая ситуация возникает из-за того, что эксперт-человек забывает о существовании необходимого правила или не знает о его существовании, как в следующем примере:

$$\text{IF } E_4 \text{ THEN } H$$

В данном случае игнорируется свидетельство E_4, а это означает, что гипотеза H не подтверждается с той основательностью, которая ей соответствует. Одним из преимуществ сетей логического вывода, подобных PROSPECTOR, является явно выраженный характер сети, что способствует упрощению поиска таких гипотез, которых невозможно или сложно достичь. Если есть недостающие правила, то необходимость во введении этих правил может быть обнаружена непосредственно из анализа самой сети логического вывода.

Пятая причина неопределенности обусловлена возникновением проблем, связанных со **сплавлением данных**. Этим термином обозначается неопределенность, возникающая, когда данные берутся из источников, представляющих информацию разных типов. Объединение таких данных и называется *сплавлением данных*. Например, врач, составляя диагноз, может рассматривать свидетельства, полученные из таких весьма разнообразных источников, как физическое обследование, результаты медицинских анализов, история болезни пациента, описание социально-экономической среды, результаты анализа ментального и эмоционального состояний, наличие проблем в семье и на работе и т.д. Все эти свидетельства несут в себе информацию разных типов, которая должна быть подвергнута сплавлению в целях обоснования окончательной гипотезы. Задача сплавления свидетельств, полученных из таких многочисленных и разных источников, является гораздо более сложной по сравнению с задачей использования свидетельств, относящихся к одной предметной области, такой как геология.

Аналогичным образом, деловые решения могут зависеть от наличия рынка для товаров, экономических условий, состояния внешней торговли, проявления попыток поглощения, политики компании, личных проблем, участия в экономических союзах и от многих других факторов. Как и в случае решения задач постановки медицинского диагноза, очень сложно присвоить всем этим факторам коэффициенты правдоподобия и определить приемлемую комбинирующую функцию.

5.3 Коэффициенты достоверности

Еще один метод учета неопределенности состоит в использовании **коэффициентов достоверности**, которые были впервые разработаны для экспертной системы MYCIN.

Трудности, связанные с применением байесовского метода

Неопределенность возникает не только при решении задач поиска полезных ископаемых, для чего применяется система PROSPECTOR, но и при решении почти любых задач медицинской диагностики. Основное различие между задачами двух указанных типов состоит в том, что в первом случае гипотезы формулируются на основе знаний в области геологии и являются не столь многочисленными, поскольку ограниченно само количество различных минералов, в формировании которых участвуют лишь 92 природных элемента. С другой стороны, количество болезнетворных микроорганизмов намного больше, поэтому значительно увеличивается и количество возможных гипотез, касающихся заболеваний.

Безусловно, теорема Байеса находит в медицине широкое применение, но получение с ее помощью точных результатов возможно, только если известно значительное количество вероятностей. Например, теорема Байеса может служить для определения вероятности конкретного заболевания, если известны некоторые симптомы, определяемые следующей формулой:

$$P(D_i \mid E) = \frac{P(E \mid D_i)P(D_i)}{P(E)} = \frac{P(E \mid D_i)P(D_i)}{\sum\limits_j P(E \mid D_j)P(D_j)}$$

В этой формуле сумма по j распространяется на все заболевания и применяются такие обозначения:

- D_i — i-е заболевание;
- E — свидетельство;
- $P(D_i)$ — априорная вероятность того, что пациент имеет заболевание D_i, установленная до получения каких-либо свидетельств;
- $P(E \mid D_i)$ — условная вероятность того, что в отношении пациента будет обнаружено свидетельство E, при условии наличия у него заболевания D_i.

Но при наличии общей совокупности обычно невозможно определить согласованные и полные значения для всех этих вероятностей.

На практике обычно происходит последовательное накопление свидетельства, часть за частью. Такое накопление требует существенных затрат времени и денег, особенно если для этого применяются медицинские анализы. Как правило, количество выполняемых медицинских анализов сводится к минимуму, необходимому для постановки качественного диагноза (и защиты от врачебной ошибки), с учетом таких факторов, как время, стоимость и потенциальный риск для пациента.

Поэтому вместо приведенной выше формулы применяется более удобная форма теоремы Байеса, которая хорошо представляет подобный процесс инкремент-

ного накопления свидетельств (см. задачу 5.1):

$$P(D_i \mid E) = \frac{P(E_2 \mid D_i \cap E_1)P(D_i \mid E_1)}{\sum\limits_j P(E_2 \mid D_j \cap E_1)P(D_j \mid E_1)}$$

В этой формуле E_2 обозначает новое свидетельство, добавляемое к существующей совокупности свидетельств E_1 для получения еще более нового, дополненного свидетельства:

$$E = E_1 \cap E_2$$

Безусловно, эта формула является точной, но обычно неизвестны все необходимые для нее вероятности. Кроме того, ситуация становится еще более затруднительной по мере накопления все большего и большего количества частей свидетельств, поскольку в связи с этим требуется все больше вероятностей.

Степени доверия и недоверия

Перед экспертами в области медицины возникает не только проблема, связанная с увеличением количества условных вероятностей, необходимых для использования байесовского метода, но и другая важная проблема, которая касается отношения между доверием и недоверием к определенной гипотезе. На первый взгляд эта проблема может показаться тривиальной, поскольку очевидно, что недоверие просто противоположно доверию. И действительно, в теории вероятностей сформулировано следующее утверждение:

$$P(H) + P(H') = 1$$

и поэтому:

$$P(H) = 1 - P(H')$$

Для апостериорной гипотезы, обоснованием для которой служит свидетельство E, справедлива следующая формула:

$$P(H \mid E) = 1 - P(H' \mid E) \tag{1}$$

Но после того как инженеры по знаниям, участвующие в разработке системы MYCIN, приступили к собеседованиям с экспертами в области медицины, было обнаружено, что врачи выражают крайнее нежелание формулировать свои знания, пользуясь уравнением (1).

Например, рассмотрим такое правило MYCIN:

```
IF 1) The stain of the organism is gram positive,
   and
   2) The morphology of the organism is coccus,
```

```
   and
   3) The growth conformation of the organism is chains
THEN There is suggestive evidence (0.7) that the
   identity of the organism is streptococcus
```

Попросту говоря, это правило гласит о том, что если бактериальный организм приобретает окраску после окрашивания по Граму и напоминает цепочки сфер, то имеется 70%-ное правдоподобие, что он относится к типу микроорганизмов, называемых стрептококками. Это утверждение может быть записано в терминах апостериорной вероятности следующим образом:

$$P(H \mid E_1 \cap E_2 \cap E_3) = 0.7 \tag{2}$$

В этой формуле выражения E_i соответствуют трем шаблонам антецедента.

Инженеры по знаниям системы MYCIN обнаружили следующее: эксперты соглашаются с тем, что уравнение (2) справедливо, но чувствуют себя растерянными и отказываются соглашаться с таким вероятностным результатом:

$$P(H' \mid E_1 \cap E_2 \cap E_3) = 1 - 0.7 = 0.3 \tag{3}$$

Это нежелание экспертов соглашаться с тем, что уравнение (3) истинно, еще раз показывает, что такие числа, как 0,7 и 0,3, представляют собой правдоподобия степени доверия, а не вероятности.

Для того чтобы полностью оценить значимость этой проблемы несогласованности между доверием и недоверием, рассмотрим следующий пример. Предположим, что вам для получения документа об образовании требуется пройти последний курс. Предположим, что ваш *средний аттестационный балл* (Grade Point Average — GPA) не слишком высок и после прохождения данного курса вам необходимо получить отличную оценку, чтобы поднять свой средний аттестационный балл. Вашу степень доверия к правдоподобию получения документа об окончании учебного заведения можно выразить с помощью следующей формулы:

$$P(\text{graduating} \mid \text{'A' in this course}) = 0.70 \tag{4}$$

Обратите внимание на то, что это правдоподобие не равно 100%. Причина, по которой оно не составляет 100%, заключается в том, что в данный момент происходит окончательное подведение итогов прохождения вами всех курсов, после чего учебным заведением будут выставлены баллы. Успешному получению документа об образовании могут также препятствовать другие проблемы, обусловленные целым рядом причин, в том числе перечисленных ниже.

1. Учебная программа изменилась так, что не все пройденные вами курсы рассматриваются как основание для получения документа об образовании.
2. Вы забыли пройти обязательный курс.

3. Вы получили отказ признать действительными курсы, пройденные в другом учебном заведении.
4. Вы получили отказ признать действительными некоторые пройденные выборочные курсы.
5. Вы должны были внести плату за обучение и пользование библиотекой, но не сделали этого, надеясь, что о ваших долгах забудут.
6. Ваш средний аттестационный балл ниже, чем вы думали, поэтому его не удастся поднять даже с помощью отличной оценки.
7. “Они” твердо решили вам напакостить.

Предположим, что вы признали наличие проблемы (4) (или, возможно, сами переоценили значение правдоподобия), поэтому, согласно уравнению (1):

$$P(\text{not graduating} \mid \text{'A' in this course}) = 0.30 \tag{5}$$

Безусловно, с вероятностной точки зрения уравнение (5) является правильным, но интуитивно кажется, что в нем есть какая-то ошибка. В частности, просто не верится, что даже после упорной работы и получения отличной оценки по данному курсу остаются 30%-ные шансы на то, что вы не получите документ об образовании. Уравнение (5) создает какое-то ощущение неуверенности, как и у эксперта в области медицины, который признает справедливость формулы

$$P(H \mid E_1 \cap E_2 \cap E_3) = 0.70$$

но не уверен в том, что справедливо ее вероятностное следствие:

$$P(H' \mid E_1 \cap E_2 \cap E_3) = 0.30$$

Фундаментальная проблема состоит в том, что значение $P(H \mid E)$ подразумевает наличие причинно-следственного отношения между E и H, тогда как между E и H' может не существовать причинно-следственное отношение. Тем не менее в следующем уравнении подразумевается наличие причинно-следственного отношения между E и H', если есть причинно-следственное отношение между E и H:

$$P(H \mid E) = 1 - P(H' \mid E)$$

Эти проблемы, связанные с теорией вероятностей, подвигли Шортлиффа (Shortliffe) на исследование других способов представления неопределенности. Он применил в системе MYCIN метод, основанный на использовании **коэффициентов достоверности**, разработанных на базе теории подтверждения Карнапа (Carnap). Карнап различает два типа вероятностей.

Одним из типов вероятности является **обычная вероятность**. Это понятие ассоциируется с частотой воспроизводимых событий. Вероятность второго типа

называется **эпистемистической вероятностью**, или **степенью подтверждения**, поскольку она подтверждает гипотезу на основании некоторого свидетельства. Этот второй тип является еще одним примером применения степени правдоподобия того, что некоторая гипотеза заслуживает доверия.

Меры, применяемые для измерения степени доверия и недоверия

В системе MYCIN степень подтверждения была первоначально определена как коэффициент достоверности, который определяется как разница между *степенью доверия* (belief) и *степенью недоверия* (disbelief):

$$CF(H,E) = MB(H,E) - MD(H,E)$$

В этой формуле применяются следующие обозначения:

- CF — коэффициент достоверности гипотезы H, обусловленный наличием свидетельства E;
- MB — мера повышения степени доверия к гипотезе H в силу наличия свидетельства E;
- MD — мера повышения степени недоверия к гипотезе H в силу наличия свидетельства E.

Коэффициент достоверности — это способ объединения двух значений, степени доверия и степени недоверия, в единственное число.

Объединение мер доверия и недоверия в единственное число осуществляется в двух целях. Прежде всего, коэффициент достоверности может использоваться для ранжирования гипотез в порядке их важности. Например, если у пациента есть некоторые симптомы, свидетельствующие о наличии нескольких возможных заболеваний, то необходимо обеспечить, чтобы на основании медицинских анализов в первую очередь было проведено обследование для диагностирования именно того заболевания, которому соответствует наибольшее значение CF (Certainty Factor — коэффициент достоверности).

Меры доверия и недоверия были определены в терминах вероятностей по следующим формулам:

$$MB(H,E) = \begin{cases} 1 & \text{если } P(H) = 1 \\ \dfrac{\max[P(H \mid E), P(H)] - P(H)}{\max[1,0] - P(H)} & \text{в противном случае} \end{cases}$$

$$MD(H,E) = \begin{cases} 1 & \text{если } P(H) = 0 \\ \dfrac{\min[P(H \mid E), P(H)] - P(H)}{\min[1,0] - P(H)} & \text{в противном случае} \end{cases}$$

Теперь отметим, что значение $\max[1, 0]$ всегда равно 1, а значение $\min[1, 0]$ всегда равно 0. Но в данных формулах значения 1 и 0 записаны в терминах $\max$ и $\min$, поскольку это позволяет показать формальную симметрию между выражениями для MB (measure of belief — мера доверия) и MD (measure of disbelief — мера недоверия). Уравнения, с помощью которых вычисляются значения MB и MD, отличаются только тем, что в первом случае применяется функция $\max$, а во втором — $\min$.

В табл. 5.1 показаны некоторые характерные случаи применения значений MB, MD и CF, которые определены на основании приведенных выше формул.

Таблица 5.1. Некоторые характерные случаи применения значений MB, MD и CF

Характеристики	**Значения**
Интервалы значений	$0 \leqslant MB \leqslant 1$ $0 \leqslant MD \leqslant 1$ $-1 \leqslant CF \leqslant 1$
Некоторая истинная гипотеза $P(H \mid E) = 1$	$MB = 1$ $MD = 0$ $CF = 1$
Некоторая ложная гипотеза $P(H' \mid E) = 1$	$MB = 0$ $MD = 1$ $CF = -1$
Отсутствие свидетельства $P(H \mid E) = P(H)$	$MB = 0$ $MD = 0$ $CF = 0$

Коэффициент достоверности CF показывает, какова чистая степень доверия к гипотезе, основанная на некотором свидетельстве. Положительное значение CF говорит о том, что свидетельство обосновывает гипотезу, поскольку $MB > MD$. Тот случай, в котором $CF = 1$, означает, что свидетельство определенно доказывает гипотезу. С другой стороны, случай $CF = 0$ соответствует одной из двух возможностей. Во-первых, $CF = MB - MD = 0$ может означать, что нулю равны и MB, и MD, иными словами, что отсутствует какое-либо свидетельство. Во-вторых, возможно, что $MB = MD$ и оба эти значения отличны от нуля, а это сводится к тому, что доверие к гипотезе опровергается такой же степенью недоверия. К сожалению, опровержение сильной степени доверия таким же недоверием ведет не просто к незнанию, но к некоторому состоянию путаницы (а это гораздо хуже). Например, что может быть неприятнее для водителя, который подъехал к перекрестку и не знает, куда повернуть, или услышал от пассажира, указывающего влево, что нужно повернуть направо!

Отрицательные значения CF показывают, что свидетельство способствует опровержению гипотезы, поскольку $MB < MD$. Иными словами эту мысль можно выразить так, что есть больше оснований не доверять гипотезе, чем доверять ей. Например, величина $CF = -70\%$ означает, что степень недоверия на 70% выше, чем степень доверия. С другой стороны, значение $CF = 70\%$ показывает, что степень доверия на 70% выше, чем степень недоверия. Обратите внимание на то, что при использовании коэффициентов достоверности никакие ограничения на значения величин MB и MD, отдельно взятых, не налагаются. Важной остается только разница между этими величинами. Например, могут наблюдаться показанные ниже и любые другие значения.

$$CF = 0.70 = 0.70 - 0 = \\ = 0.80 - 0.10$$

Коэффициенты достоверности позволяют эксперту выразить степень доверия, не задумываясь над тем, какое значение может иметь степень недоверия. В частности, в задаче 5.2 показано, что справедлива следующая формула:

$$CF(H, E) + CF(H', E) = 0$$

Эта формула означает, что если свидетельство подтверждает гипотезу на некоторое значение $CF(H \mid E)$, то подтверждение отрицания гипотезы не равно $1 - CF(H \mid E)$, как следовало бы ожидать согласно теории вероятностей. Таким образом, верно следующее:

$$CF(H, E) + CF(H', E) \neq 1$$

Тот факт, что $CF(H \mid E) + CF(H' \mid E) = 0$, означает, что свидетельство, поддерживающее гипотезу, снижает поддержку отрицания гипотезы на равную величину, так что сумма всегда остается нулевой.

В примере со студентом, который рассчитывает на получение документа об окончании учебного заведения, если получит отличную оценку за данный курс, имеет место следующее:

$$CF(H, E) = 0.70$$
$$CF(H', E) = -0.70$$

Эти формулы означают:

(6) I am 70 percent certain that I will graduate if I get an 'A' in this course.

(7) I am -70 percent certain that I will not graduate if I get an 'A' in this course.

Обратите внимание на то, что значение -70% получено в связи с тем, что коэффициенты достоверности определяются в следующем интервале, где 0 означает отсутствие свидетельства:

$$-1 \leqslant CF(H, E) \leqslant +1$$

Таким образом, положительные коэффициенты достоверности свидетельствуют в пользу гипотезы, а отрицательные коэффициенты достоверности — в пользу отрицания гипотезы. Утверждения (6) и (7) становятся эквивалентными, если коэффициенты достоверности используются по аналогии с тем фактом, что "да = не нет".

Приведенные выше значения CF можно было бы выявить у учащегося, задав ему следующий вопрос, если свидетельство должно подтвердить гипотезу:

```
How much do you believe that getting an 'A' will help you
graduate?
```

А если нужно опровергнуть гипотезу, то вопрос формулируется таким образом:

```
How much do you disbelieve that getting an 'A' will help
you graduate?
```

Если учащийся на любой из этих вопросов ответит "70%", то указанным величинам будут присвоены значения $CF(H \mid E) = 0.70$ и $CF(H' \mid E) = -0.70$. А в системе MYCIN эксперты не должны были указывать достоверности в процентах, поскольку им предлагали выразить свое представление о достоверности гипотезы с помощью шкалы от 1 до 10, где значение 10 показывало полную уверенность в справедливости гипотезы. Кроме того, пользователи могли давать ответ `UNK` (сокращение от unknown), если свидетельство не известно; это соответствует значению $CF = 0$.

Вычисления, проводимые с использованием коэффициентов достоверности

Как уже было сказано, первоначально применялось приведенное выше определение CF, но при использовании этого определения возникали сложности, поскольку даже единственная часть опровергающего свидетельства могла влиять на подтверждение со стороны многих других частей свидетельства.

$$CF = MB - MD$$

Например, при использовании 10 частей свидетельства могло было быть получено значение $MB = 0.999$, после чего единственная опровергающая часть с $MD = 0.799$ могла привести к получению такого результата:

$$CF = 0.999 - 0.799 = 0.200$$

В системе MYCIN значение CF антецедента правила должно быть больше 0.2, для того чтобы антецедент рассматривался как истинный и активизировал правило. Такое значение 0.2 рассматривается как **пороговое значение**, но не определено как фундаментальная аксиома теории коэффициентов достоверности. Вместо этого пороговое значение рассматривается как произвольный способ сведения к минимуму возможности активизации правил, которые лишь в незначительной степени подтверждают гипотезу. Без использования порогового значения могут активизироваться многочисленные правила, которые являются малозначительными или вообще не имеют значения, поэтому эффективность системы существенно уменьшается.

В 1977 году приведенное выше определение CF в системе MYCIN было изменено и приняло такой вид:

$$CF = \frac{MB - MD}{1 - \min(MB, MD)}$$

Это было сделано в целях ослабления влияния единственных частей опровергающих свидетельств на многочисленные подтверждающие части свидетельств. Если при использовании указанного определения применяются такие же значения $MB = 0.999$, $MD = 0.799$, то значение CF принимает вид:

$$CF = \frac{0.999 - 0.799}{1 - \min(0.999, 0.799)} = \frac{0.200}{1 - 0.799} = 0.995$$

Это значение весьма существенно отличается от значения, полученного согласно предыдущему определению, при котором результат был равен $0.999 - 0.799 = = 0.200$, и поэтому не происходила активизация правила, поскольку значение не было больше порогового значения 0.2. С другой стороны, при использовании современного определения полученное значение 0.995 вызывает активизацию правила.

В табл. 5.2 показаны правила, применяемые в системе MYCIN для комбинирования свидетельств в антецедентах правил. Обратите внимание на то, что эти правила совпадают с правилами системы PROSPECTOR, основанными на нечеткой логике.

Таблица 5.2. Правила MYCIN, применяемые для комбинирования свидетельств, заданных элементарными выражениями, в антецедентах правил

Свидетельство *E*	**Достоверность антецедента**
E_1 AND E_2	$\min[CF(H, E_1), CF(H, E_2)]$
E_1OR E_2	$\max[CF(H, E_1), CF(H, E_2)]$
NOT E	$-CF(H, E)$

Например, если дано следующее логическое выражение, применяемое для комбинирования свидетельств:

$$E = (E_1 \text{ AND } E_2 \text{ AND } E_3) \text{ OR } (E_4 \text{ AND NOT } E_5)$$

то значение свидетельства E можно вычислить таким образом:

$$E = \max[\min(E_1, E_2, E_3), \min(E_4, -E_5)]$$

При использовании значений

$$E_1 = 0.9 \qquad E_2 = 0.8 \qquad E_3 = 0.3$$
$$E_4 = -0.5 \qquad E_5 = -0.4$$

получаем следующий результат:

$$\begin{aligned} E &= \max[\min(0.9, 0.8, 0.3), \min(-0.5, -(-0.4))] = \\ &= \max[0.3, -0.5] \\ &= 0.3 \end{aligned}$$

Фундаментальная формула определения коэффициента достоверности CF для правила

$$\text{IF } E \text{ THEN } H$$

задается следующей формулой:

$$CF(H, e) = CF(E, e)CF(H, E) \tag{8}$$

В этой формуле применяются такие обозначения:

- $CF(E, e)$ — коэффициент достоверности свидетельства E, формирующего антецедент правила на основе неопределенного свидетельства e;
- $CF(H, E)$ — коэффициент достоверности гипотезы, в которой предполагается, что свидетельство известно со всей достоверностью, когда $CF(E, e) = 1$;
- $CF(H, e)$ — коэффициент достоверности гипотезы, основанной на неопределенном свидетельстве e.

Таким образом, если все свидетельства в антецеденте известны со всей достоверностью, то формула для коэффициента достоверности гипотезы принимает следующий вид, поскольку $CF(E, e) = 1$:

$$CF(H, e) = CF(H, E)$$

В качестве примера применения таких коэффициентов достоверности рассмотрим значение CF для правила определения наличия стрептококковой инфекции, описанного выше:

```
IF 1) The stain of the organism is gram positive,
   and
2) The morphology of the organism is coccus,
   and
3) The growth conformation of the organism is chains
THEN There is suggestive evidence (0.7) that the
identity of the organism is streptococcus
```

В этом правиле коэффициент достоверности гипотезы при наличии достоверного свидетельства определяется следующей формулой и именуется также **коэффициентом ослабления** (attenuation factor):

$$CF(H, E) = CF(H, E_1 \cap E_2 \cap E_3) = 0.7$$

Это определение коэффициента ослабления основано на предположении, что все свидетельства (E_1, E_2 и E_3) известны с полной достоверностью. Таким образом, справедлива следующая формула, в которой e представляет собой наблюдаемое свидетельство, ведущее к заключению, что каждое из свидетельств E_i известно с полной достоверностью:

$$CF(E_1, e) = CF(E_2, e) = CF(E_3, e) = 1$$

Эти значения CF аналогичны условным вероятностям свидетельств в системе PROSPECTOR, $P(E \mid e)$. Коэффициент ослабления выражает степень достоверности, относящуюся к гипотезе, если даны некоторые достоверные свидетельства.

Так же как и в системе PROSPECTOR, если не известны все свидетельства с полной достоверностью, возникают сложности. Поэтому в системе PROSPECTOR при наличии недостоверных свидетельств использовалась интерполяционная формула $P(H \mid e)$. С другой стороны, в системе MYCIN для определения результирующего значения CF должна применяться фундаментальная формула (8), поскольку формула $CF(H, E_1 \cap E_2 \cap E_3) = 0.7$ для недостоверных свидетельств больше не действительна.

Например, примем следующие предположения:

$$CF(E1, e) = 0.5$$
$$CF(E2, e) = 0.6$$
$$CF(E3, e) = 0.3$$

В этом случае получаем такой результат:

$$\begin{aligned} CF(E, e) &= CF(E_1 \cap E_2 \cap E_3, e) = \\ &= \min[CF(E_1, e), CF(E_2, e), CF(E_3, e)] = \\ &= \min[0.5, 0.6, 0.3] = \\ &= 0.3 \end{aligned}$$

Таким образом, коэффициент достоверности антецедента, $CF(E,e) > 0.2$, поэтому антецедент рассматривается как истинный и правило активизируется. Коэффициент достоверности заключения равен следующему:

$$\begin{aligned} CF(H,e) &= CF(E,e)CF(H,E) = \\ &= 0.3 \cdot 0.7 = \\ &= 0.21 \end{aligned}$$

Предположим, что та же гипотеза следует также из другого правила, но с другим коэффициентом достоверности. Коэффициенты достоверности правил, из которых следует одна и та же гипотеза, вычисляются с помощью **комбинирующей функции** для коэффициентов достоверности, которая определена таким образом:

$$CF_{\text{COMBINE}}(CF_1, CF_2) = \begin{cases} CF_1 + CF_2(1 - CF_1) & \text{оба} > 0 \\ \dfrac{CF_1 + CF_2}{1 - \min(|CF_1|, |CF_2|)} & \text{один} < 0 \\ CF_1 + CF_2(1 + CF_1) & \text{оба} < 0 \end{cases} \tag{9}$$

В этом выражении выбор формулы для CF_{COMBINE} зависит от того, являются ли отдельные коэффициенты достоверности положительными или отрицательными. Применение комбинирующей функции к коэффициентам достоверности, количество которых превышает два, осуществляется инкрементно. Таким образом, величина CF_{COMBINE} вычисляется для двух значений CF, а затем величина CF_{COMBINE} комбинируется на основании формулы (9) с третьим значением CF и т.д. Итоговые результаты вычислений с использованием коэффициентов достоверности для двух правил на основе недостоверного свидетельства и вывода одной и той же гипотезы приведены на рис. 5.5. Обратите внимание на то, что дерево, показанное на этом рисунке, не является деревом AND/OR, поскольку величина CF_{COMBINE} не имеет ничего общего с целями AND и OR.

Если по условиям рассматриваемого примера следует заключение о наличии стрептококков с коэффициентом достоверности $CF_2 = 0.5$ еще из одного правила, то комбинированный коэффициент достоверности, вычисленный с использованием первой формулы (9), выражается следующим образом:

$$CF_{\text{COMBINE}}(0.21, 0.5) = 0.21 + 0.5(1 - 0.21) = 0.605$$

Предположим, что такой же вывод следует из третьего правила, но на этот раз $CF_3 = -0.4$. В таком случае используется вторая формула (9), что приводит к получению следующего значения:

$$\begin{aligned} CF_{\text{COMBINE}}(0.605, -0.4) &= \frac{0.605 - 0.4}{1 - \min(|0.605|, |-0.4|)} = \\ &= \frac{0.205}{1 - 0.4} = 0.34 \end{aligned}$$

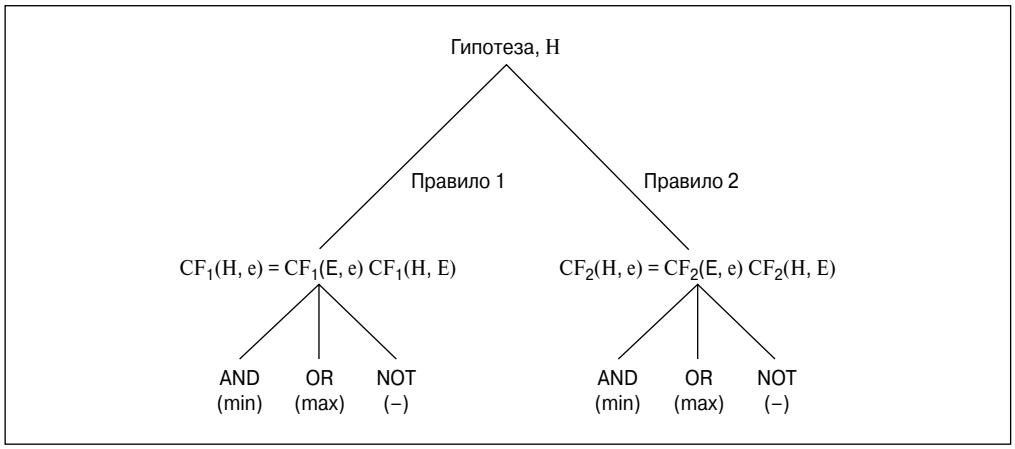


Рис. 5.5. Коэффициенты достоверности двух правил с одной и той же гипотезой, основанных на неопределенном свидетельстве

Формула CF_{COMBINE} сохраняет свойство коммутативности свидетельств. Это означает, что справедлива приведенная ниже формула, поэтому порядок получения свидетельств не влияет на конечный результат.

$$CF_{\text{COMBINE}}(X, Y) = CF_{\text{COMBINE}}(Y, X)$$

В системе MYCIN хранятся не отдельные значения MB и MD для каждой гипотезы, а вместе с каждой гипотезой хранится текущее значение CF_{COMBINE}, которое затем комбинируется с новым свидетельством, как только последнее становится доступным:

$$CF_{\text{COMBINE}}(CF_1, CF_2) = \begin{cases} CF_1 + CF_2(1 - CF_1) & \text{оба} > 0 \\ \dfrac{CF_1 + CF_2}{1 - \min(|CF_1|, |CF_2|)} & \text{один} < 0 \\ CF_1 + CF_2(1 + CF_1) & \text{оба} < 0 \end{cases}$$

Сложности, связанные с использованием коэффициентов достоверности

Безусловно, система MYCIN оказалась очень успешным средством медицинской диагностики, но при разработке теоретического фундамента, лежащего в основе применения коэффициентов достоверности, возникли сложности. Коэффициенты достоверности базируются на некоторых понятиях теории вероятностей и теории подтверждения. Однако сами эти коэффициенты достоверности отчасти применялись на основании произвольно выбранного подхода. Основное преимущество коэффициентов достоверности состояло в том, что вычисления, с помо-

щью которых можно было распространять в системе информацию о неопределенности, были несложными. Кроме того, назначение коэффициентов достоверности можно было легко понять и четко отделить степень доверия от степени недоверия.

Тем не менее при использовании коэффициентов достоверности возникали проблемы. Одна из проблем состояла в том, что значения коэффициентов достоверности могли быть противоположными условным вероятностям. Например, если даны такие значения:

$$P(H_1) = 0.8 \qquad P(H_2) = 0.2$$
$$P(H_1 \mid E) = 0.9 \qquad P(H_2 \mid E) = 0.8$$

то получаем следующее:

$$CF(H_1, E) = 0.5 \text{ и } CF(H_2, E) = 0.75$$

К тому же одним из назначений коэффициентов достоверности является ранжирование гипотез в терминах вероятных диагнозов, поэтому возникает противоречие, если некоторое заболевание отличается более высокой условной вероятностью $P(H \mid E)$, но все равно характеризуется более низким коэффициентом достоверности, $CF(H, E)$.

Еще одна важная проблема, связанная с использованием коэффициентов достоверности, состоит в том, что, вообще говоря, справедлива следующая формула:

$$P(H \mid e) \neq P(H \mid i)P(i \mid e)$$

где i — некоторая промежуточная гипотеза, основанная на свидетельстве e. Но, несмотря на это, коэффициент достоверности двух правил в цепи логического вывода вычисляется с использованием независимых вероятностей по такой формуле:

$$CF(H, e) = CF(H, i)CF(i, e)$$

Приведенная выше формула справедлива только в том частном случае, когда статистическая совокупность со свойством H содержится в совокупности i, которая содержится в совокупности со свойством e. То, что система MYCIN оказалась успешной, несмотря на наличие этих проблем, возможно, обусловлено применением коротких цепей логического вывода и простых гипотез. Но при попытке использовать коэффициенты достоверности в других прикладных областях, не позволяющих применять короткие цепи логического вывода и простые гипотезы, могут возникнуть реальные проблемы. И действительно, Адамс (Adams) показал, что теория коэффициентов достоверности фактически является аппроксимацией по отношению к классической теории вероятностей.

5.4 Теория Демпстера–Шефера

В настоящем разделе рассматривается один из методов формирования нестрогих рассуждений, называемый **теорией Демпстера–Шефера**, или теорией Шефера–Демпстера. Этот метод основан на работе, первоначально выполненной Демпстером, который предпринял попытку смоделировать неопределенность, задавая ряд вероятностей, а не отдельное вероятностное значение. В дальнейшем Шефер дополнил и уточнил результаты, полученные Демпстером, в книге *A Mathematical Theory of Evidence*, опубликованной в 1976 году. Еще одно расширение, получившее название **рассуждений на основе свидетельств**, касалось той информации, которая, согласно ожиданиям, является неопределенной, неточной и иногда неправильной. Теория Демпстера–Шефера имеет хороший теоретический фундамент. К тому же можно показать, что теория коэффициентов достоверности является частным случаем теории Демпстера–Шефера, что позволяет перевести методы, основанные на использовании коэффициентов достоверности, на теоретическую, а не произвольную основу [31]. Кроме того, теория Демпстера–Шефера нашла свое применение в интеллектуальных базах данных, в которых используется анализ скрытых закономерностей в данных для извлечения образов [7].

Рамки различения

Теория Демпстера–Шефера основана на предположении о том, что задано фиксированное множество взаимоисключающих и исчерпывающих элементов, называемое **средой** и символически обозначаемое греческой буквой Θ:

$$\Theta = \{\theta_1, \theta_2, \ldots \theta_N\}$$

Термин “среда” аналогичен термину “универсум”, применяемому в теории множеств для обозначения объектов, о которых идет речь. Иными словами, среда — это множество объектов, представляющих для нас интерес. Ниже приведены некоторые примеры определения вариантов среды.

$$\Theta = \{\text{airliner}, \text{bomber}, \text{fighter}\}$$
$$\Theta = \{\text{red}, \text{green}, \text{blue}, \text{orange}, \text{yellow}\}$$
$$\Theta = \{\text{barn}, \text{grass}, \text{person}, \text{cow}, \text{car}\}$$

Обратите внимание на то, что элементы в каждой среде являются взаимоисключающими. Например, авиалайнер (airliner) — не бомбардировщик (bomber) и не истребитель (fighter), красный цвет (red) — не зеленый (green), трава (grass) — не корова (cow) и т.д. Предположим, что все возможные элементы универсума находятся в заданном множестве, таким образом, множество является

исчерпывающим. Кроме того, чтобы упростить приведенное здесь описание, предположим, что множество Θ является конечным. Тем не менее были проведены исследования, основанные на использовании вариантов среды Демпстера–Шефера, элементами которых являются непрерывные переменные, такие как время, расстояние, скорость и т.д. Один из способов размышления о структуре множества Θ состоит в том, что рассматриваются вопросы и ответы, относящиеся к этому множеству. Предположим, что дано приведенное ниже множество, а вопрос, касающийся элементов этого множества, сформулирован так: "Какие из этих самолетов имеют военное назначение?".

$$\Theta = \{\text{airliner}, \text{bomber}, \text{fighter}\}$$

Ответом становится следующее подмножество множества Θ:

$$\{\theta_2, \theta_3\} = \{\text{bomber}, \text{fighter}\}$$

Аналогичным образом, ответом на вопрос: "Какие из этих самолетов имеют гражданское назначение?" является множество

$$\{\theta_1\} = \{\text{airliner}\}$$

Такое множество называется **одноэлементным множеством**, поскольку содержит только один элемент.

Каждое подмножество множества Θ может интерпретироваться как возможный ответ на некоторый вопрос. А так как все элементы являются взаимоисключающими и среда — исчерпывающей, то правильным ответом на любой вопрос может служить только одно подмножество. Безусловно, не все возможные вопросы могут иметь смысл, к тому же не представляет интереса сам по себе поиск ответов на все возможные вопросы. Но важно понять, что именно все подмножества среды представляют собой все возможные действительные ответы в этом универсуме элементов, о которых идет речь. Каждое подмножество может рассматриваться как подразумеваемое высказывание, например, как в приведенных ниже случаях и в других подобных случаях, касающихся всех подмножеств, где начало высказывания "правильным ответом является" подразумевается для данного подмножества.

Правильным ответом является $\{\theta_1, \theta_2, \theta_3\}$
Правильным ответом является $\{\theta_1, \theta_2\}$

Все возможные подмножества в среде представления типов самолетов приведены на рис. 5.6. Линии, проведенные на этом рисунке, показывают отношения между подмножествами. Для обозначения элементов airliner (авиалайнер), bomber (бомбардировщик) и fighter (истребитель) используются буквы A, B и F. Эта

диаграмма изображена в виде иерархической решетки, в которой множество Θ находится вверху, а пустое множество $\varnothing = \{\}$ — внизу. Пустое множество обычно явно не показывают, поскольку оно всегда соответствует ложному ответу. Так как множество $\varnothing$ не имеет элементов, его выбор в качестве приведенного ниже ответа противоречит предположению о том, что данная среда является исчерпывающей.

Правильным ответом является отсутствие элемента

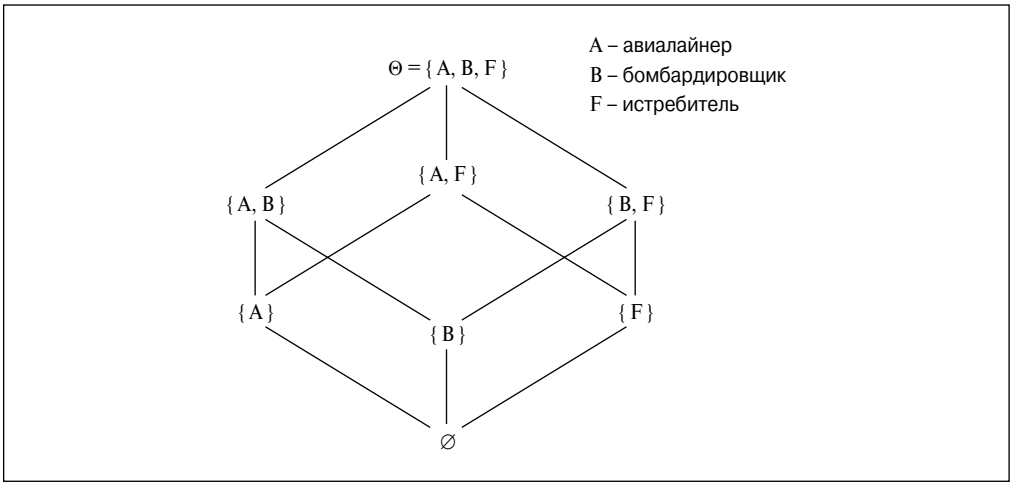


Рис. 5.6. Все подмножества среды, в которой рассматриваются типы самолетов

Обратите внимание на то, что диаграмма, приведенная на рис. 5.6, представляет собой решетку, а не дерево, поскольку узлы подмножеств могут иметь больше одного родительского узла. Кроме того, эта решетка является иерархической, так как она вычерчивается, начиная с более крупных множеств и заканчивая более мелкими. Например, один из путей от узла Θ к узлу $\varnothing$ выражает иерархическое отношение подмножеств, соединяющее родительские подмножества с дочерними, как показано ниже.

$$\varnothing \subset \{A\} \subset \{A, B\} \subset \{A, B, F\}$$

Напомним, что в разделе 2.10 было указано — наличие такого отношения между двумя множествами, X и Y:

$$X \subseteq Y$$

означает, что все элементы X являются элементами Y и записываются более формально следующим образом:

$$X \subseteq Y = \{x \mid x \in X \rightarrow x \in Y\}$$

Приведенная выше формула представляет собой утверждение, что если x — элемент множества X, из этого следует, что x также является элементом множества Y. Если $X \subseteq Y$, но $X \neq Y$, то имеется по меньшей мере один элемент Y, не являющийся элементом X. В таком случае X называется строгим подмножеством Y и приведенное выше выражение записывается так:

$$X \subset Y$$

Если все элементы среды могут интерпретироваться как возможные ответы и только один ответ является правильным, то среду называют **рамками различения** (frame of discernment). В данном случае термин "различение" показывает, что существует возможность отличить один правильный ответ от всех других возможных ответов на вопрос. Если ответ не находится в рамках, то рамки необходимо расширить, чтобы в них можно было задать дополнительные знания об элементах θ_{N+1}, θ_{N+2} и т.д. А для того чтобы только один ответ был правильным, требуется, чтобы множество являлось исчерпывающим, а все подмножества — несвязанными.

Множество с количеством элементов N имеет точно 2^N подмножеств, включая самого себя. Все эти подмножества определяют степенное множество (см. задачу 2.11), а это записывается как $P(\Theta)$. Таким образом, для среды, представляющей типы самолетов, имеет место следующее:

$$P(\Theta) = \{\varnothing, \{A\}, \{B\}, \{F\}, \{A, B\}, \{A, F\}, \{B, F\}, \{A, B, F\}\}$$

Степенное множество среды включает в качестве своих элементов все ответы на все возможные вопросы из рамок различения. Это означает, что существует взаимно однозначное соответствие между элементами $P(\Theta)$ и подмножествами множества Θ.

Массовые функции и незнание

Согласно байесовской теории, апостериорная вероятность по мере приобретения все новых и новых свидетельств изменяется. Аналогичным образом, в теории Демпстера–Шефера может изменяться степень доверия к свидетельству. Кроме того, в теории Демпстера–Шефера принято рассматривать степень доверия к свидетельству как аналогичную **массе** физического объекта. Иными словами, масса свидетельства поддерживает степень доверия. **Мера доверия к свидетельству**, символически обозначаемая буквой m, аналогична мере, с помощью которой можно судить о массе. Вместо термина "масса" применяется также термин **основное присваивание вероятности** (**basic probability assignment — bpa**), а иногда просто **основное присваивание**, поскольку по своей форме уравнения, описывающие плотности вероятностей и массы, являются аналогичными. Но при использовании

таких терминов может возникнуть путаница с терминами теории вероятностей, поэтому в настоящей книге такие термины не используются и речь идет просто о массе. Причина, по которой применяется аналогия с объектом, обладающим массой, состоит в том, что степень доверия может рассматриваться как величина, которую можно перемещать, дробить и комбинировать. Иногда удобно также рассматривать объект, обладающий массой, как состоящий из мягкой глины, что дает возможность удалять и снова прилеплять части этого объекта.

Фундаментальное различие между теорией Демпстера–Шефера и теорией вероятностей состоит в том, что в этих теориях понятие **незнания** трактуется поразному. Как было описано в главе 4, согласно теории вероятностей рассматриваемая вероятность должна подразделяться на равные части даже в случае незнания. Например, если отсутствуют априорные знания, то мы обязаны исходить из предположения, что вероятность P каждого возможного случая определяется следующей формулой, где N — количество возможных случаев:

$$P = \frac{1}{N}$$

Кроме того, как было сказано в главе 4, такое присваивание значения P применяется “в безвыходной ситуации” или, если используются термины, которые звучат менее впечатляюще, на основании **принципа безразличия**.

Крайний случай применения принципа безразличия возникает, если имеются только два возможных случая, таких как наличие нефти или отсутствие нефти, что можно обозначить символически как H и H'. В случаях, подобных этому, $P = 50\%$, даже если вообще нет никаких знаний о том, имеется ли нефть или нет, поскольку теория вероятностей гласит, что справедлива следующая формула:

$$P(H) + P(H') = 1$$

Таким образом, все, что не обосновывает гипотезу, должно ее опровергать, поскольку возможность незнания не допускается.

Если подобный подход применяется без размышлений, то могут обнаруживаться некоторые нелепые последствия. Например, допустим, что человек размышляет, есть месторождение нефти под его домом или нет. Согласно принципу безразличия, если полностью отсутствуют какие-либо другие знания, то вероятность наличия месторождения нефти под домом равна 50%. Стоит только об этом подумать, можно прийти к выводу, что 50%-ные шансы наличия нефти являются весьма впечатляющими и предоставляют гораздо лучшую возможность быстро разбогатеть, чем с помощью любых других законных капиталовложений. А поскольку есть 50%-ные шансы найти нефть, то не следует ли немедленно снять все свои сбережения, нанять буровую установку и приступить к бурению скважины в кухне?!

Если следовать той же линии рассуждений, то применение принципа безразличия и теории вероятностей позволяет прийти к выводу о наличии 50%-ных шансов найти под своим домом не только то, что перечислено ниже, но и все, что можно себе представить:

```
алмазы
сокровища пиратов
меховые шубы
зеленый сыр
ваше очередное домашнее задание
```

(Но в действительности вам, возможно, удастся разбогатеть, просто отправившись на телевидение и сообщив всем и каждому, что они могут стать богатыми, отправив вам 9,95 доллара за вашу книгу по теории вероятностей. В таком подходе к трактовке незнания также состоит одно из лучших оправданий для тех, кто не любит чистить зубы. А если вы не знаете, что находится там, где вас нет, то у вас всегда имеются 50%-ные шансы стать богатым, обнаружив в том месте кое-что ценное.)

Но даже если принцип безразличия не используется, следующее ограничение принудительно диктует необходимость присваивания вероятности отрицанию гипотезы и при отсутствии свидетельства, относящегося к отрицанию:

$$P(H) + P(H') = 1$$

Как было описано в разделе 5.3, такое предположение не совсем приемлемо по отношению к степеням доверия многих типов, например, касающихся медицинских знаний. Но теория вероятностей требует, чтобы свидетельство, которое не обосновывает гипотезу, опровергало ее.

С другой стороны, теория Демпстера–Шефера не вынуждает назначать степень доверия незнанию или опровержению гипотезы. Вместо этого масса присваивается только тем подмножествам среды, которым желательно назначить некоторую степень доверия. Вся степень доверия, не присвоенная конкретному подмножеству, рассматривается как **степень отсутствия доверия** (no belief), или **степень нехватки доверия** (nonbelief), и связывается со средой Θ. А степень доверия, которая опровергает гипотезу, представляет собой *степень недоверия* (disbelief), которую не следует путать со степенью отсутствия доверия.

Например, предположим, что некоторый датчик, такой как датчик системы опознавания “свой–чужой” (Identification Friend or Foe — IFF), не получает ответа от радиомаяка-ответчика самолета. Работа системы опознавания “свой–чужой” основана на использовании радиопередатчика/приемника, который передает радиограмму на самолет и принимает ответ. Если самолет относится к собственному воздушному флоту (является “своим”), его радиомаяк-ответчик должен отреагировать на радиограмму, отправив в ответ идентификационный код. Самолеты,

которые не отвечают, по умолчанию рассматриваются как “чужие”. Но самолет может не ответить на сигналы системы опознавания “свой–чужой” по многим причинам, в частности, описанным ниже.

- Неисправность в системе опознавания “свой–чужой”.
- Неисправность в радиомаяке-ответчике самолета.
- Отсутствие на самолете системы опознавания “свой–чужой”.
- Исчезновение сигнала опознавания “свой–чужой” в помехах.
- Получение приказа соблюдать режим радиомолчания.

Предположим, что неудачная попытка системы опознавания “свой–чужой” получить ответ указывает на наличие степени доверия 0.7 к свидетельству, что рассматриваемый самолет является “чужим”, причем как “чужие” самолеты рассматриваются только бомбардировщики и истребители. Таким образом, присваивание массы подмножеству $\{B, F\}$ осуществляется по следующей формуле, в которой m_1 обозначает первое свидетельство датчика опознавания “свой–чужой”:

$$m_1(\{B, F\}) = 0.7$$

Остальная часть степени доверия присваивается среде, Θ, как степень отсутствия доверия:

$$m_1(\Theta) = 1 - 0.7 = 0.3$$

Каждое подмножество в степенном множестве среды, имеющее массу больше 0, рассматривается как **фокальный элемент**. Термин “фокальный элемент” применяется в связи с тем, что подмножество X, такое что $m(X) > 0$, представляет собой элемент степенного множества, в котором фокусируется (или концентрируется) доступное свидетельство.

В этом теория Демпстера–Шефера существенно отличается от теории вероятностей, в которой было бы принято следующее предположение:

$$P(\text{hostile}) = 0.7$$
$$P(\text{non-hostile}) = 1 - 0.7 = 0.3$$

Согласно теории вероятностей, если степень доверия к гипотезе, что самолет является “чужим”, равна 0.7, то степень недоверия тому, что он “чужой”, должна быть равна 0.3. А в теории Демпстера–Шефера значение 0.3 рассматривается не как степень недоверия, а как степень отсутствия доверия к среде, выраженная в виде $m(\Theta)$. Это означает, что ни доверие, ни недоверие к свидетельству не имеют степень 0.3. Мы доверяем гипотезе, что рассматриваемый самолет является “чужим”, в степени 0.7 и резервируем суждение, соответствующее степени 0.3, за недоверием и дополнительным доверием к гипотезе, что самолет “чужой”. Очень важно понимать, что присваивание значения 0.3 среде Θ не представляет собой

присваивание какого-либо значения другим подмножествам множества Θ, даже несмотря на то, что в число этих подмножеств входят подмножества, охватывающие разные варианты появления "чужих" самолетов, $\{B, F\}$, $\{B\}$ и $\{F\}$.

Возвратившись к примеру с учащимся, который рассматривался в последнем разделе, отметим, что присваивание

$$m(\text{getting an 'A' and graduating}) = 0.7$$

не означает автоматически применение такого присваивания:

$$m(\text{getting an 'A' and not graduating}) = 0.3$$

если обеим этим массам не были специально присвоены значения.

Как показано в табл. 5.3, применение понятия массы обеспечивает намного бóльшую свободу выбора по сравнению с понятием вероятности.

Таблица 5.3. Сравнение возможностей теории Демпстера–Шефера (в которой применяется понятие массы) и теории вероятностей (в которой применяется понятие вероятностей)

Теория Демпстера–Шефера	**Теория вероятностей**
Значение $m(\Theta)$ не обязательно должно быть равно 1	$\sum_i P_i = 1$
Если $X \subseteq Y$, то требование о соблюдении равенства $m(X) = m(Y)$ не является обязательным	$P(X) \leqslant P(Y)$
Не требуется наличия связи между $m(X)$ и $m(X')$	$P(X) + P(X') = 1$

Каждую массу можно формально представить с помощью функции, которая отображает каждый элемент степенного множества в вещественное число, находящееся в интервале от 0 до 1. Под этим подразумевается, что степень доверия к некоторому подмножеству может принимать любые значения от 0 до 1. Указанное отображение формально представляется с помощью следующей формулы:

$$m : P(\Theta) \to [0, 1]$$

В соответствии с принятым соглашением масса пустого множества обычно определяется как равная нулю:

$$m(\varnothing) = 0$$

а сумма масс всех подмножеств X степенного множества равна 1:

$$\sum_{X \in P(\emptyset)} m(X) = 1$$

Например, в среде, в которой рассматриваются типы самолетов, справедлива следующая формула:

$$\sum_{X \in P(\emptyset)} m(X) = m(\{B, F\}) + m(\theta) = 0.7 + 0.3 = 1$$

Комбинирование свидетельств

Теперь рассмотрим случай, в котором становятся доступными дополнительные свидетельства. При этом хотелось бы иметь возможность комбинировать все свидетельства, чтобы выработать лучшую оценку степени доверия к свидетельству. Чтобы было проще ознакомиться с тем, как это делается, вначале рассмотрим пример, в котором применяется частный случай общей формулы комбинирования свидетельств.

Предположим, что для распознавания самолетов применен датчик второго типа, который определяет, что рассматриваемый самолет представляет собой бомбардировщик, со степенью доверия к свидетельству, равной 0.9. Теперь массы свидетельств, полученных от обоих датчиков, принимают следующий вид:

$$m_1(\{B, F\}) = 0.7 \qquad m_1(\Theta) = 0.3$$
$$m_2(\{B\}) = 0.9 \qquad m_2(\Theta) = 0.1$$

В этих формулах переменные m_1 и m_2 относятся к датчикам первого и второго типов.

Полученные свидетельства можно скомбинировать с помощью следующей специальной формы **правила комбинирования Демпстера** для получения такой **комбинированной массы**:

$$m_1 \oplus m_2(Z) = \sum_{X \cap Y = Z} m_1(X) m_2(Y)$$

В этой формуле операция суммирования распространяется на все элементы, для которых пересечение $X \cap Y = Z$. Знак **операции** $\oplus$ соответствует операции **ортогональной суммы**, или **прямой суммы**. Результат этой операции определяется путем суммирования пересечений произведений масс правой части правила. Правило комбинирования Демпстера позволяет комбинировать массы для получения новой массы, представляющей собой **консенсус** по отношению к оригинальным, возможно конфликтующим свидетельствам. Такую новую массу принято называть *консенсусом*, поскольку использование ее значения, как правило, способствует достижению соглашения, а не возникновению разногласий, так как в пересечения множеств включаются только массы. Пересечения множеств представляют общие элементы свидетельств. Важно отметить, что это правило должно применяться для комбинирования свидетельств, имеющих взаимно независимые ошибки, а это — не то же самое, что независимо собранные свидетельства.

В табл. 5.4 в виде одной таблицы показаны массы и пересечения произведений для среды с самолетами разных типов. За каждым обозначением пересечения множеств следует соответствующее ему числовое значение произведения масс.

Записи в этой таблице были вычислены путем перекрестного умножения произведений масс по строкам и столбцам, как показано ниже, где T_{ij} обозначает i-ю

Таблица 5.4. Подтверждение свидетельств

	$\boldsymbol{m_2(\{B\}) = 0.9}$	$\boldsymbol{m_2(\Theta) = 0.1}$
$\boldsymbol{m_1(\{B, F\}) = 0.7}$	$\{B\}0.63$	$\{B, F\}0.07$
$\boldsymbol{m_1(\Theta) = 0.3}$	$\{B\}0.27$	$\Theta 0.03$

строку и j-й столбец таблицы.

$$T_{11}(\{B\}) = m_1(\{B, F\})m_2(\{B\}) = (0.7)(0.9) = 0.63$$
$$T_{21}(\{B\}) = m_1(\Theta)m_2(\{B\}) = (0.3)(0.9) = 0.27$$
$$T_{12}(\{B, F\}) = m_1(\{B, F\})m_2(\Theta) = (0.7)(0.1) = 0.07$$
$$T_{22}(\Theta) = m_1(\Theta)m_2(\Theta) = (0.3)(0.1) = 0.03$$

Вслед за вычислением отдельных произведений масс в соответствии с описанными формулами выполняется сложение произведений по общим пересечениям множеств, согласно правилу Демпстера, следующим образом:

$m_3(\{B\}) = m_1 \oplus m_2(\{B\}) = 0.63 + 0.27 = 0.90$	бомбардировщик
$m_3(\{B, F\}) = m_1 \oplus m_2(\{B, F\}) = 0.07$	бомбардировщик или истребитель
$m_3(\Theta) = m_1 \oplus m_2(\Theta) = 0.03$	отсутствие доверия

Значение $m_3(\{B\})$ выражает доверие к тому, что рассматриваемый самолет представляет собой бомбардировщик и только бомбардировщик. Но под значениями $m_3(\{B, F\})$ и $m_3(\Theta)$ подразумевается дополнительная информация. Соответствующие множества включают бомбардировщик, поэтому правдоподобно допущение, что их ортогональные суммы могут внести свой вклад в определение степени доверия к тому, что рассматриваемый самолет является бомбардировщиком. Поэтому значение суммы $0.07 + 0.03 = 0.1$ для этих множеств может быть добавлено к степени доверия, касающейся подмножества бомбардировщика, для получения максимальной степени доверия к гипотезе, что самолет может быть бомбардировщиком — к степени 0,90, т.е. правдоподобной степени доверия. Таким образом, степень доверия не ограничивается одним значением, а выражается в виде **ряда степеней доверия** к свидетельству. В данном случае ряд степеней доверия начинается с минимального значения 0.9, согласно которому известно, что рассматриваемый самолет — бомбардировщик, до максимального правдоподобного значения степени доверия, равного $0.90+0.1 = 1$, что этот самолет может представлять собой бомбардировщик. При этом предполагается, что истинная степень доверия находится где-то в диапазоне от 0.9 до 1.

В таких рассуждениях на основании свидетельств считается, что свидетельство вводит **интервал проявления свидетельства** (evidential interval). При этом

в рассуждениях на основе свидетельств **нижняя граница** интервала называется **обоснованием** (**support — Spt**), а в теории Демпстера–Шефера обозначается как **Bel** (belief). С другой стороны, **верхнюю границу** принято называть **правдоподобием** (**plausibility — Pls**). Для данного примера интервал свидетельств равен $[0.90, 1]$, т.е. нижняя граница равна 0.90, а верхняя граница — 1. Обоснование представляет собой минимальную степень доверия, основанную на свидетельстве, а правдоподобие — максимальную степень доверия, которую желательно достичь. Вообще говоря, диапазоны, в которых изменяются Bel и Pls, выражаются соотношением $0 \leqslant \text{Bel} \leqslant \text{Pls} \leqslant 1$. В теории Демпстера–Шефера нижнюю и верхнюю границы иногда называют нижней и верхней вероятностями, согласно оригинальной статье Демпстера. В табл. 5.5 показаны некоторые широко применяемые интервалы проявления свидетельств.

Таблица 5.5. Некоторые широко применяемые интервалы проявления свидетельств

Интервал проявления свидетельства	Область определения переменной	Значение
$[1, 1]$		Полностью истинный
$[0, 0]$		Полностью ложный
$[0, 1]$		Полностью неизвестный
$[\text{Bel}, 1]$	$0 < \text{Bel} < 1$	Как правило, обосновывающий
$[0, \text{Pls}]$	$0 < \text{Pls} < 1$	Как правило, опровергающий
$[\text{Bel}, \text{Pls}]$	$0 < \text{Bel} \leqslant \text{Pls} < 1$	Как правило, и обосновывающий, и опровергающий

Обоснование, или **доверительная функция**, **Bel**, представляет собой общую степень доверия к множеству и всем его подмножествам. Таким образом, Bel — это вся масса, которая обосновывает множество и определяется в терминах массы:

$$\text{Bel}(X) = \sum_{Y \subseteq X} m(Y)$$

Например, в данной среде с типами самолетов для первого датчика справедливо следующее соотношение:

$$\begin{aligned}\text{Bel}_1(\{B, F\}) &= m_1(\{B, F\}) + m_1(\{B\}) + m_1(\{F\}) = \\ &= 0.7 + 0 + 0 = 0.7\end{aligned}$$

Функцию `Bel` иногда называют **мерой доверия**, или просто **доверием**. Но следует отметить, что доверительная функция весьма отличается от массы, которая представляет собой степень доверия к свидетельству, присвоенную единственному множеству. Например, предположим, что вы владеете автомобилем марки Ford

и услышали, что полиция разыскивает какой-то Ford, на котором преступники убежали от погони после ограбления банка. Но сообщение о том, что полиция разыскивает какой-то Ford, весьма отличается от сообщения, согласно которому ведется розыск именно вашего автомобиля Ford. Масса — это степень доверия к множеству, а не к какому-либо из его подмножеств, а доверительная функция применяется к множеству и ко всем его подмножествам. Значение Bel представляет собой суммарную степень доверия и поэтому является более глобальной, чем локальная степень доверия, выражаемая массой. В связи с тем, что в теории Демпстера–Шефера определены такие взаимосвязи между значениями массы и функции Bel, ее также называют **теорией доверительных функций**. Таким образом, правило Демпстера в определенном смысле можно интерпретировать как способ комбинирования доверительных функций. Масса и доверительная функция связаны следующим соотношением:

$$m(X) = \sum_{Y \subseteq X} (-1)^{|X-Y|} \text{Bel}(Y)$$

В этой формуле $|X - Y|$ представляет собой **кардинальность** множества:

$$X - Y = \{x \mid x \in X \text{ и } x \notin Y\}$$

Таким образом, $|X - Y|$ — это количество элементов в множестве $X - Y$.

Итак, доверительные функции определяются в терминах масс, поэтому комбинация двух доверительных функций также может быть выражена в терминах ортогональных сумм масс множества и всех его подмножеств, например, как показано ниже.

$$\begin{aligned}\text{Bel}_1 \oplus \text{Bel}_2(\{B\}) &= m_1 \oplus m_2(\{B\}) + m_1 \oplus m_2(\varnothing) = \\ &= 0.90 + 0 = 0.90\end{aligned}$$

В обычном случае масса пустого множества не записывается, поскольку, вообще говоря, она определяется как равная нулю. Суммарная степень доверия к подмножеству $\{B, F\}$, состоящему из бомбардировщика и истребителя, включает больше подмножеств, чем приведенное выше множество:

$$\begin{aligned}\text{Bel}_1 \oplus \text{Bel}_2(\{B, F\}) &= m_1 \oplus m_2(\{B, F\}) + m_1 \oplus m_2(\{B\}) + m_1 \oplus m_2(\{F\}) = \\ &= 0.07 + 0.90 + 0 = 0.97\end{aligned}$$

В это выражение включены термы для множеств $\{B\}$ и $\{F\}$, поскольку они представляют собой подмножества множества $\{B, F\}$. В том, что множество $\{B, F\}$ имеет подмножества $\{B\}$ и $\{F\}$, можно убедиться на основании рис. 5.6. Подмножеству $\{F\}$ масса не присвоена, поэтому $m(\{F\}) = 0$, и это подмножество не вносит никакого вклада в сумму. В действительности $m(\{F\})$ и другие

массы, равные нулю, вообще не вводились в табл. 5.4, поскольку результат любого перекрестного произведения между ними будет равен нулю. Если бы массы были присвоены каждому подмножеству множества $\{A, B, F\}$, кроме пустого множества (имеющего нулевую массу), то табл. 5.4 представляла бы собой таблицу из 49 ячеек, согласно расчету $(2^3 - 1)(2^3 - 1) = 7 \cdot 7 = 49$.

Комбинированная доверительная функция для Θ, основанная на всех свидетельствах, представляет собой следующее:

$$\begin{aligned}\text{Bel}_1 \oplus \text{Bel}_2(\Theta) &= m_1 \oplus m_2(\Theta) + m_1 \oplus m_2(\{B, F\}) + m_1 \oplus m_2(\{B\}) = \\ &= 0.03 + 0.07 + 0.90 = 1\end{aligned}$$

Фактически $\text{Bel}(\Theta) = 1$ во всех случаях, поскольку сумма масс всегда должна быть равна 1. При комбинировании свидетельств просто происходит перераспределение масс по разным подмножествам.

Интервал проявления свидетельства множества S, $EI(S)$, может быть определен в терминах степени доверия следующим образом:

$$EI(S) = [\text{Bel}(S), 1 - \text{Bel}(S')]$$

Если $S = \{B\}$, то $S' = \{A, F\}$ и имеет место следующая формула, поскольку имеются элементы, отличные от фокальных, то масса нефокальных элементов равна нулю:

$$\begin{aligned}\text{Bel}(\{A, F\}) &= m_1 \oplus m_2(\{A, F\}) + m_1 \oplus m_2(\{A\}) + m_1 \oplus m_2(\{F\}) = \\ &= 0 + 0 + 0 = 0\end{aligned}$$

Таким образом, интервал проявления свидетельств для $\{B\}$ представляет собой следующее:

$$\begin{aligned}EI(\{B\}) &= [0.90, 1 - 0] = \\ &= [0.90, 1]\end{aligned}$$

Аналогичным образом, если $S = \{B, F\}$, то $S' = \{A\}$ и имеет место следующее, поскольку $\{A\}$ — нефокальный элемент:

$$\text{Bel}(\{A\}) = 0$$

Кроме того, справедливы приведенные ниже соотношения, в которых интервал проявления свидетельств $[0, 1]$ отражает суммарную степень незнания применительно к подмножеству $\{A\}$.

$$\begin{aligned}\text{Bel}(\{B, F\}) &= \text{Bel1} \oplus \text{Bel2}(\{B, F\}) = 0.97 \\ EI(\{B, F\}) &= [0.97, 1 - 0] = [0.97, 1] \\ EI(\{A\}) &= [0, 1]\end{aligned}$$

Таким образом, интервал проявления свидетельств [суммарная_степень_доверия,правдоподобие] может быть выражен таким образом:

[обосновывающее_свидетельство,обосновывающее_свидетельство + незнание]

В соответствии с теорией вероятностей этот интервал сводится к следующей единственной точке, поскольку в теории вероятностей наличие незнания не допускается:

[обосновывающее_свидетельство,обосновывающее_свидетельство]

Это означает, что в теории вероятностей свидетельство, которое не обосновывает гипотезу, должно ее опровергать, например, по такому принципу: "Если данные перчатки вам не подходят, то вы должны ходить вообще без перчаток".

Правдоподобие определяется как степень, в которой свидетельство не в состоянии опровергнуть гипотезу X:

$$\text{Pls}(X) = 1 - \text{Bel}(X') = 1 - \sum_{Y \subseteq X} m(X')$$

А правдоподобная степень доверия, Pls, расширяет степень доверия до абсолютного максимума, в рамках которого неприсвоенная степень доверия $m(\Theta)$, возможно, окажется способной внести свой вклад в эту степень доверия. Безусловно, масса $m(\Theta)$ может включать бомбардировщик, истребитель или авиалайнер, но согласно предположению о правдоподобии подразумевается, что эта масса вносит дополнительную степень доверия только в одно из подмножеств множества Θ. А поскольку $\{B\}$ — подмножество множества Θ, то правдоподобно, что степень доверия 0.3, соответствующая массе $m_1(\Theta)$, может быть присвоена бомбардировщику. Как было отмечено в разделе 4.15, понятие правдоподобной степени доверия немного строже, чем понятие возможной степени доверия, но не обязательно соответствует степени доверия, поддерживаемой строгим свидетельством. Еще одно важное замечание состоит в том, что Θ — это не единственный тип множества, которое распространяет степень доверия на какое-либо подмножество X. Такое же действие выполняет любое множество, которое пересекается с X и с его дополнением.

Сомнительность (dubiety — **Dbt**), или **сомнение**, представляет собой степень, в которой выражается недоверие к гипотезе X или эта гипотеза опровергается. С другой стороны, **незнанием** (ignorance — **Igr**) называется степень, в которой масса обосновывает гипотезы X и X'. Эти значения определяются следующим образом:

$$\text{Dbt}(X) = \text{Bel}(X') = 1 - \text{Pls}(X)$$
$$\text{Igr}(X) = \text{Pls}(X) - \text{Bel}(X)$$

Нормализация степени доверия

Предположим, что теперь поступает конфликтующее свидетельство от третьего датчика, согласно которому рассматриваемый самолет представляет собой авиалайнер:

$$m_3(\{A\}) = 0.95 m(\Theta) = 0.05$$

В табл. 5.6 показано, как в данном случае вычисляются перекрестные произведения.

Таблица 5.6. Комбинирование свидетельств с учетом дополнительного свидетельства m_3

	$\boldsymbol{m_1 \oplus m_2(\{B\})}$ **0.90**	$\boldsymbol{m_1 \oplus m_2(\{B,F\})}$ **0.07**	$\boldsymbol{m_1 \oplus m_2(\Theta)}$ **0.03**
$\boldsymbol{m_3(\{A\}) = 0.95}$	$\varnothing 0$	$\varnothing 0$	$\{A\}0.0285$
$\boldsymbol{m_3(\Theta) = 0.05}$	$\{B\}0.045$	$\{B,F\}0.0035$	$\Theta 0.0015$

В данной таблице появилось пустое множество, $\varnothing$, в связи с тем, что подмножества $\{A\}$ и $\{B\}$ не имеют общих элементов и таковых не имеют также подмножества $\{A\}$ и $\{B,F\}$, поэтому их перекрестное произведение равно 0, а не 0.855 и 0.0665. Но вскоре будет показано, что такие нулевые значения находят свое применение в нормализации. Перекрестное произведение представляет собой произведение столбцов на строки и выражается следующими формулами:

$$m_1 \oplus m_2 \oplus m_3(\{A\}) = 0.0285$$
$$m_1 \oplus m_2 \oplus m_3(\{B\}) = 0.045$$
$$m_1 \oplus m_2 \oplus m_3(\{B,F\}) = 0.0035$$
$$m_1 \oplus m_2 \oplus m_3(\Theta) = 0.0015$$
$$m_1 \oplus m_2 \oplus m_3(\varnothing) = 0 \quad \text{(согласно определению пустого множества)}$$

Обратите внимание на то, что в данном примере сумма всех масс меньше 1:

$$\sum m_1 \oplus m_2 \oplus m_3(X) = 0.0285 + 0.045 + 0.0035 + 0.0015 = 0.0785$$

В этом выражении операция суммы охватывает все фокальные элементы. Тем не менее сумма должна быть равна 1, поскольку комбинированное свидетельство $m_1 \oplus m_2 \oplus m_3$ имеет действительную массу, а сумма по всем фокальным элементам должна составлять 1. Тот факт, что в данном случае сумма меньше 1, свидетельствует о наличии проблемы.

Решение этой проблемы состоит в осуществлении **нормализации** фокальных элементов путем деления каждого фокального элемента на следующее выражение:

$$1 - \kappa$$

В этом выражении значение κ (каппа) определено для каждого множества X и Y таким образом:

$$\kappa = \sum_{X \cap Y = \emptyset} m_1(X) m_2(Y)$$

В данном примере:

$$\kappa = 0.855 + 0.0665 = 0.9215$$

и поэтому рассматриваемое выражение равно:

$$1 - \kappa = 1 - 0.9215 = 0.0785$$

Деление каждого фокального элемента $m_1 \oplus m_2 \oplus m_3$ на $1 - \kappa$ приводит к получению следующих нормализованных значений:

$$\begin{aligned} m_1 \oplus m_2 \oplus m_3\{A\} &= 0.363 \\ m_1 \oplus m_2 \oplus m_3\{B\} &= 0.573 \\ m_1 \oplus m_2 \oplus m_3\{B, F\} &= 0.045 \\ m_1 \oplus m_2 \oplus m_3(\Theta) &= 0.019 \end{aligned}$$

Теперь общее нормализованное значение степени доверия $\{B\}$ равно следующему:

$$\text{Bel}(\{B\}) = m_1 \oplus m_2(\{B\}) = 0.573$$

Обратите внимание на то, что теперь лишь одно свидетельство в пользу подмножества $\{A\}$ существенно уменьшило степень доверия к подмножеству $\{B\}$, т.е., как и следовало ожидать, степень доверия уменьшилась с 0.90 до 0.573, или почти вдвое:

$$\begin{aligned} \text{Bel}(\{B\}') = \text{Bel}(\{A, F\}) = \\ = m_1 \oplus m_2 \oplus m_3(\{A, F\}) + \\ + m_1 \oplus m_2 \oplus m_3(\{A\}) + \\ + m_1 \oplus m_2 \oplus m_3(\{F\}) = \\ = 0 + 0.363 + 0 = 0.363 \end{aligned}$$

Таким образом, после этого интервал проявления свидетельств принимает следующий вид:

$$\begin{aligned} EI(\{B\}) = [\text{Bel}(\{B\}), 1 - \text{Bel}(\{B\}')] = \\ = [0.573, 1 - 0.363] = \\ = [0.573, 0.637] \end{aligned}$$

Обратите внимание на то, что обоснование и правдоподобие $\{B\}$ существенно уменьшились под влиянием конфликтующего свидетельства в пользу подмножества $\{A\}$. Итак, правило комбинирования Демпстера имеет такую общую форму:

$$m_1 \oplus m_2(Z) = \frac{\sum\limits_{X \cap Y = Z} m_1(X) m_2(Y)}{1 - \kappa}$$

В этой формуле значение κ еще раз определено для удобства. Если $\kappa = 1$, то ортогональная сумма не определяется:

$$\kappa = \sum_{X \cap Y = \emptyset} m_1(X) m_2(Y)$$

Значение κ показывает величину **конфликта свидетельств**. Если $\kappa = 0$, то имеет место полная совместимость гипотез, а если равно 1, — полное противоречие. Значения $0 < \kappa < 1$ показывают частичную совместимость.

Движущиеся массы и множества

Для лучшего объяснения таких понятий, как обоснование и правдоподобие, полезна аналогия с движущимися массами. Основные применяемые при этом концепции приведены ниже.

- *Обоснование* — это масса, присвоенная множеству и всем его подмножествам.
- Масса множества может свободно перемещаться на его подмножества.
- Масса множества не может перемещаться на его надмножества.
- Перемещение массы с множества на его подмножество позволяет внести вклад только в повышение правдоподобия подмножества, а не в его обоснование.
- Масса среды, т.е. множества Θ, может перемещаться на любое подмножество множества Θ, поскольку множество Θ находится вне всех подмножеств.

Как показано на рис. 5.7, *а*, предполагается, что вся масса находится в подмножестве X, поэтому имеет место следующее:

$$m(X) = 1$$

Это означает, что обоснование X равно 1. Правдоподобие X также равно 1, поскольку вся масса находится в X и нет никакого надмножества, которое могло бы внести дополнительную массу. Таким образом, имеем следующее:

$$EI(X) = [1, 1]$$

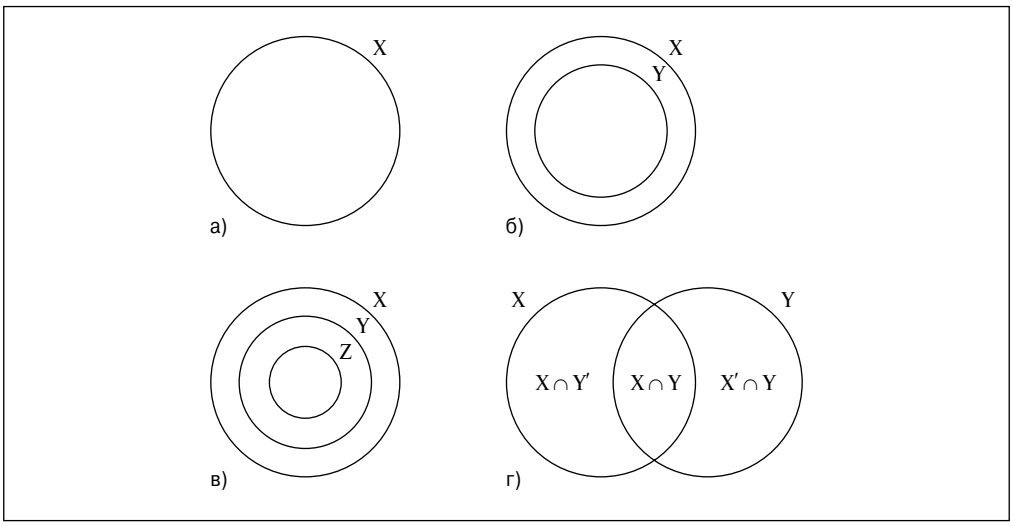


Рис. 5.7. Множества, применяемые для иллюстрации понятий обоснования и правдоподобия

Если $m(X) = 0.5$, то $EI(X) = [0.5, 0.5]$, и, вообще говоря, если $m(X) = a$, где a — любая константа, то $EI(X) = [a, a]$.

А на рис. 5.7, *б* принято предположение, что $m(X) = 0.6$ и $m(Y) = 0.4$, причем эти значения являются также обоснованиями для соответствующих подмножеств. Правдоподобие X равно 0.6, поскольку масса Y не может быть перемещена на X. Но масса X может быть перемещена внутрь Y, поскольку Y — подмножество X и поэтому правдоподобие Y равно $0.4 + 0.6 = 1$. Таким образом, интервалы проявления свидетельств для X и Y являются таковыми:

$$EI(\{X\}) = [0.6, 0.6]$$
$$EI(\{Y\}) = [0.4, 1]$$

Следует отметить, что рис. 5.7, *в* и *г* используются в задаче 5.4.

Сложности, связанные с применением теории Демпстера–Шефера

Одна из сложностей, связанных с использованием теории Демпстера–Шефера, возникает в связи с необходимостью применения нормализации и может привести к появлению результатов, противоречащих нашим ожиданиям. Проблема, связанная с нормализацией, состоит в том, что игнорируется степень доверия к тому, что рассматриваемый объект не существует.

Заде привел пример с двумя врачами, которые высказали степени доверия A и B к гипотезе о наличии у пациента определенного заболевания. В этой задаче

с пациентом рассматривались следующие значения степени доверия:

$$m_A(meningitis) = 0.99 \quad \text{(менингит)}$$
$$m_A(braintumor) = 0.01 \quad \text{(опухоль головного мозга)}$$
$$m_B(concussion) = 0.99 \quad \text{(сотрясение мозга)}$$
$$m_B(braintumor) = 0.01 \quad \text{(опухоль головного мозга)}$$

Обратите внимание на то, что оба врача соглашаются с наличием очень небольших шансов (`0.01`), что у пациента имеется опухоль головного мозга, но, когда речь идет об основной проблеме, их мнения в значительной степени расходятся. При этом применение правила комбинирования Демпстера приводит к получению комбинированной степени доверия к гипотезе о наличии опухоли головного мозга, равной 1. Этот результат воспринимается как неожиданный и противоречит нашей интуиции, поскольку оба врача согласились с тем, что опухоль головного мозга весьма маловероятна. Тот же результат, равный 1, касающийся опухоли головного мозга, возникает независимо от того, каковы другие вероятности. Это — хороший пример, подтверждающий истину, согласно которой недостаточно просто сойтись с кем-то во мнениях на общей почве для того, чтобы эта почва не оказалась зыбкой.

5.5 Приближенные рассуждения

В этом разделе рассматривается теория неопределенности, основанная на **нечеткой логике**. Эта теория в основном посвящена проблемам оценки количества и формирования рассуждений с использованием естественного языка, в котором многие слова, такие как “стройный”, “жаркий”, “опасный”, “немного”, “очень немного” и т.д., имеют неоднозначный смысл. Люди постоянно используют нечеткие правила `IF-THEN`, подобные следующему: “`IF` будильник наконец-то отключится `THEN` я еще немного посплю”. Но, как мы все знаем, выражение “еще немного поспать” — очень неточное, и каждый толкует его по-своему. А в последнее время нечеткие правила широко используются в программах [82], будучи либо непосредственно встроенными в инструментальное средство, либо добавленными к такому основному инструментальному средству, как CLIPS.

Нечеткая логика — это надмножество обычной (булевой) логики, которая была дополнена с учетом понятия частичной истинности — истинностных значений, лежащих между “полностью истинными” и “полностью ложными”. Определение понятия *нечеткой логики* не следует воспринимать буквально, поскольку нечеткая логика — это не туманный, расплывчатый, неопределенный способ мышления. В действительности данный подход к формированию рассуждений является абсолютно противоположным неопределенному. Неопределенным становится то, что воспринимается как слишком сложное для полного понимания. Чем сложнее

оказывается некоторое понятие, тем более неточным или “нечетким” оно становится. Нечеткая логика предоставляет точный подход к учету неопределенности, обусловленной сложностью человеческого поведения.

Понятие нечеткого множества было впервые сформулировано Заде в оригинальной статье, опубликованной в 1965 году. Это понятие стало теоретической основой для нечетких компьютерных систем и аппаратных средств, которые появились двадцать лет спустя. Теория, сформулированная Заде, привела к появлению принципиально новых ветвей математики, инженерии и науки. Появился термин **мягкие вычисления** (soft computing), означающий вычисления, не основанные на классической двухзначной логике [53]. Мягкие вычисления охватывают нечеткую логику, нейронные сети и вероятностные рассуждения. В наши дни термин “вероятностный” в искусственном интеллекте распространяется не только на классическую теорию вероятностей, но и на байесовские сети доверия, эволюционные вычисления, включая вычисления по принципу ДНК, теорию хаоса и квантовые вычисления.

Теория нечетких множеств в течение продолжительного времени своего развития была дополнена и применена во многих областях, таких как создание автоматических камер, отслеживающих отдельные объекты в пространстве [39]. Во многих приложениях нечеткая логика применяется также в сочетании с нейронными сетями [68]. Кроме того, нечеткая логика используется во многих моделях видео- и фотокамер. Несколько наиболее важных приложений нечеткой логики приведено в табл. 5.7. Чтобы ознакомиться с каким-то конкретным прило-

Таблица 5.7. Некоторые приложения теории нечетких множеств

Приложения
Алгоритмы управления
Медицинская диагностика
Принятие решений
Экономика
Инженерия
Охрана окружающей среды
Литература
Исследование операций
Распознавание образов
Психология
Теория надежности
Безопасность
Наука

жением, достаточно воспользоваться машиной поиска, и вас буквально захлестнет лавина информации.

Нечеткие множества и естественный язык

Традиционный способ представления информации о том, какие объекты являются элементами множества, состоит в использовании **характеристической функции**, иногда называемой **различительной функцией**. Если некоторый объект является элементом множества, то его характеристическая функция равна 1, а если объект не является элементом множества, то его характеристическая функция равна 0. Это определение можно кратко представить с помощью следующей характеристической функции, в которой объекты x являются элементами некоторого универсума X [80]:

$$\mu_A(x) = \begin{cases} 1, & \text{если } x - \text{элемент множества } A; \\ 0, & \text{если } x \text{ не является элементом множества } A \end{cases}$$

Характеристическая функция может быть также определена в терминах функционального отображения (см. раздел 1.10, посвященный описанию функционального программирования):

$$\mu_A(x) : X \to \{0, 1\}$$

Данная формула представляет собой формулировку утверждения, что характеристическая функция отображает универсальное множество X на множество, состоящее из элементов 0 и 1. В этом определении лишь выражается классическое понятие множества, согласно которому некоторый объект либо принадлежит, либо не принадлежит к множеству. Множества, к которым применяется это понятие, называются **четкими множествами**, в отличие от нечетких множеств. Указанный подход к изучению понятия множеств исходит из аристотелевских представлений о **двухзначной логике**, в которой возможны только истинные и ложные значения.

В отличие от традиционной, или классической, логики, в которой предпринимаются попытки классифицировать всю информацию с помощью бинарных шаблонов, таких как “белое/черное”, “истина/ложь”, “да/нет” или “все/ничего”, в нечеткой логике уделяется внимание “исключенному третьему” и предпринимается попытка учесть наличие “полутонов”, т.е. ситуаций, в которых приходится сталкиваться с частично истинными и частично ложными утверждениями, лежащими в основе большей части рассуждений человека в повседневной жизни. Нечеткая логика основана на предположении, что любые оценки выражаются в степенях, измеряемых по скользящей шкале, будь то истина, возраст, красота, благосостояние, цвет, скорость или любое другое понятие, оценка которого зависит от динамического характера поведения и восприятия человека.

Основной недостаток двухзначной логики обусловлен тем, что люди живут в аналоговом, а не в цифровом мире. В реальной действительности любые предметы обычно не находятся лишь в том или другом состоянии. Цифровые микросхемы, в которых соблюдается двухзначная логика, можно найти только в обычной компьютерной архитектуре. Но реальный мир гораздо точнее представляют такие искусственные объекты, как нейронные системы и системы, основанные на нечеткой логике, которые созданы в результате развития аналоговых теорий вычисления.

Особенностью нечетких множеств является то, что любой объект может принадлежать к множеству лишь частично. Степень принадлежности к нечеткому множеству измеряется с помощью **функции принадлежности**, или **функции совместимости**, являющейся обобщением *характеристической функции*, которая определяется следующим образом:

$$\mu_A(x) : X \to [0, 1]$$

Безусловно, на первый взгляд это определение весьма напоминает определение характеристической функции, но фактически между этими определениями существуют очень важные различия. Характеристическая функция отображает все элементы универсума X на один из элементов, количество которых точно равно двум: 0 или 1. В отличие от этого функция принадлежности отображает универсум X на область значений вещественных чисел, определенную в **интервале** от 0 до 1 включительно, который символически обозначается как $[0, 1]$. Таким образом, функция принадлежности представляет собой вещественное число и соответствует следующему ограничению, в котором 0 означает отсутствие принадлежности, а 1 обозначает полную принадлежность к множеству:

$$0 \leqslant \mu_A \leqslant 1$$

Конкретное значение функции принадлежности, такое как 0.5, называется **степенью принадлежности**.

Разумеется, вначале способ рассуждений о частичной принадлежности какого-то элемента к множеству может показаться странным, но фактически подобные рассуждения являются более естественными, чем рассуждения о классических двухзначных множествах. Несмотря на то что многие хотели бы, чтобы так и было, но реальный мир не сводится лишь к утверждению или отрицанию, черному или белому, правильному или неправильному, существующему или несуществующему. Так же как в действительности можно видеть много оттенков серого, а не просто белое и черное, в реальном мире обнаруживается много различных градаций смысла. Точными обязаны быть только долговые записи и исходные коды компьютерных программ.

Применение функции принадлежности позволяет описывать ситуации, которые складываются в реальном мире. В качестве очень простого примера рассмотрим облачные дни. В описании на основе четкого множества требуется принимать произвольные решения в отношении того, что представляет собой облачную погоду. Следует ли рассматривать как облачную такую погоду, в которой наблюдается немного облаков, много облаков, полностью закрытое облаками небо, частично закрытое облаками небо или какое-то другое определение? Следует ли считать дождливой погодой такую погоду, в которую наблюдается количество осадков, равное 2,5, 5 или 7,5 см, или же наблюдается строго определенная интенсивность осадков?

Нечеткие множества и понятия широко используются в естественном языке, что можно показать на примере следующих предложений, в которых слова, выделенные курсивным шрифтом, относятся к нечетким множествам и **кванторам**:

"John is *tall*" (Джон – *высокий*)
"The weather is *hot*" (Погода – *жаркая*)
"Turn the dial a *little higher*" (*Немного* поверните ручку настройки *в сторону увеличения*)
"Most tests are *hard*" (*Большинство* заданий – *сложные*)
"If the dough is *much too thick*, add *a lot of* water" (Если тесто *слишком крутое*, добавьте *побольше* воды)

Теория нечетких множеств позволяет представлять и оперировать со всеми подобными нечеткими множествами и кванторами. В частности, как вскоре будет показано, нечеткая логика позволяет справиться с квантором “большинство”, который оказался причиной наиболее серьезных ограничений в логике предикатов (см. раздел 2.16).

В естественном языке термины “расплывчатый” и “нечеткий” иногда используются как синонимы. Но в контексте теории нечетких множеств между этими терминами существует важное различие. **Нечеткое утверждение** может содержать такие слова, как “высокий” (tall), которые служат в качестве идентификаторов нечетких множеств, в данном случае TALL. В настоящей книге соблюдается соглашение по использованию для обозначения нечетких множеств, подобных этому, только прописных букв. В отличие от классического высказывания, такого как “Джон имеет рост, точно равный пяти футам”, которое представляет собой высказывание, являющееся либо истинным, либо ложным, нечеткое утверждение может иметь определенную степень истинности. Например, нечеткое утверждение “Джон имеет высокий рост” может быть истинным в определенной степени: “далекий от истины”, “немного более близкий к истине”, “довольно близкий к истине”, “весьма близкий к истине” и т.д. Нечеткое истинностное значение называется **нечетким спецификатором** и может использоваться в качестве нечеткого множества или модифицировать нечеткое множество. В отличие от четких вы-

сказываний, в которых не допускается наличие кванторов, нечеткие утверждения могут иметь **нечеткие кванторы**, такие как “большинство”, “много”, “обычно” и т.д., поэтому по отношению к ним не проводятся такие различия, как в классической логике, т.е. различия между утверждениями и высказываниями.

Термин “неопределенный” используется в том смысле, что некоторая информация является неполной. Например, предложение “Джон находится где-то здесь” является неопределенным, если в нем отсутствует информация, достаточная для принятия какого-то решения. Нечеткое утверждение, такое как “Он — высокий”, может стать неопределенным, если неизвестно, к кому относится местоимение “он”. Неопределенность также может измеряться некоторыми степенями. Такие высказывания, как “Джон — высокий”, являются менее неопределенными, чем “Он — высокий”, но все еще остаются неопределенными, если неизвестно, кто такой Джон.

В естественном языке используется много нечетких слов, например, таких, как показано в табл. 5.8. Вскоре будет описано, что смысл этих слов можно определить в терминах нечетких множеств. Кроме того, теория нечетких множеств позволяет определять сложные термины и манипулировать ими (табл. 5.9).

Таблица 5.8. Некоторые нечеткие термины естественного языка

Нечеткий термин
Высокий
Жаркий
Низкий
Средний
Высокий
Очень
Нет
Мало
Несколько
Немного
Много
Больше
Больше всего
Около
Приблизительно
Представитель левых

Таблица 5.9. Сложные нечеткие термины естественного языка

Сложный нечеткий термин
Более или менее низкий
Довольно низкий
Не низкий
Не очень низкий
Более или менее высокий
Средний или даже скорее высокий
Выше чем достаточно низкий
Низкий или даже скорее средний
Самый высокий
Либеральный представитель левых
Ультралиберальный представитель левых

Безусловно, трудно себе представить, как может объект лишь частично принадлежать множеству, поэтому гораздо проще воспользоваться другой трактовкой, согласно которой функция принадлежности представляет степень, в которой некоторый объект имеет определенный атрибут. Такое представление о степени наличия атрибута выражается с помощью альтернативного толкования функции принадлежности как функции совместимости. Термин **совместимость** показывает, насколько полно некоторый объект согласуется с определенным атрибутом, поэтому действительно позволяет лучше описывать нечеткие множества. Но в литературе гораздо чаще используется термин “функция принадлежности”, и в связи с этим данный термин будет применяться и в настоящей книге. Проводя рассуждения о нечетких множествах, можно обнаружить, что элементы нечеткого множества удобно представлять с помощью триплета “объект–атрибут–значение” (см. главу 2). Применительно к четким множествам могут рассматриваться только объект и атрибут, поскольку предполагается, что значение равно либо 0, либо 1. Это означает, что, проводя рассуждения о четком множестве, можно считать, что некоторый элемент либо принадлежит, либо не принадлежит к множеству. С другой стороны, для нечетких множеств значение может находиться в любом месте интервала от 0 до 1.

Для иллюстрации понятия нечеткого множества еще раз рассмотрим приведенный выше пример:

```
"John is tall" (Джон – высокий)
```

Одна из возможных функций принадлежности, касающихся того случая, когда рассматривается взрослый человек, показана на рис. 5.8. Согласно этому рисунку

считается, что любой человек, имеющий рост приблизительно 7 футов и выше, является высоким и имеет значение функции принадлежности, равное 1.0. С другой стороны, любой человек, имеющий рост меньше 5 футов, не считается принадлежащим к нечеткому множеству TALL, поэтому для него функция принадлежности равна 0. А между 5 и 7 футами функция принадлежности монотонно возрастает вместе с ростом. Обратите внимание на то, что добавление квантора "очень" приводит к получению другой функции принадлежности. Любопытно узнать, какую кривую предложил бы читатель для описания среднего роста?

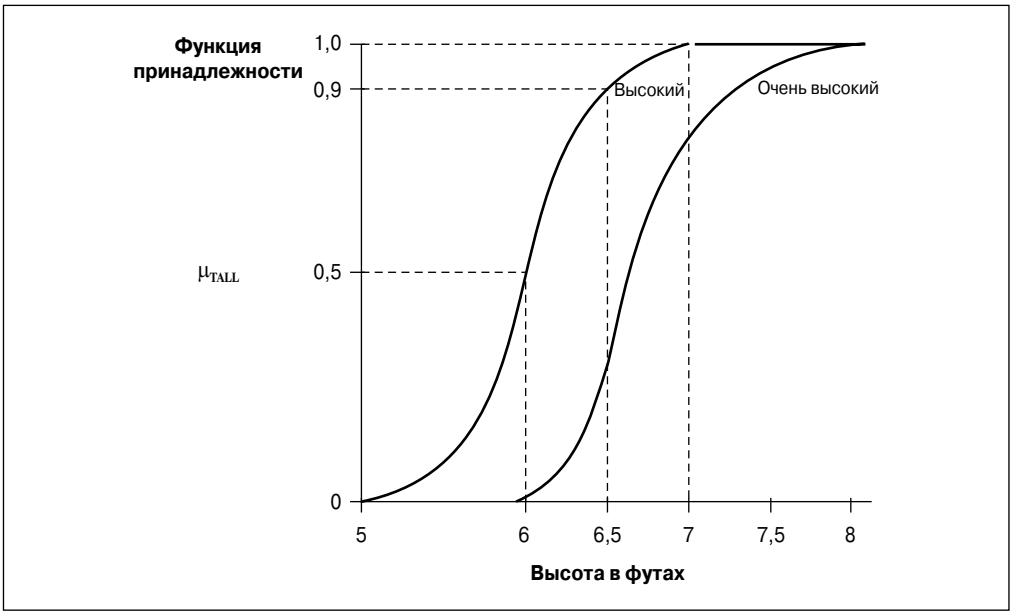


Рис. 5.8. Функция принадлежности для нечеткого множества TALL

Данная функция принадлежности является лишь одной из многих возможных функций. Для обычных людей, баскетболистов, жокеев и т.д. должны применяться весьма разные функции принадлежности. Например, жокей, имеющий рост пять футов, считается довольно высоким для своей профессии, даже несмотря на то, что значение функции принадлежности для лица с ростом пять футов на рис. 5.8 равно 0.

Функция принадлежности может быть сформирована на основании мнений одного человека или группы людей, в зависимости от приложения. В экспертной системе функция принадлежности формируется на основании мнения эксперта, моделируемого системой. Безусловно, маловероятно, что в какой-то экспертной системе будет моделироваться мнение о том, какие люди являются высокими, но широко применяется моделирование мнений, высказываемых по другому пово-

ду. В качестве некоторых примеров можно указать кредитный риск, относящийся к определенной ссуде, враждебные намерения неизвестного самолета, качество товара, пригодность кандидата для выполнения определенной работы и т.д. Обратите внимание на то, что мнения, подобные указанным, не сводятся к простым решениям, таким как “да” или “нет”. Хотя и возможно установить пороговые значения для принятия решений по принципу “да” или “нет”, существуют весьма реальные сомнения в отношении обоснованности четкого порога. Например, следует ли отказать какому-то человеку в предоставлении ссуды из-за того, что его доход составляет 29 999,99 доллара, а пороговое значение равно 30 000,00 долларам?

Кроме того, интуитивно можно себе представить, что функция принадлежности для группы людей может рассматриваться в терминах опроса общественного мнения. Предположим, что лица, проводящие опрос, останавливают на улице случайных прохожих и просят указать минимальное значение роста, к которому относится определение “высокий”. По-видимому, ни один из обычных людей не скажет, что высоким может считаться рост меньше 5 футов. Аналогичным образом, все участники опроса, скорее всего, назовут высоким человека, имеющего рост 7 футов и больше. А в интервале между 5 и 7 футами процентная доля людей, согласных с определением понятия “высокий рост”, будет примерно такой, как показывает кривая функции принадлежности, приведенная на рис. 5.8. По мере того как значение роста будет увеличиваться от 5 до 6 футов, все больше и больше людей будут соглашаться с тем, что некоторое лицо с таким ростом является высоким. Для данной конкретной функции принадлежности **точка пересечения** для понятия “высокий” равна 6 футам. *Точкой пересечения* называется такая точка, в которой $\mu = 0.5$. В терминах применяемой здесь аналогии с опросом общественного мнения можно считать, что 50% опрошенных согласятся с утверждением, будто человек, имеющий рост 6 футов, является высоким. На уровне 6,5 фута процент соглашающихся с этим утверждением людей составляет 90%. Начиная с роста 7 футов и выше все мнения по поводу высокого роста совпадают, поэтому кривая функции принадлежности устанавливается на уровне 1.

Данный пример сформулирован в терминах опроса общественного мнения, в котором участвует группа людей, но важно понять, что полученную функцию принадлежности фактически нельзя рассматривать как распределение частот. В главе 4 было показано, что для учета повторяющихся наблюдений над одними и теми же объектами используются вероятности. Но мнения должны оцениваться с точки зрения понятия правдоподобия, даже несмотря на то, что каждый человек из группы, принимающий участие в опросе, может повторно выразить то же мнение, если ему снова будет задан вопрос, поскольку участники опроса выражают лишь свое личное убеждение.

При описании нечетких множеств в качестве функции принадлежности часто используется математическая функция, называемая **S-функцией**. Эта функция

определяется следующим образом:

$$S(x;\alpha,\beta,\gamma)=\begin{cases}0 & \text{для } x\leqslant\alpha;\\ 2\left(\dfrac{x-\alpha}{\gamma-\alpha}\right)^2 & \text{для } \alpha\leqslant x\leqslant\beta;\\ 1-2\left(\dfrac{x-\gamma}{\gamma-\alpha}\right)^2 & \text{для } \beta\leqslant x\leqslant\gamma;\\ 1 & \text{для } x\geqslant\gamma.\end{cases}$$

График S-функции показан на рис. 5.9.

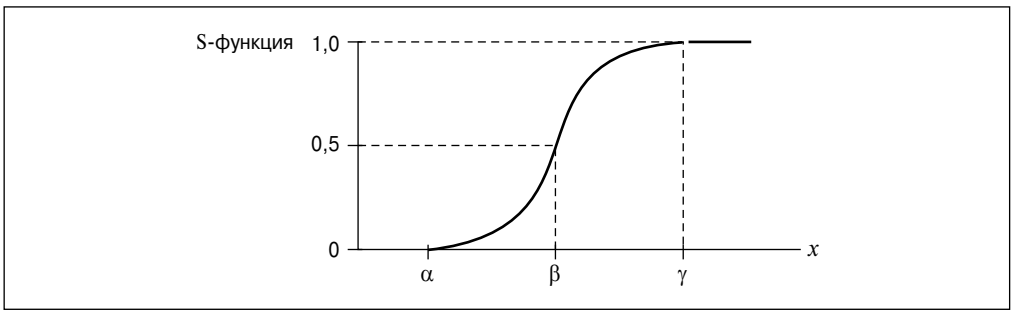


Рис. 5.9. S-функция

S-функция показала себя ценным инструментальным средством при определении таких нечетких функций, как в случае со словом “высокий”. Такая функция позволяет отказаться от сопровождения таблицы с данными, определяющими функцию принадлежности, и компактно представить все данные с помощью одной формулы. В приведенном выше определении символы α, β и γ представляют собой параметры, которые могут быть откорректированы для достижения соответствия с данными, описывающими принадлежность к множеству. В зависимости от имеющихся данных о принадлежности может существовать возможность достичь точного соответствия при некоторых значениях α, β и γ или добиться лишь приближенного соответствия. Безусловно, функция принадлежности может быть определена исключительно как S-функция, без ссылки на какие-либо табличные данные. В зависимости от приложения могут быть также определены другие функции, например, имеющие график треугольной формы.

График S-функции становится плоским, принимая значение 0 при $x=\alpha$ и значение 1 — при $x=\gamma$. В интервале от α до γ та же S-функция принимает вид квадратичной функции от x. Как показано на рис. 5.9, параметр β соответствует точке пересечения 0.5 и имеет значение $(\alpha+\gamma)/2$. Применительно к функции принадлежности нечеткого множества `TALL`, показанной на рис. 5.8, S-функция

представляет собой следующее:

$$(1) \qquad S(x;5,6,7) = \begin{cases} 0 & \text{для } x \leqslant 5; \\ 2\left(\dfrac{x-5}{7-5}\right)^2 = \dfrac{(x-5)^2}{2} & \text{для } 5 \leqslant x \leqslant 6; \\ 1-2\left(\dfrac{x-7}{7-5}\right)^2 = 1-\dfrac{(x-7)^2}{2} & \text{для } 6 \leqslant x \leqslant 7; \\ 1 & \text{для } x \geqslant 7. \end{cases}$$

На рис. 5.10 показана функция принадлежности для нечеткого утверждения "X приближается к γ". Например, эта функция принадлежности позволяет представить все числа, приближающиеся к указанному значению γ, что выражается, например, с помощью такой формулировки, как "X примерно равно 6", где универсальное множество X может быть определено как $\{5.9, 6, 6.1\}$. В таком случае функцию принадлежности можно выразить следующим образом:

$$\mu_{\text{CLOSE}}(x) = \frac{1}{1+\left(\frac{x-\gamma}{\beta}\right)^2}$$

с такими точками пересечения:

$$x = \gamma \pm \beta$$

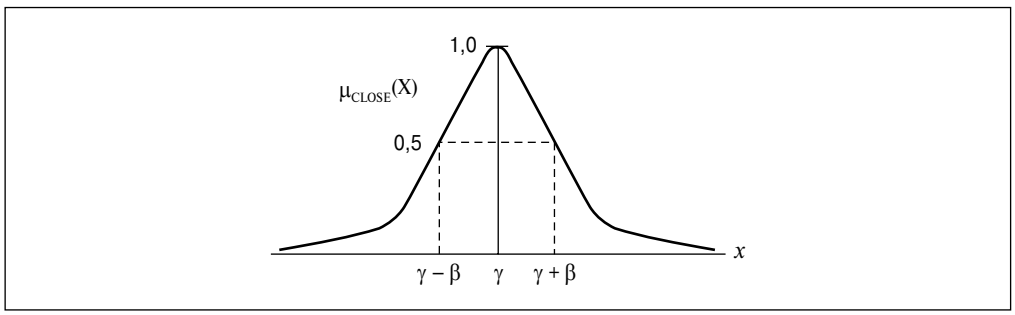


Рис. 5.10. Функция принадлежности для нечеткого утверждения "x приближается к γ"

Параметр β называют **полушириной** кривой в точке пересечения, как показано на рис. 5.10. Бóльшие значения β соответствуют более широкой кривой, а меньшие значения соответствуют более узкой кривой. Увеличение значения β говорит о том, что числа должны быть более близкими к γ, чтобы приобрести значительно более высокое значение функции принадлежности. Обратите внимание на то, что при таком определении функции принадлежности нулевое значение достигается только на бесконечно большом удалении.

Ниже приведена функция, которая также позволяет получить подобную кривую, но достигает нуля в указанных точках.

$$\prod(x;\beta,\gamma) = \begin{cases} S(x;\gamma-\beta,\gamma-\beta/2,\gamma) & \text{для } x \leqslant \gamma \\ 1-S(x;\gamma,\gamma+\beta/2,\gamma+\beta) & \text{для } x \geqslant \gamma \end{cases}$$

График такой $\prod$**-функции** приведен на рис. 5.11. Следует отметить, что теперь параметр β в точках пересечения определяет **ширину интервала**, или **общую ширину**. $\prod$-функция достигает нуля в следующих точках:

$$x = \gamma \pm \beta$$

В этом случае координаты точек пересечения определяются таким выражением:

$$x = \gamma \pm \frac{\beta}{2}$$

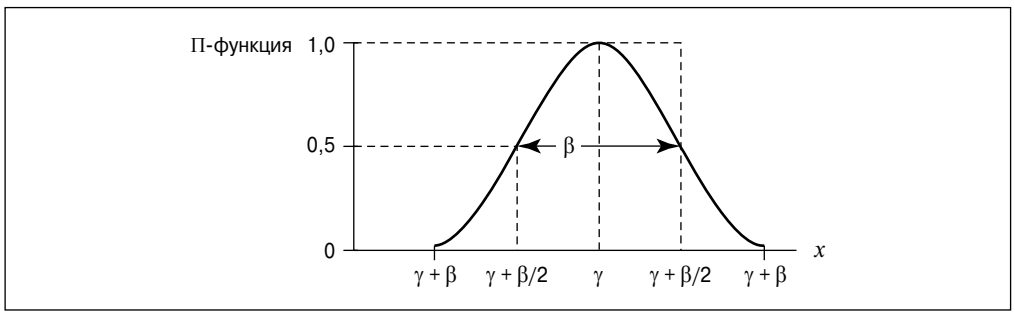


Рис. 5.11. График $\prod$-функции

Функция принадлежности не только может выражаться непрерывной функцией, но и представлять собой конечное множество элементов. Например, предположим, что универсум значений роста определен следующим образом:

$$U = \{5, 5.5, 6, 6.5, 7, 7.5, 8\}$$

В таком случае нечеткое подмножество можно определить как конечное множество элементов множества TALL следующим образом:

$$\text{TALL} = \{0/5, 0.125/5.5, 0.5/6, 0.875/6.5, 1/7, 1/7.5, 1/8\}$$

В этом нечетком множестве символ / отделяет степени принадлежности от чисел, соответствующих значениям роста. Следует отметить, что в общепринятой системе обозначений нечетких множеств символ / не означает деление. Элементы нечеткого множества, для которых $\mu(x) > 0$, формируют **обоснование** нечеткого

множества. Для множества TALL обоснованием являются все элементы, кроме $0/5$.

Конечное нечеткое подмножество из N элементов может быть представлено в стандартной системе обозначений нечетких множеств как объединение нечетких одноэлементных множеств μ_i/x_i, где знаки $+$ представляют собой знаки операции булева объединения:

(2)
$$F = \mu_1/x_1 + \mu_2/x_2 + \ldots \mu_N/x_N$$
$$F = \sum_{i=1}^{N} \mu_i/x_i$$
$$F = \bigcup_{i=1}^{N} \mu_i/x_i$$

В некоторых статьях символ / не записывается, и поэтому подмножество F определяется в виде следующих формул:

(3)
$$F = \mu_1 x_1 + \mu_2 x_2 + \ldots \mu_N x_N$$
$$F = \sum_{i=1}^{N} \mu_i/x_i$$
$$F = \bigcup_{i=1}^{N} \mu_i/x_i$$

Оба этих уравнения, (2) и (3), представляют конечные нечеткие подмножества из N элементов. Уравнение (3) является удобным для записи нечетких множеств в компактной символической форме. Но если применяются только числа, то формы, предусмотренные в уравнении (3), становятся сложными для чтения людьми, как в следующем примере:

$$F = .127 + .385$$

Без разделителя / невозможно понять, имеют ли степени принадлежности значения .1 и .3, или .12 и .38, или какие-то другие значения. Именно поэтому для работы с числами удобнее система обозначений с разделителями, как показано ниже.

$$F = .1/27 + .38/5$$

Более формально, обоснование нечеткого множества F можно определить как подмножество универсального множества X следующим образом:

$$\text{support}(F) = \{x \mid x \in X \text{ и } \mu_F(x) > 0\}$$

Для обозначения обосновывающего множества обычно для краткости используется сокращение от слова support, **supp**, как в следующем примере:

$$\text{supp}(F)$$

Применение обоснования имеет преимущество в том, что нечеткое множество F может быть записано таким образом:

$$F = \{\mu_F(x)/x \mid x \in \text{supp}(F)\}$$

Это выражение означает, что свой вклад в формирование множества F вносят только те нечеткие элементы, для которых функция принадлежности больше нуля. Таким образом, множество TALL может быть записано без элемента $0/5$ следующим образом, поскольку этот элемент не принадлежит к обосновывающему множеству:

$$\text{TALL} = \{0.125/5.5, 0.5/6, 0.875/6.5, 1/7, 1/7.5, 1/8\}$$

Безусловно, в данном случае достигается лишь незначительное сокращение, соответствующее одному элементу, но применительно к нечетким множествам с большим количеством элементов, имеющих нулевую степень принадлежности, сокращение количества элементов может оказаться весьма значительным. С другой стороны, при решении какой-то другой задачи может представлять интерес и информация о том, какие элементы имеют нулевую степень принадлежности.

С понятием обоснования связано понятие α**-отсечения**; α*-отсечением* множества называется множество, отличное от нечеткого подмножества универсального множества, элементы которого имеют значение функции принадлежности, большее или равное некоторому значению α:

$$F_\alpha = \{x \mid \mu_F(x) = \alpha\} \quad \text{для } 0 < \alpha = 1$$

Ниже приведены некоторые примеры α-отсечений для множества TALL.

$$\begin{aligned}
\text{TALL}_{0.1} &= \{5.5, 6, 6.5, 7, 7.5, 8\} \\
\text{TALL}_{0.5} &= \{6, 6.5, 7, 7.5, 8\} \\
\text{TALL}_{0.8} &= \{6.5, 7, 7.5, 8\} \\
\text{TALL}_{1} &= \{7, 7.5, 8\}
\end{aligned}$$

Обратите внимание на то, что все α-отсечения некоторого множества являются подмножествами обоснования. Значения α могут выбираться произвольно, но, как правило, их подбор осуществляется для выделения желаемых подмножеств универсума.

Еще одним термином, часто применяемым с нечеткими множествами, является **высота**; это понятие определяет максимальную степень принадлежности элемента. Для рассматриваемого множества TALL максимальная степень принадлежности равна 1. Если в нечетком множестве имеется хотя бы один элемент, приобретающий максимально возможную степень, такое множество называется **нормализованным**. Обычно степени принадлежности определяются в замкнутом интервале $[0, 1]$, и поэтому максимально возможная степень принадлежности равна 1. Но степени могут определяться в других интервалах, и поэтому степень принадлежности не обязательно должна быть равна 1.

Произвольное нечеткое подмножество универсума, заданного как **континуум**, записывается в форме интеграла. Термином "континуум" обозначается множество вещественных чисел. Интеграл представляет собой объединение нечетких одноэлементных множеств, $\mu(x)/x$. Например, можно определить следующее соотношение, используя для множества TALL такую S-функцию:

$$\begin{aligned}\text{TALL} &= \int_x \mu_{\text{TALL}}(x)/x = \\ &= \int_5^8 \mu_{\text{TALL}}(x)/x = \\ &= \int_5^6 \frac{(x-5)^2}{2}/x + \int_6^7 \left[1 - \frac{(x-7)^2}{2}\right]/x + \int_7^8 1/x\end{aligned}$$

В этой формуле знаки $+$, разделяющие интегралы, представляют собой знаки операции объединения (как при использовании системы обозначений в булевой логике), а не знаки арифметического сложения.

Нечеткие множества подразделяются на несколько типов. Элементарное **нечеткое подмножество первого типа**, F, универсума X определяется следующим образом:

$$\mu_F : X \to [0, 1]$$

Иными словами, нечеткое подмножество первого типа определяется путем присваивания числовых значений его функции принадлежности в замкнутом интервале вещественных чисел от 0 до 1. Например, следующее множество является множеством первого типа, поскольку областью определения его степеней принадлежности являются все вещественные числа в интервале $[0, 1]$.

$$\text{TALL} = .125/5.5 + .5/6 + .875/6.5 + +1/7 + 1/7.5 + 1/8$$

Аналогичным образом, согласно уравнению (1), следующее множество является нечетким подмножеством первого типа:

$$\mu_{\text{TALL}}(x) = S(x; 5, 6, 7)$$

Вообще говоря, **нечеткое подмножество типа** N определяется путем создания для μ_F отображения универсума на множество нечетких подмножеств типа $N-1$. Например, **нечеткое подмножество второго типа** определяется в терминах подмножества первого типа. Что касается данных о росте, то нечетким множеством второго типа может быть следующее множество, в котором все элементы LESS THAN AVERAGE, AVERAGE и GREATER THAN AVERAGE представляют собой нечеткие подмножества первого типа:

$$\mu_{\text{TALL}}(5) = \text{LESS THAN AVERAGE}$$
$$\mu_{\text{TALL}}(6) = \text{AVERAGE}$$
$$\mu_{\text{TALL}}(7) = \text{GREATER THAN AVERAGE}$$

Эти множества могут быть определены как такие нечеткие подмножества:

$$\mu_{\text{LESS THAN AVERAGE}}(x) = 1 - S(x; 4.5, 5, 5.5)$$
$$\mu_{\text{AVERAGE}}(x) = \prod(x; 1, 5.5)$$
$$\mu_{\text{GREATER THAN AVERAGE}}(x) = S(x; 5.5, 6, 6.5)$$

Операции с нечеткими множествами

Обычное четкое множество представляет собой частный случай нечеткого множества с функцией принадлежности $\{0, 1\}$. В пределе к четким множествам должны быть применимыми все определения, доказательства и теоремы нечетких множеств, после того как нечеткость приближается к нулю и нечеткое множество становится четким множеством. Таким образом, теория нечетких множеств имеет более широкий спектр приложений, чем теория четких множеств, в частности, позволяет справляться с ситуациями, в которых должны учитываться субъективные мнения. Основной замысел, лежащий в основе теории нечетких множеств, состоит в том, что эта теория позволяет определять такие нечеткие концепции реального мира, как рост, с помощью множества нечетких элементов (TALL), не требуя определения резкой бинарной границы. Ниже приведены общие сведения о некоторых операциях с нечеткими множествами в универсуме X.

- Равенство множеств

$$A = B$$
$$\mu_A(x) = \mu_B(x) \quad \text{для всех } x \in X.$$

- Дополнение множества

$$A'$$
$$\mu_{A'}(x) = 1 - \mu_A(x) \quad \text{для всех } x \in X.$$

На рис. 5.12 показано нечеткое дополнение множества. Это определение дополнения обосновано Беллманом (Bellman) и Гирцем (Giertz).

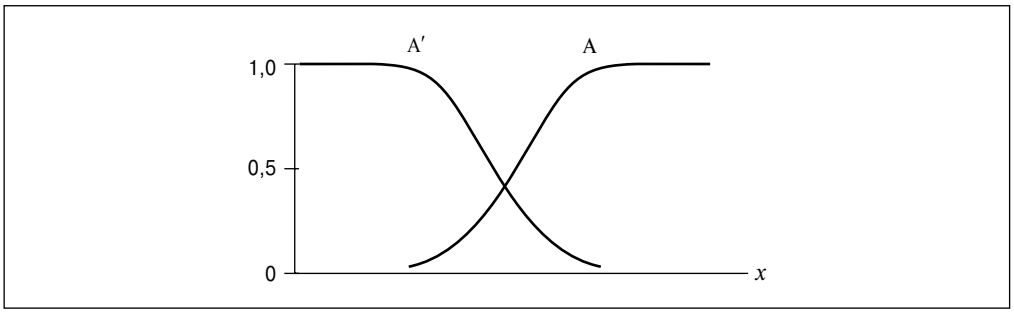


Рис. 5.12. Нечеткое дополнение

- Включение множества

$$A \subseteq B$$

Нечеткое множество A содержится в множестве B, или является **подмножеством** множества B тогда и только тогда, когда выполняется следующее соотношение:

$$\mu_A(x) \leqslant \mu_B(x) \quad \text{для всех } x \in X$$

Нечеткое множество A является **строгим подмножеством** множества B, если A — подмножество B и эти два множества не равны, как показано ниже.

$$\mu_A(x) \leqslant \mu_B(x) \text{ и } \mu_A(x) < \mu_B(x) \quad \text{по меньшей мере для одного } x \in X$$

- Объединение множеств

$$A \cup B$$
$$\mu_A \cup_B (x) = \vee(\mu_A(x), \mu_B(x)) \quad \text{для всех } x \in X.$$

В этой формуле **операция соединения**, $\vee$, обозначает операцию определения максимального параметра.

- Пересечение множеств

$$A \cap B$$
$$\mu_{A \cap B}(x) = \wedge(\mu_A(x), \mu_B(x)) \quad \text{для всех } x \in X.$$

В этой формуле **операция соприкосновения**, $\wedge$, обозначает операцию определения минимального параметра.

В подразделе "Комбинирование свидетельств с использованием нечеткой логики" раздела 4.14 уже упоминались операции соединения (join) и соприкосновения (meet) при обсуждении способов использования нечеткой логики для комбинирования антецедентов в системе PROSPECTOR. А обоснование возможности применения функций $\max$ и $\min$ вместо операций соединения и соприкосновения было предложено Беллманом.

Использование этих определений позволяет распространить на нечеткие множества основные законы четких множеств, такие как коммутативность, ассоциативность и т.д., за исключением закона исключенного третьего и закона противоречия. Таким образом, для нечеткого множества A можно сформулировать следующие утверждение, в которых $\varnothing$ обозначает пустое множество, а $\mathcal{U}$ — универсум:

$$A \cup A' \neq \mathcal{U} \text{ и } A \cap A' \neq \varnothing$$

Поскольку нечеткие множества могут не иметь резкой границы, то при условии определения $\mu(x)$ в замкнутой области определения $[0, 1]$ и $\mu_{A'}(x) = 1 - \mu_A(x)$ единственным ограничением, налагаемым на пересечение, становится следующее:

$$A \cap A' = \min(\mu_A(x), \mu'_A(x)) = 0.5$$

Аналогичным образом, единственное ограничение на объединение таково:

$$A \cup A' = \max(\mu_A(x), \mu'_A(x)) = 0.5$$

Нечеткие множества A и A' могут пересекаться, поэтому может оказаться, что эти два множества не охватывают универсум полностью.

Если будут определены закон исключенного третьего и закон противоречия, которые соблюдались бы для нечетких множеств, то может оказаться, что не соблюдаются закон идемпотентности и распределительный закон. Формулировка закона идемпотентности для нечетких множеств состоит в следующем:

$$A \cup A \neq A \quad A \cap A \neq A$$

А поскольку в данном случае множества имеют нечеткий характер, кажется более обоснованным принятие предположения, что для нечетких множеств не соблюдаются закон исключенного третьего и закон противоречия, чтобы для них оставался справедливым закон идемпотентности. Описание полезных свойств нечетких множеств приведено в приложении В.

- Произведение множеств

$$A\ B$$
$$\mu_{AB}(x) = \mu_A(x)\mu_B(x).$$

- Возведение множества в степень

$$A^N$$
$$\mu_{AN}(x) = (\mu_A(x))^N$$

- Вероятностная сумма

$$A \hat{+} B$$
$$\mu_{A\hat{+}B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x) = 1 - (1 - \mu_A(x))(1 - \mu_B(x))$$

В последней формуле знаки $+$ и $-$ обозначают обычные арифметические операции.

- Ограниченная сумма или резко выраженное объединение

$$A \oplus B$$
$$\mu_{A\oplus B}(x) = \wedge(1, (\mu_A(x) + \mu_B(x)))$$

В последней формуле знак $\wedge$ обозначает функцию определения минимума, а знак $+$ соответствует обычной арифметической операции.

- Ограниченное произведение или резко выраженное пересечение

$$A \odot B$$
$$\mu_{A\odot B}(x) = \vee(0, (\mu_A(x) + \mu_B(x) - 1))$$

В последней формуле знак $\vee$ обозначает функцию определения максимума, а знак $+$ соответствует обычной арифметической операции. Операции ограниченной суммы и произведения не удовлетворяют законам идемпотентности, дистрибутивности и поглощения, но удовлетворяют законам коммутативности, ассоциативности, де Моргана, соотношениям $A \oplus \mathcal{U} = \mathcal{U}$, $A \odot \varnothing = \varnothing$, закону исключенного третьего и закону противоречия (общие сведения см. в приложении В).

- Ограниченная разность

$$A \mid - \mid B$$
$$\mu_{A|-|B}(x) = \vee(0, (\mu_A(x) - \mu_B(x)))$$

В последней формуле знак $-$, разделяющий μ_A и μ_B, представляет собой знак арифметической операции вычитания. Выражение $A \mid - \mid B$ представляет те элементы, которые в большей степени содержатся в A, чем в B. Отметим, что в терминах универсального множества, $\mathcal{U}$, и ограниченной разности операцию дополнения можно определить следующим образом:

$$A' = \mathcal{U} \mid - \mid A$$

- Концентрация

$$\text{CON}(A)$$
$$\mu_{\text{CON}(A)}(x) = (\mu_A(x))^2$$

Операция CON "концентрирует" элементы нечеткого множества, уменьшая значения степени принадлежности элементов тем больше, чем меньше эти значения. Пример применения операции CON приведен на рис. 5.13. Эта операция и описанные ниже операции DIL, NORM и INT не имеют аналогов среди обычных операций с множествами. Операция CON может использоваться для грубой аппроксимации результата применения лингвистического модификатора Very (Очень). Таким образом, для некоторого нечеткого множества F справедлива такая формула:

$$\text{Very}F = F^2$$

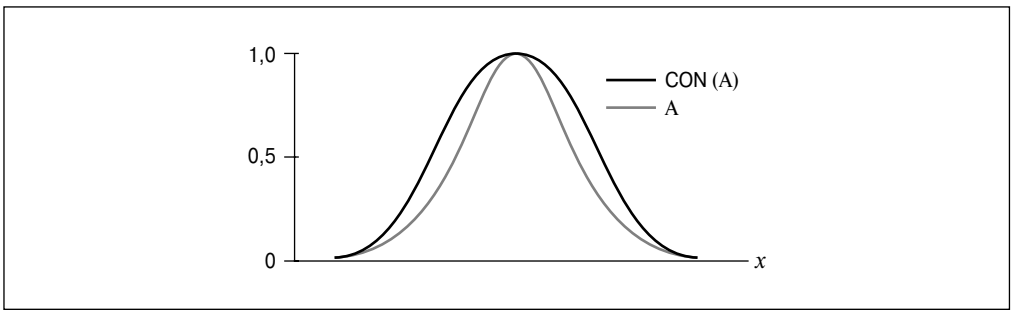


Рис. 5.13. Пример применения операции концентрации к нечеткому множеству

Рассмотрим в качестве примера следующее множество TALL:

$$\text{TALL} = .125/5 + .5/6 + .875/6.5 + 1/7 + 1/7.5 + 1/8$$

В таком случае при использовании двух значащих цифр получаем:

$$\text{Very TALL} = .016/5 + .25/6 + .76/6.5 + 1/7 + 1/7.5 + 1/8$$

Обратите внимание на то, как уменьшились значения степени принадлежности для всех элементов, кроме тех, что имеют степень принадлежности 1. Общий эффект состоит в том, что нечеткое множество Very TALL включает меньше элементов с малыми значениями степени принадлежности.

- Растворение

$$\text{DIL}(A)$$
$$\mu_{\text{DIL}(A)}(x) = (\mu_A(x))^{0.5}$$

Операция DIL "растворяет" нечеткие элементы, увеличивая значения степени принадлежности элементов тем больше, чем меньше эти значения. Пример применения операции DIL показан на рис. 5.14. Обратите внимание на то, что операция растворения противоположна операции концентрации в том, что первой соответствует показатель степени, равный 0.5, а второй — равный 2:

$$A = \text{DIL}(\text{CON}(A)) = \text{CON}(\text{DIL}(A))$$

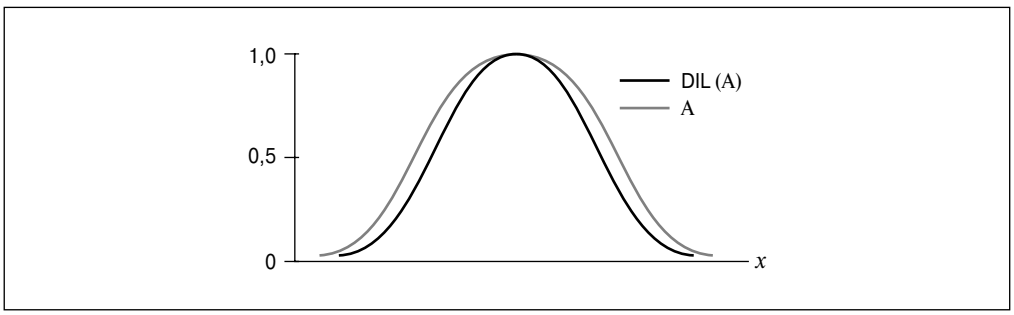


Рис. 5.14. Пример применения операции растворения к нечеткому множеству

Операция растворения грубо аппроксимируется лингвистическим модификатором More Or Less (Более или менее). Таким образом, для любого нечеткого множества F соблюдаются следующие зависимости:

$$\text{More Or Less } F = F^{0.5} = DIL(F)$$

- Интенсификация

$$\text{INT}(A)$$
$$\mu_{\text{INT}(A)}(x) = \begin{cases} 2(\mu_A(x))^2 & \text{для } 0 \leqslant \mu_A(x) \leqslant 0.5 \\ 1 - 2(1 - \mu_A(x))^2 & \text{для } 0.5 \leqslant \mu_A(x) \leqslant 1 \end{cases}$$

Операция INT напоминает операцию интенсификации (повышения контрастности) изображения. Как показано на рис. 5.15, в результате интенсификации повышается степень принадлежности элементов, находящихся между точками пересечения, и снижается степень принадлежности элементов, находящихся за пределами точек пересечения. Рассматривая в качестве

аналогии электронный усилитель, можно представить себе точки пересечения как определяющие полосу частот сигнала. В результате интенсификации сигнал в пределах полосы частот усиливается, а “шум”, выходящий за пределы полосы частот, ослабляется. Таким образом, интенсификация увеличивает различие между степенями принадлежности элементов, находящихся между точками пересечения, по сравнению с элементами, выходящими за пределы точек пересечения.

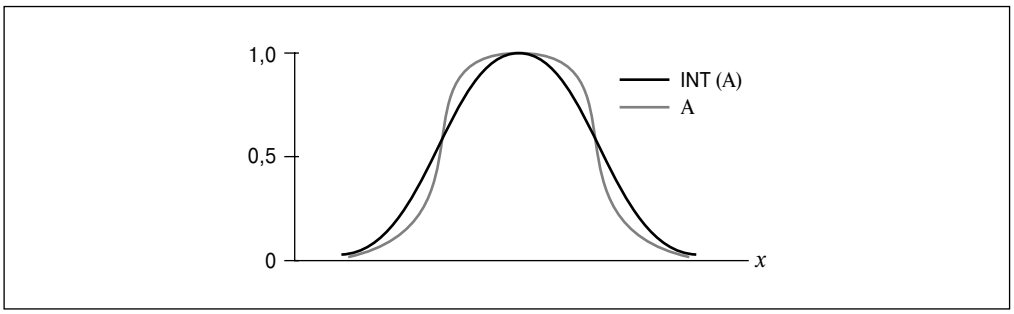


Рис. 5.15. Пример применения операции интенсификации к нечеткому множеству

- Нормализация

$$\text{NORM}(A)$$
$$\mu_{\text{NORM}(A)}(x) = \mu_A(x) / \max\{\mu_A(x)\}$$

В последней формуле функция $\max$ возвращает максимальное значение степени принадлежности для всех элементов x. Если максимальная степень принадлежности меньше 1, то повышаются все значения степени принадлежности. Если максимальная степень принадлежности равна 1, то значения степени принадлежности остаются неизменными.

Нечеткие отношения

Нечеткие множества позволяют моделировать еще одну важную концепцию — **отношение**. Интуитивное представление об отношении состоит в том, что отношение — это определенная ассоциация между элементами. Ниже приведены некоторые примеры отношений, в которых слова, выделенные курсивом, обозначают нечеткие понятия.

Боб и Эллис – *друзья*
Лос-Анджелес и Нью-Йорк – *очень далеко друг от друга*
1, 2, 3 и 4 – *гораздо меньше* чем 100
1, 2 и 3 – *малые* числа
яблоки и апельсины – фрукты *круглой формы*

Декартово произведение N четких множеств определено как четкое множество, элементами которого являются N-элементные кортежи $(x_1, x_2, x_3, \ldots x_N)$, где каждый элемент x_i является элементом своего четкого множества X_i. Для двух множеств A и B:

$$A \times B = \{(a, b) \mid a \in A \text{ и } b \in B\}$$

Рассмотрим следующее определение множества:

$$A = \{\text{chocolate}, \text{strawberry}\}$$
$$B = \{\text{pie}, \text{milk}, \text{candy}\}$$

В таком случае декартово произведение принимает вид

$$\begin{aligned} A \times B = \{&(\text{chocolate}, \text{pie}), (\text{chocolate}, \text{milk}), \\ &(\text{chocolate}, \text{candy}), (\text{strawberry}, \text{pie}), \\ &(\text{strawberry}, \text{milk}), (\text{strawberry}, \text{candy})\} \end{aligned}$$

Обратите внимание на то, что в общем произведение $B \times A$ не равно произведению $A \times B$, если множества A и B содержат разные элементы. Это означает, что в общем $(a, b) \neq (b, a)$. Говорят, что произведение $A \times B$ определяет **бинарную переменную** (a, b).

Отношение R представляет собой подмножество декартового произведения. Например, отношение $R =$ LIKES PIE (Любит торты) может быть определено как подмножество произведения $A \times B$ следующим образом:

$$R = \{(\text{chocolate}, \text{pie}), (\text{strawberry}, \text{pie})\}$$

А отношение $R =$ LIKES SWEETS (Любит сладости) может быть определено так:

$$\begin{aligned} R = \{&(\text{chocolate}, \text{pie}), (\text{chocolate}, \text{candy}), \\ &(\text{strawberry}, \text{pie}), (\text{strawberry}, \text{candy})\} \end{aligned}$$

Отношение иногда называют также **отображением**, поскольку оно связывает элементы из одной области определения с элементами из другой области определения. Если дано произведение $A \times B$, то отношение — это отображение из $A \rightarrow B$, где $\rightarrow$ в данном контексте обозначает операцию отображения. Элемент chocolate (шоколад), который определен в множестве A, отображается, или ассоциируется с элементами pie (торт) и candy (конфета) в множестве B, и аналогичная ситуация создается для элемента strawberry (земляника).

Если X и Y — универсальные множества, то следующее выражение определяет нечеткое отношение на $X \times Y$:

$$R = \{\mu_R(x, y)/(x, y) \mid (x, y) \subseteq X \times Y\}$$

По существу, нечеткое отношение представляет собой нечеткое подмножество в универсуме декартова произведения. Другое определение нечетких множеств в большей степени подходит для работы с нечеткими графами.

Нечеткое отношение для N множеств определяется как расширение четкого отношения, которое включает обозначения степеней принадлежности. Таким образом, имеет место следующее соотношение, которое определяет ассоциацию со степенью принадлежности каждого N-элементного кортежа:

$$R = \{\mu_R(x_1, x_2, \ldots x_N)/(x_1, x_2, \ldots x_N) \mid x_i \in X_i, i = 1, ..N\}$$

Для бинарного отношения, применяемого в примере с тортом, в качестве нечеткого отношения можно использовать другое определение, позволяющее показать, что некоторое лицо гораздо больше любит шоколадный торт, чем земляничный:

$$R = \{.9/(\text{chocolate}, \text{pie}), .2/(\text{strawberry}, \text{pie})\}$$

Удобный способ представления отношения состоит в использовании матрицы. Например, для отношения LIKES SWEETS может быть определено следующее соотношение, в котором M_R обозначает матричное представление нечеткого отношения:

$$M_R = \begin{array}{c} \\ \text{chocolate} \\ \text{strawberry} \end{array} \begin{array}{c} \begin{array}{cc} \text{pie} & \text{candy} \end{array} \\ \begin{bmatrix} 0.9 & 0.7 \\ 0.2 & 0.1 \end{bmatrix} \end{array}$$

Следует отметить, что для четких множеств матрица M_R состояла бы только из нулей и единиц, поэтому позволяла бы лишь представить такую ситуацию, в которой человек либо очень любит, либо совсем не любит определенное сочетание начинки и кондитерского изделия, которое представлено в виде упорядоченной пары $(\text{flavor}, \text{sweet})$. (По-видимому, этот пример является лучшим доказательством того, что наши представления о мире фактически легче выразить с помощью нечетких, а не четких формулировок.)

Композиция отношений представляет собой чистый результат применения одного отношения после другого. Для случая двух бинарных отношений P и Q композиция отношений представляет собой следующее бинарное отношение R:

$$R(A, C) = Q(A, B) \circ P(B, C)$$

В этой формуле применяются следующие обозначения, а знак $\circ$ служит для обозначения **операции композиции**:

$R(A, C)$ — отношение между A и C;

$Q(A, B)$ — отношение между A и B;

$P(B, C)$ — отношение между B и C;

A, B и C — множества.

Отношение R выражает результат, полученный после применения вначале отношения P, а затем Q. В терминах степеней принадлежности эту мысль можно выразить следующим образом:

$$R = \{\mu_R(a, c)/(a, c) \mid a \in A, c \in C\}$$

В этой формуле значение μ_R определено так:

$$\begin{aligned}\mu_R(A, C) &= \bigvee_{b \in B}[\mu_Q(A, B) \wedge \mu_P(B, C)] \\ &= \max_{b \in B}[\min(\mu_Q(A, B), \mu_P(B, C))]\end{aligned}$$

Указанную композицию принято называть **матричным произведением max-min**, или просто **max-min**. При выполнении операций с матрицами функции $\max$ и $\min$ могут использоваться вместо операций сложения и умножения. В качестве примера определим следующие отношения:

$$Q = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \quad P = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.0 & 0.4 \end{bmatrix}$$

В таком случае композиция R этих отношений будет выражаться таким образом:

$$R = Q \circ P = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \circ \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.0 & 0.4 \end{bmatrix}$$

$$R = Q \circ P = \begin{bmatrix} \max(0.1, 0.2) & \max(0.1, 0.0) & \max(0.1, 0.2) \\ \max(0.1, 0.2) & \max(0.3, 0.0) & \max(0.3, 0.4) \end{bmatrix} =$$

$$= \begin{bmatrix} 0.2 & 0.1 & 0.2 \\ 0.2 & 0.3 & 0.4 \end{bmatrix}$$

К числу других широко применяемых определений реляционных операций относятся **max-произведение** и **реляционное соединение**.

Как будет показано ниже, нечеткие отношения имеют важные приложения в области приближенных рассуждений. Еще одним важным понятием теории нечетких множеств является **проекция** отношения. По существу, проекция позволяет удалить указанные элементы. Прежде чем перейти к рассмотрению формальных определений, ознакомимся с простым примером, приведенным в табл. 5.10.

Таблица 5.10. Отношение и его проекции

	y_1	y_2	y_3	**Первая проекция**	
x_1	0.2	0.1	0.2	0.2	
x_2	0.2	0.3	0.4	0.4	
Вторая проекция	0.2	0.3	0.4	0.4	Общая проекция

В табл. 5.10 показаны проекции приведенного выше отношения для R. В этой таблице для удобства строкам и столбцам были присвоены идентификаторы x_i и y_i. Обратите внимание на то, что столбец, обозначенный как первая проекция, содержит значение максимальной степени принадлежности для данной строки. Аналогичным образом, строка, обозначенная как вторая проекция, содержит значение максимальной степени принадлежности для каждого столбца. Ячейка со значением 0.4 в нижнем правом углу, обозначенная как общая проекция, представляет собой максимальную степень принадлежности для всего отношения.

Отношение для R может быть записано в терминах x_i и y_i следующим образом:

$$R = .2/x_1, y_1 + .1/x_1, y_2 + .2/x_1, y_3 + .2/x_2, y_1 + .3/x_2, y_2 + .4/x_2, y_3$$

Первая проекция обозначается как R_1; эта проекция может быть получена путем удаления всех элементов, кроме первой точки x_i декартовой пары x_i, y_j, в которой крайний левый элемент определен как первый:

$$R_1 = .2/x_1 + .1/x_1 + .2/x_1 + .2/x_2 + .3/x_2 + .4/x_2$$

После получения этой проекции уравнение для R_1 еще больше сокращается, поскольку знак $+$ соответствует знаку операции объединения и поэтому сохраняется только максимальный нечеткий элемент:

$$R_1 = .2/x_1 + .4/x_2$$

Аналогичным образом, во второй проекции сохраняется только вторая точка каждой декартовой пары, y_j:

$$R_2 = .2/y_1 + .1/y_2 + .2/y_3 + .2/y_1 + .3/y_2 + .4/y_3$$

Это соотношение в результате применения операции объединения сводится к следующему:

$$R_2 = .2/y_1 + .3/y_2 + .4/y_3$$

В общем случае, когда в отношении участвуют N декартовых точек, необходимо удалить все компоненты N-элементного кортежа, кроме тех, по которым

должна быть выполнена проекция. Например, если отношение определено на N точках, то операция R_{136} удаляет все точки, кроме первой, третьей и шестой.

Для отношения, заданного на универсальном множестве $X \times Y$,

$$R = \{\mu_R(x,y)/(x,y)\} \text{ для всех } (x,y) \in X \times Y$$

первая проекция определяется по следующей формуле:

$$\text{proj}(R;X) = R_1$$

в которой

$$R_1 = \{\max_y \mu_R(x,y)/x \mid (x,y) \in X \times Y\}$$

и значение $\max$ определяется по всем точкам y. Аналогичным образом, имеет место формула

$$\text{proj}(R;Y) = R_2$$

в которой

$$R_2 = \{\max_x \mu_R(x,y)/y \mid (x,y) \in X \times Y\}$$

и значение $\max$ определяется по всем значениям x для отношения R_2.

Цилиндрическое расширение отношения проекции определено как наибольшее нечеткое отношение, совместимое с проекцией. *Цилиндрическая проекция* в определенной мере аналогична обычной проекции, поскольку распространяет проектируемое значение (которое является максимальным) на все другие элементы, например, как показано ниже.

$$\text{proj}(R;X) = R_1 = .2/x_1 + .4/x_2$$

Поэтому имеет место следующее:

$$\begin{aligned} R_1 = {} & .2/x_1, y_1 + .2/x_1, y_2 + .2/x_1, y_3 + \\ & + .4/x_2, y_1 + .4/x_2, y_2 + .4/x_2, y_3 \\ R_2 = {} & .2/y_1, x_1 + .2/y_1, x_2 + .3/y_2, x_1 + \\ & + .3/y_2, x_1 + .3/y_2, x_2 + \\ & + .4/y_2, x_1 + .4/y_3, x_2 \end{aligned}$$

Это означает, что происходит замена второй переменной, $.2/x_1, \{\text{все_переменные}\} + .4/x_2, \{\text{все_переменные}\}$. Операция проекции позволяет получить значение $\max \mu$, поэтому цилиндрическое расширение представляет собой наибольшее отношение, совместимое с проекцией.

На основании предыдущего примера может быть получено следующее соотношение, в котором вертикальная черта над проекцией символически обозначает цилиндрическое расширение проекции:

$$\overline{R_1} = \begin{bmatrix} 0.2 & 0.2 & 0.2 \\ 0.4 & 0.4 & 0.4 \end{bmatrix}$$

$$\overline{R_2} = \begin{bmatrix} 0.2 & 0.3 & 0.4 \\ 0.2 & 0.3 & 0.4 \end{bmatrix}$$

Композиция может быть определена в терминах операций проекции и цилиндрического расширения. Для бинарного отношения R, определенного на универсальном множестве $\mathcal{U}_1 \times \mathcal{U}_2$, и отношения S, определенного на $\mathcal{U}_2 \times \mathcal{U}_3$, композиция выражается следующей формулой:

$$R \circ S = \text{proj}(\overline{R} \cap \overline{S}); (\mathcal{U}_1 \times \mathcal{U}_3)$$

Лингвистические переменные

Одной из очень важных областей применения нечетких множеств является **компьютерная лингвистика**. Перед этой наукой поставлена цель — обеспечить проведение вычислений над предложениями естественного языка таким же образом, как логика обеспечивает обработку логических высказываний. Для количественной оценки смысла предложений естественного языка применяются нечеткие множества и **лингвистические переменные**, после чего появляется возможность манипулировать этими предложениями. Лингвистическим переменным присваиваются значения, представляющие собой такие выражения, как слова, фразы или предложения естественного или искусственного языка. В табл. 5.11 показаны некоторые лингвистические переменные и типичные значения, которые могут быть им присвоены.

Таблица 5.11. Лингвистические переменные и типичные значения

Лингвистическая переменная	Типичные значения
Рост	Карликовый, небольшой, средний, высокий, гигантский
Количество предметов	Почти отсутствуют, несколько, немного, много
Этапы развития человека	Младенец, малыш, ребенок, подросток, взрослый
Цвет	Красный, синий, зеленый, желтый, оранжевый
Свет	Тусклый, слабый, нормальный, яркий, интенсивный
Десерт	Торт, пирог, мороженое, эклер

Безусловно, возможно определить значения, такие как красный цвет, соответствующие этим лингвистическим значениям, но по своему характеру такие определения являются весьма субъективными. Например, красный цвет соответствует ряду значений цвета, воспринимаемых глазом как красный, а не одному лишь значению. Дополнительные сложности связаны с такими цветами, как аквамариновый. Следует ли рассматривать его как синий или зеленый?

Лингвистические переменные обычно используются в эвристических правилах. Но значения переменных могут устанавливаться с помощью логического вывода, как показывают первые два правила в табл. 5.12.

Таблица 5.12. Некоторые эвристические правила, в которых применяются лингвистические переменные, полученные путем логического вывода

Правило
IF звук слишком слаб THEN повернуть регулятор громкости в сторону увеличения
IF вода слишком горяча THEN добавить холодной воды
IF давление слишком высоко THEN открыть предохранительный клапан
IF процентные ставки повышаются THEN покупать облигации
IF процентные ставки снижаются THEN покупать акции

В этих правилах подразумевается использование таких лингвистических переменных, как “громкость звука”, “температура воды” и т.д. Некоторые лингвистические переменные, такие как громкость звука, могут рассматриваться как метки нечетких множеств второго порядка. Например, как разные значения громкости звука могут рассматриваться уровни басового, дискантового и реверберирующего звука, где каждое из этих значений может представлять собой лингвистические переменные, принимающие такие значения, которые сами являются нечеткими множествами. Поэтому лингвистические переменные могут быть упорядочены в иерархию, соответствующую порядку их нечетких множеств. В конечном итоге достигается нечеткое множество первого порядка, такое как TALL (высокий) или BASS (басовый), которое определяется как отображение на замкнутый интервал $[0, 1]$, и поэтому лингвистическое значение становится числовой областью определения.

Множеством термов $T(L)$ лингвистической переменной L называется множество значений, которые может принимать эта переменная. В качестве примера можно привести следующее множество, где каждый из термов в $T(\text{PIE})$ представляет собой метку нечеткого множества:

$$T(\text{PIE}) = \text{CHOCOLATE} + \text{APPLE} + \text{STRAWBERRY} + \text{PECAN}$$

Такие множества могут быть объединением других множеств, состоящих из подмножеств, например, как показано ниже.

$$\begin{aligned}\text{CHOCOLATE} = {} & \text{SEMI-SWEET CHOCOLATE} + \text{MILK CHOCOLATE} \\ & + \text{DUTCH CHOCOLATE} + \text{DARK CHOCOLATE} + \ldots\end{aligned}$$

В других определениях для нечеткого множества CHOCOLATE могут применяться **барьеры** (hedge), или кванторы, модифицирующие смысл множества. Например, нечеткое множество CHOCOLATE, описывающее шоколадные изделия одного типа, может быть определено следующим образом:

$$\begin{aligned}\text{CHOCOLATE} = {} & \text{Very CHOCOLATE} + \text{Very Very CHOCOLATE} + \\ & + \text{More Or Less CHOCOLATE} + \\ & + \text{Slightly CHOCOLATE} + \\ & + \text{Plus CHOCOLATE} + \text{Not Very CHOCOLATE} + \ldots\end{aligned}$$

Как показано в табл. 5.13, стандартные барьеры могут быть определены в терминах некоторых операций с нечеткими множествами и некоторого нечеткого множества F.

Таблица 5.13. Некоторые лингвистические барьеры и операции

Барьер	Определение операции
Very F	$\text{CON}(F) = F^2$
More or Less F	$\text{DIL}(F) = F^{0.5}$
Plus F	$F^{1.25}$
Not F	$1 - F$
Not Very F	$1 - \text{CON}(F)$
Slightly F	INT [NORM (PLUS F And NOT (VERY F))]

Как показывает барьер “Not Very” (Не очень много), комбинируя операции, можно создавать другие составные барьеры. Обратите внимание на то, что слово “And” в описании барьера “Slightly” (Немного) — это знак операции с нечеткими множествами, соответствующей пересечению, $\cap$, которая распространяется на нечеткие множества “Plus” (Больше чем) и “Not (Very F)” (Не очень F).

На рис. 5.16 показана иерархия термов лингвистической переменной Appetite (Аппетит). Предполагается, что нечеткие множества LIGHT (Слабый) и HEAVY (Сильный) выражаются S-функциями, а для представления множества MODERATE (Умеренный) используется $\prod$-функция.

Обратите внимание на то, что пересекаются не только такие нечеткие множества, как LIGHT и MODERATE, но даже LIGHT и HEAVY. В данном случае

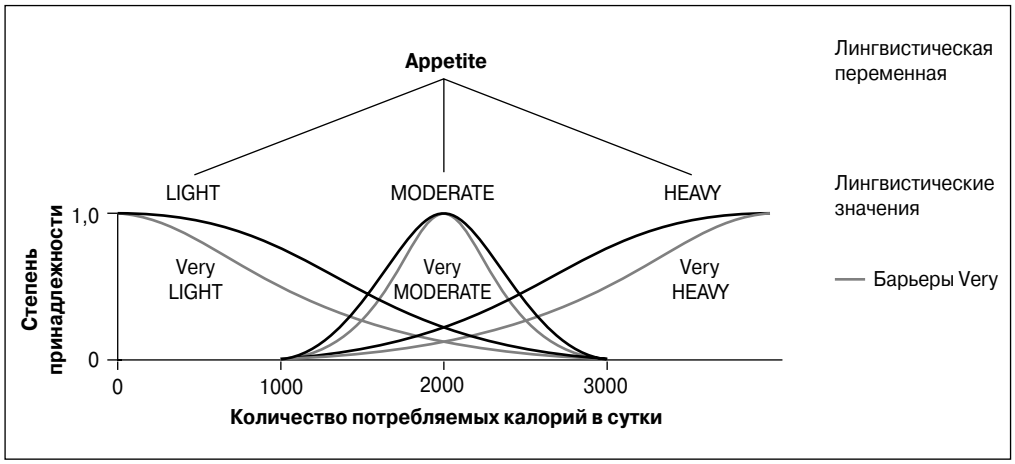


Рис. 5.16. Лингвистическая переменная Appetite и ее значения

между классическими четкими множествами не должно быть пересечения, поскольку все подобные множества являются непересекающимися. Это означает, что при использовании четкого определения аппетит LIGHT не может иметь ничего общего с аппетитом MODERATE или HEAVY. С другой стороны, при использовании нечетких множеств резкие границы между множествами обычно отсутствуют (если такие границы не определены явно).

Барьерные множества отображаются в виде курсивных кривых внутри границ самих нечетких множеств. Такие барьеры, как Very (Очень), применяются к лингвистическим переменным в качестве модификаторов для получения нечетких множеств Very LIGHT, Very MODERATE и Very HEAVY.

Лингвистическая переменная должна иметь действительные синтаксис и семантику, определяемые нечеткими множествами или правилами. А **синтаксические правила** служат для определения формально правильных выражений в множестве термов $T(L)$. В частности, следующее множество термов:

$$T(\text{Age}) = \{\text{OLD}, \text{Very OLD}, \text{Very Very OLD}, \dots\}$$

можно сформировать рекурсивно с использованием такого синтаксического правила:

$$T^{i+1} = \{\text{OLD}\} \cup \{\text{Very } T^i\}$$

Например, это правило позволяет сформировать следующий ряд множеств:

$$T^0 = \varnothing \quad (\text{пустое множество})$$
$$T^1 = \{\text{OLD}\}$$
$$T^2 = \{\text{OLD}, \text{Very OLD}\}$$
$$T^3 = \{\text{OLD}, \text{Very OLD}, \text{Very Very OLD}\}$$

Семантическое правило, связанное с множеством термов $T(L)$, определяет **смысл** терма L_i в множестве L на основании нечеткого множества. Например, семантическое правило для терма Very OLD (Очень старый) можно определить таким образом:

$$\text{Very OLD} = \mu^2_{\text{OLD}}$$

В этом правиле функция принадлежности может быть определена как следующая S-функция:

$$\mu_{OLD}(X) = S(x; 60, 70, 80)$$

Первичным термом является такой терм, как YOUNG, OLD, CHOCOLATE, STRAWBERRY и т.д., смысл которого должен быть определен до установки барьера. Барьеры модифицируют смысл первичных термов для получения других термов в множествах термов, таких как Very YOUNG (Очень молодой), Very OLD (Очень старый), Very CHOCOLATE (Имеющий большое содержание шоколада), Slightly CHOCOLATE (Содержащий небольшую добавку шоколада) и т.д. Смысл барьерных нечетких множеств можно определить, применяя соответствующие операции с нечеткими множествами, например, как показано ниже.

$$\mu_{\text{Very CHOCOLATE}} = \mu^2_{\text{CHOCOLATE}}$$
$$\mu_{NotCHOCOLATE} = 1 - \mu_{CHOCOLATE}$$
$$\mu_{MoreOrLessCHOCOLATE} = \mu^{0.5}_{\text{CHOCOLATE}}$$
$$\mu_{NotVeryCHOCOLATE} = 1 - \mu_{VeryCHOCOLATE}$$

С помощью такой системы обозначений, как нормальная форма Бэкуса–Наура (см. раздел 2.2), можно определять грамматики не только обычных языков, но и **нечеткие грамматики**. Фактически в грамматике, описанной в разделе 2.2, использовался нечеткий модификатор “heavy” (Имеющий большой вес), как в следующем правиле:

`<adjective>` $\rightarrow$ `heavy`

для формирования такой нечеткой продукции:

an eater was the heavy man

В этой продукции слово heavy (имеющий большую массу тела) является барьером, установленным на нечетком множестве man (мужчина). На первый взгляд может показаться, что множество man не должно быть нечетким. Но в том, что это действительно так, можно убедиться, попытавшись ответить на вопрос — когда мальчик становится мужчиной? У некоторых народов мальчик становится мужчиной в 12 лет или после религиозной церемонии. В законодательствах некоторых стран дано определение мужчины как лица мужского пола, достигшего 18 лет или 21 года. А в США в газетных статьях, по-видимому, принята практика называть лицо мужского пола в возрасте от 17 до 19 лет мужчиной, если его обвиняют в совершении преступления, если же таким же лицом в возрасте от 17 до 19 лет совершен какой-то похвальный поступок, его называют юношей. Лиц мужского пола, которые служат в вооруженных силах, иногда называют парнями, а иногда бойцами, особенно в речах политиков.

Нечеткую грамматику можно определить в системе обозначений на основе нормальной формы Бэкуса–Наура, вводя нетерминальные символы, как в следующем примере:

```
<Range Phrase> ::= <Hedged Primary> TO
                   <Hedged Primary>
<Hedged Primary> ::= <Hedge> <Primary> | <Primary>
<Hedge> ::= Very | More Or Less | A Little
<Primary> ::= SHORT | MEDIUM | TALL
```

Нечеткая грамматика может применяться для выработки примерно таких продукций:

Very SHORT TO Very TALL
A Little TALL TO Very SHORT
More Or Less MEDIUM TO TALL

Впервые концепция лингвистической переменной была применена в автомобиле, управляемом на основе теории нечетких множеств, который был разработан Сугено (Sugeno) в Токийском технологическом институте. В автомобиле Сугено используется система управления, основанная на нечеткой логике, которая обеспечивает автономное передвижение автомобиля на площадке прямоугольной формы. Автомобиль обладает способностью парковаться в указанном пространстве, а также обучаться на основе примеров. Лингвистические переменные использовались в правилах, на основе которых осуществлялось управление движением автомобиля. Кроме того, было создано большое количество систем управления на основе нечеткой логики других типов, предназначенных для управления устройствами и производственными установками, такими как цементные печи, применяемые для производства цемента.

Принцип расширения

Принцип расширения — очень важное понятие в теории нечетких множеств. Этот принцип определяет способ расширения области определения данной конкретной четкой функции для включения в нее нечетких множеств. С помощью принципа расширения можно расширить любую обычную, или четкую функцию из области математики, науки, инженерии, экономики и т.д. для работы в нечеткой области определения с нечеткими множествами. Благодаря использованию этого принципа нечеткие множества становятся применимыми во всех сферах человеческой деятельности.

Предположим, что f — обычная функция, которая создает отображение из универсума X в универсум Y. Если F — нечеткое подмножество универсума X, такое, что справедливо следующее выражение:

$$F = \int_x \mu_F(x)/x$$

то согласно принципу расширения определяется образ нечеткого множества F, полученный с помощью функции отображения $f(x)$, как показано ниже.

$$f(F) = \int_x \mu_F(x)/f(x)$$

Например, предположим, что функция $f(x)$ определена как четкая функция, которая возводит в квадрат значение своего параметра:

$$f(x) = x^2$$

В таком случае принцип расширения позволяет определить способ реализации функции возведения в квадрат элементов нечетких множеств:

$$f(F) = \int_x \mu_F(x)/f(x) = \int_x \mu_F(x)/x^2$$

Например, определим универсумы X и Y как закрытый интервал действительных чисел $[0, 1000]$ и нечеткое множество F следующим образом:

$$F = .3/15 + .8/20 + 1/30$$

В таком случае согласно принципу расширения определим отображение $f(F)$ таким образом:

$$\begin{aligned} f(F) &= \int_x \mu_F(x)/f(x) = \\ &= .3/f(15) + .8/f(20) + 1/f(30) = \\ &= .3/225 + .8/400 + 1/900 \end{aligned}$$

Нечеткая логика

Классическая логика лежит в основе обычных экспертных систем, а нечеткая логика формирует основу **нечетких экспертных систем**. Нечеткие экспертные системы позволяют не только справиться с неопределенностью, но и дают возможность моделировать **рассуждения на основе здравого смысла**, а эта задача с большим трудом поддается решению с помощью обычных систем. Но важной проблемой при моделировании рассуждений на основе здравого смысла является то, что при этом приходится воссоздавать колоссальную онтологию информации, которую люди используют в своих рассуждениях, даже не задумываясь об этом. Дополнительные сведения по этой теме можно найти по адресу `www.OpenCyc.org`.

Существенным ограничением классической логики является то, что она сводится к двум истинностным значениям — истина и ложь. Как было описано в главах 2 и 3, такое ограничение имеет свои преимущества и недостатки. Основным преимуществом является то, что системы, основанные на двухзначной логике, можно легко моделировать с помощью дедуктивных правил, поэтому легко обеспечивается возможность получения точных логических выводов. А основной недостаток состоит в том, что лишь очень небольшая часть реального мира действительно является двухзначной. Реальный мир — это аналоговый мир, а не цифровой (иначе могут думать только те, кто верит происходящему в фильме "Матрица").

Ограничения *двухзначной логики* были известны еще со времен Аристотеля. Безусловно, Аристотель впервые сформулировал силлогистические правила вывода и закон исключенного третьего, но он признавал, что высказывания, касающиеся будущих событий, фактически нельзя считать ни истинными, ни ложными до тех пор, пока не произошли сами события.

В дальнейшем был сформулирован целый ряд различных логических теорий, которые были основаны на множественных значениях истинности; в их число входят теории Лукашевича (Lukasiewicz), Бочвара (Bochvar), Клина (Kleene), Хейтинга (Heyting) и Рейхенбаха (Reichenbach). Широко применяются такие логические теории, которые основаны на трех значениях истинности, представляющих истину (`TRUE`), ложь (`FALSE`) и неизвестное (`UNKNOWN`). Такая **трехзначная логика** обычно представляет три истинностных значения, соответственно TRUE, FALSE и UNKNOWN, или 1, 0 и $1/2$.

Кроме того, разработаны некоторые обобщенные логики с N истинностными значениями, где N — произвольное целое число, большее или равное двум. Впервые N-значную логику разработал Лукашевич в 1930-х годах. В N-значной логике предполагается, что множество T_N истинностных значений распределено

равномерно по замкнутому интервалу $[0, 1]$:

$$T_N = \left\{ \frac{i}{N-1} \right\} \text{ для } 0 \leqslant i < N$$

В качестве примера можно привести следующие определения:

$$T_2 = \{0, 1\} \quad T_3 = \{0, 1/2, 1\}$$

В табл. 5.14 даны определения некоторых **операций логики Лукашевича** для N-значной логики, где $N \geqslant 2$. Как показано в задаче 5.13, эти операции при $N = 2$ сводятся к стандартным логическим операциям. Обратите внимание на то, что операции минус $(-)$, $\min$ и $\max$ являются такими же, как в нечеткой логике.

Таблица 5.14. Примитивные N-значные логические операции Лукашевича

Операция	Определение операции
x'	$1 - x$
$x \wedge y$	$\min(x, y)$
$x \vee y$	$\max(x, y)$
$x \rightarrow y$	$\min(1, 1 + y - x)$

Обозначение каждой N-значной логики Лукашевича, или ***L*-логики**, записывается как $\boldsymbol{L_N}$, где N — количество истинностных значений. Таким образом, L_2 — это классическая двухзначная логика, которая является одним из предельных проявлений логики Лукашевича, а другим предельным проявлением является $N = \infty$, где L_∞ определяет **бесконечнозначную логику** с истинностными значениями в множестве T_∞. Множество T_∞ определено на рациональных числах, а альтернативная бесконечнозначная логика может быть определена на континууме, который представляет собой множество всех вещественных чисел. Термин "бесконечнозначная логика" обычно применяется к этой альтернативной логике, в которой истинностными значениями являются вещественные числа в интервале $[0, 1]$; эта логика носит название L_1.

Но логику L_1 не следует путать с **унарной логикой**, в которой $N = 1$. Унарная логика — это вообще не логика Лукашевича, поскольку ***L*-логики** определены только для $N \geqslant 2$. Цифра 1 в обозначении L_1 фактически является сокращением для $\aleph_1$ (читается "алеф 1"), кардинальности вещественных чисел. Число $\aleph_1$ является не конечным, а **трансфинитным числом**. Эта теория была впервые разработана Кантором (Cantor) как способ проведения вычислений с бесконечными числами. Вместо одного бесконечного числа Кантор определил различные порядки бесконечности. Например, наименьшим трансфинитным числом является $\aleph_0$, которое представляет собой кардинальность натуральных чисел. Число $\aleph_1$ соответствует бесконечности более высокого порядка, чем $\aleph_0$, поскольку каждому

натуральному числу соответствует бесконечно большое количество вещественных чисел.

Нечеткая логика может рассматриваться как расширение многозначной логики. Но назначение и область применения нечеткой логики являются другими, поскольку нечеткая логика — это логика **приближенных рассуждений**, а не точных многозначных рассуждений. По существу, приближенные, или **нечеткие**, **рассуждения** представляют собой способ логического вывода заключений, которые могут оказаться неточными, из множества посылок, которые также могут быть неточными. Люди очень хорошо знакомы с приближенными рассуждениями, поскольку таковые представляют собой наиболее распространенный тип рассуждений, осуществляемый в реальном мире, и служат основой для многих эвристических правил. Ниже приведены некоторые примеры эвристических правил, применяемых в приближенных рассуждениях.

```
IF стереозапись звучит слишком тихо
THEN немного увеличьте громкость звука

IF стереозапись звучит так громко, что в стену стучат
   соседи
THEN увеличьте громкость звука еще больше

IF автомобильное движение становится более интенсивным
THEN автомобили, движущиеся в одном направлении, начинают
   чаще переходить с одной полосы на другую

IF вы слишком растолстели, потому что часто едите
   банановые сплиты, торты, мороженые и пирожные
THEN сократите потребление бананов
```

В ходе проведения приближенных рассуждений приходится иметь дело с такими рассуждениями, которые не являются ни точными, ни полностью неточными (такими как чистая догадка). Приближенные рассуждения наиболее тесно связаны с рассуждениями, касающимися предложений естественного языка и логических выводов, которые следуют из этих предложений. Нечеткая логика так же относится к приближенным рассуждениям, как двухзначная логика относится к точным рассуждениям. Как описано в главе 3, примерами применения точных, или строгих, рассуждений являются дедукция и доказательство теорем.

Существует возможность создания многих разновидностей теорий нечетких множеств, нечеткой логики и приближенных рассуждений. В настоящей главе в дальнейшем будет рассматриваться только разновидность нечеткой логики, основанная на теории приближенных рассуждений Заде, в которой используется нечеткая логика с основой в виде логики L_1 Лукашевича. В этой нечеткой логи-

ке истинностными значениями являются лингвистические переменные, которые в конечном итоге представлены с помощью нечетких множеств.

В табл. 5.15 даны определения операций нечеткой логики, которые основаны на операциях логики Лукашевича, приведенных в табл. 5.14. В табл. 5.15 принято обозначение $x(A)$ как числового истинностного значения в интервале $[0, 1]$, представляющего истинность высказывания "x есть A", которое может интерпретироваться как степень принадлежности $\mu_A(x)$.

Таблица 5.15. Некоторые операции нечеткой логики

Операция	**Определение 1**	**Определение 2**
$x(A')$	$x(\text{NOT } A)$	$1 - \mu_A(x)$
$x(A) \wedge x(B)$	$x(A \text{ AND } B)$	$\min(\mu_A(x), \mu_B(x))$
$x(A) \vee x(B)$	$x(A \text{ OR } B)$	$\max(\mu_A(x), \mu_B(x))$
$x(A) \rightarrow x(B)$	$x(A \rightarrow B)$	$x((\sim A) \vee B) = \max[(1 - \mu_A(x)), \mu_B(x)]$

В качестве примера применения операций нечеткой логики предположим, что имеется нечеткое множество TRUE, определенное следующим образом:

$$\text{TRUE} = .1/.1 + .3/.5 + 1/.8$$

Использование операций из табл. 5.15 позволяет получить следующее:

$$\begin{aligned}\text{FALSE} &= 1 - \text{TRUE} \\ &= (1 - .1)/.1 + (1 - .3)/.5 + (1 - 1)/.8 \\ &= .9/.1 + .7/.5\end{aligned}$$

А применяя операцию CON для постановки барьера Very (Очень), получаем такие выражения:

$$\begin{aligned}\text{Very TRUE} &= .01/.1 + .09/.5 + 1/.8 \\ \text{Very FALSE} &= .81/.1 + .49/.5\end{aligned}$$

Нечеткие правила

В качестве простого примера использования операций с нечеткими множествами рассмотрим задачу распознавания образов. Образами могут считаться такие объекты, исследуемые в целях проверки качества, как изготовленные детали или свежесобранные фрукты [44]. К числу других важных задач распознавания образов относятся диагностика по медицинским снимкам, анализ сейсмических данных, полученных при проведении разведки запасов минерального сырья и нефти, а также распознавание лиц.

В табл. 5.16 приведены некоторые гипотетические данные, представляющие степень принадлежности к нечетким множествам ракеты, истребителя и авиалайнера, которые представлены на некоторых изображениях. Эти изображения могут быть получены от систем дальнего действия, а трактовка данных изображений связана с неопределенностью, касающейся движения и ориентации объекта, обусловленной наличием шума, и т.д.

Таблица 5.16. Данные о степени принадлежности для изображений

	Степень принадлежности		
Изображение	**Ракета**	**Истребитель**	**Авиалайнер**
1	1.0	0.0	0.0
2	0.9	0.0	0.1
3	0.4	0.3	0.2
4	0.2	0.3	0.5
5	0.1	0.2	0.7
6	0.1	0.6	0.4
7	0.0	0.7	0.2
8	0.0	0.0	1.0
9	0.0	0.8	0.2
10	0.0	1.0	0.0

Объединение нечетких множеств, относящихся к каждому изображению, может служить мерилом общей неопределенности в идентификации объекта. На рис. 5.17 показаны объединения нечетких множеств для десяти изображений объектов, приведенных в табл. 5.16. Безусловно, в реальной ситуации обнаруживаются многие другие возможные изображения, отличные от этих десяти, в зависимости от разрешающей способности системы и расстояния до объекта. Неопределенность вносят не только аппаратные средства системы, но и примитивные нечеткие множества, относящиеся к ракете, истребителю и авиалайнеру, поскольку данные о степенях принадлежности назначаются субъективно, на основе знаний о физических конфигурациях типичной ракеты, истребителя и авиалайнера. В реальной ситуации может обнаруживаться много типов для каждого из таких примитивных множеств, в зависимости от различных рассматриваемых типов объектов в воздушном пространстве.

Объединения нечетких множеств, показанные на рис. 5.17, могут рассматриваться как способы представления примерно таких правил, в которых E — наблюдаемое изображение, а H — объединение нечетких множеств:

IF E THEN H

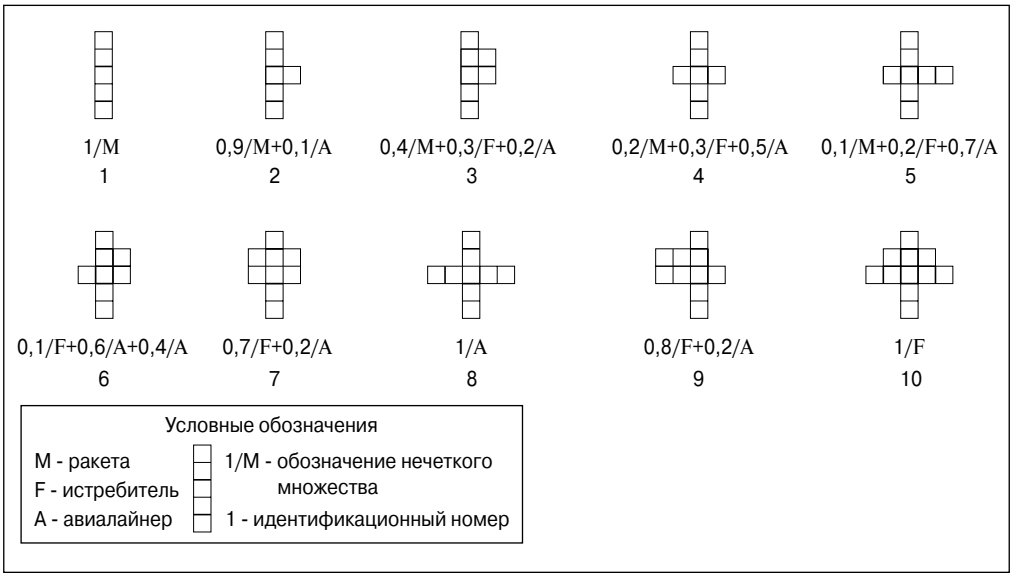


Рис. 5.17. Нечеткие множества, применяемые для распознавания объектов в воздушном пространстве

Например, может быть дано следующее правило, в котором выражение в круглых скобках представляет собой объединение нечетких множеств, соответствующих объекту:

$$\text{IF IMAGE4 THEN TARGET } (.2/M + .3/F + .5/A)$$

Еще один вариант состоит в том, что правило может быть выражено таким образом:

$$\text{IF IMAGE4 THEN TARGET4}$$

В этом выражении используется такое обозначение:

$$\text{TARGET4} = .2/M + .3/F + .5/A$$

Предположим, что имеется дополнительное время для получения данных еще одного наблюдения за объектом и наблюдается изображение IMAGE6. Такое предположение соответствует следующему правилу:

$$\text{IF IMAGE6 THEN TARGET6}$$

В этом правиле применяется обозначение:

$$\text{TARGET6} = .1/M + .6/F + .4/A$$

Общее количество элементов, для которых получены результаты измерения, относящиеся к объекту, определяется с помощью следующей формулы, в которой знак + обозначает объединение множеств:

$$\text{TARGET} = \text{TARGET4} + \text{TARGET6}$$

Таким образом, получаем следующее определение нечеткого множества TARGET, в котором сохранены только максимальные значения степени принадлежности для каждого элемента:

$$\text{TARGET} = .2/M + .3/F + .5/A + .1/M + .6/F + .4/A$$
$$\text{TARGET} = .2/M + .6/F + .5/A$$

Если в качестве наиболее вероятного объекта рассматривается элемент с максимальной степенью принадлежности, то, скорее всего, объект представляет собой истребитель, поскольку именно истребитель имеет самое высокое значение степени принадлежности, равное 0.6. Но если бы степень принадлежности для авиалайнера также была равна 0.6, то все, чего мы могли бы добиться, — это утверждать, что объект с равной вероятностью может быть истребителем или авиалайнером.

Вообще говоря, если даны N наблюдений и правил, как в следующем множестве правил, где все частные гипотезы H_i опираются на некоторую общую гипотезу H, то степень принадлежности для гипотезы H определяется объединением гипотез:

$$\text{IF } E_1 \text{ THEN } H_1$$
$$\text{IF } E_2 \text{ THEN } H_2$$
$$\vdots$$
$$\text{IF } E_N \text{ THEN } H_N$$

Таким образом, справедлива следующая формула:

$$\mu H = \max(\mu_{H_1}, \mu_{H_2}, \ldots \mu_{H_N})$$

Обратите внимание на то, что этот теоретический результат отличается от результатов. полученных на основании теории коэффициентов достоверности и теории Демпстера–Шефера. Значение μ_H гипотезы H называется **истинностным значением** гипотезы H.

Кроме того, является вполне приемлемым (за исключением политической логики) такое допущение, что истинностное значение гипотезы не может быть больше истинностного значения свидетельств. В терминах правил это предположение

соответствует тому, что истинность консеквента не может быть больше истинности его антецедентов. Таким образом, справедливо следующее соотношение, в котором каждое значение E_i может представлять собой некоторое нечеткое выражение:

$$\begin{aligned}\mu H &= \max(\mu_{H_1}, \mu_{H_2}, \ldots \mu_{H_N}) \\ &= \max[\min(\mu_{E_1}), \min(\mu_{E_2}), \ldots \min(\mu_{E_N})]\end{aligned}$$

Например, свидетельство E_1 может быть определено следующим образом, а для вычисления этого выражения могут использоваться операции нечеткой логики:

$$E_1 = E_A \text{ AND } (E_B \text{ OR NOT } E_C)$$

Поэтому справедливо следующее выражение:

$$\mu_{E_1} = \min(\mu_{E_A}, \max(\mu_{E_B}, 1 - \mu_{E_C}))$$

Такое комбинированное значение степени принадлежности антецедента называется **истинностным значением антецедента**. Это значение аналогично частичным свидетельствам, определяемым антецедентами, в правилах системы PROSPECTOR. И действительно, напомним, что свидетельства в антецедентах системы PROSPECTOR комбинировались с использованием нечеткой логики на произвольной основе. А теперь мы можем убедиться в том, что применение такого способа комбинирования обосновано правилом композиции операции логического вывода теории нечетких множеств.

Композиция max-min

Приведенное выше уравнение для гипотезы H представляет собой **правило композиции max-min операций логического вывода**, применяемого в нечеткой логике. В простейшем случае, как в следующих примерах, имеются по два элемента свидетельств в расчете на каждое правило:

$$\begin{aligned}&\text{IF } E_{11} \text{ AND } E_{12} \text{ THEN } H_1 \\ &\text{IF } E_{21} \text{ AND } E_{22} \text{ THEN } H_2 \\ &\qquad\vdots \\ &\text{IF } E_{N_1} \text{ AND } E_{N_2} \text{ THEN } H_N\end{aligned}$$

поэтому правило композиции max-min операций логического вывода принимает такой вид:

$$\mu_H = \max[\min(\mu E_{11}, \mu E_{12}), \min(\mu E_{21}, \mu E_{22}), \ldots \min(\mu E_{N_1}, \mu E_{N_2})]$$

Аналогичные расширения могут быть предусмотрены для дополнительных свидетельств E_{i_3}, E_{i_4} и т.д. В качестве еще одного примера применения правила композиции операций логического вывода рассмотрим, как оно используется к отношениям. Определим нечеткое отношение $R(x, y) =$ APPROXIMATELY EQUAL (Приблизительно равный) на бинарном отношении, в котором сравнивается масса тела людей в диапазоне от 120 до 160 фунтов (табл. 5.17).

Таблица 5.17. Отношение APPROXIMATELY EQUAL, применяемое для сравнения массы тела людей

			Y		
x	**120**	**130**	**140**	**150**	**160**
120	1.0	0.7	0.4	0.2	0.0
130	0.7	1.0	0.6	0.5	0.2
140	0.4	0.6	1.0	0.8	0.5
150	0.2	0.5	0.8	1.0	0.8
160	0.0	0.2	0.5	0.8	1.0

При подготовке этой таблицы значения степеней принадлежности были определены таким образом, чтобы отклонение от среднего между двумя значениями соответствовало уменьшению на $0,075\%$. Например, если значения x и y равны 150 и 130, то среднее составляет 140. Абсолютное отклонение значений 150 и 130 от среднего составляет $10/140 = 7,1\%$. Это значение умножается на постоянный коэффициент, равный $-0,075\%$, что приводит к получению значения $-0,5$. Таким образом, окончательное значение степени принадлежности для 150 и 130 составляет $1 - 0,5 = 0,5$, и таковым становится значение в элементе таблицы. Альтернативным и более простым с точки зрения проведения вычислений было бы определение, в котором любое изменение, равное 10, рассматривалось бы как фиксированное уменьшение значение степени принадлежности, такое как $0,3$. Но это альтернативное определение не позволяет получать приемлемые результаты для таких малых значений массы тела, как 10 и 20, которые соответствовали бы отношению APPROXIMATELY EQUAL в степени $0,7$.

Обратите внимание на то, как действует отношение $R(x, y)$ в качестве **нечеткого ограничения** на любые два значения x и y, которые имеют ненулевое значение степени принадлежности $R(x, y)$. Нечеткое отношение действует подобно **эластичному ограничению**, поскольку допускает использование ряда значений степеней принадлежности, но не требует применения такого же жесткого ограничения, как в четких отношениях. В действительности терм APPROXIMATELY EQUAL даже невозможно определить в логике, отличной от нечеткой. Жесткое ограничение, заданное в виде четких отношений, потребовало бы, чтобы значения

были либо точно равны, либо точно не равны. Иными словами, либо элемент x точно равен y, либо эти элементы не равны.

В качестве примера нечеткого ограничения рассмотрим следующее высказывание p, в котором F — некоторое нечеткое множество, действующее в качестве нечеткого ограничения на лингвистическую переменную X:

$$p = X \text{ is } F$$

Ниже приведены некоторые примеры нечетких утверждений, заданных в указанной форме.

```
John is tall
Sue is over 21
The concrete mix is too thick
Target is friendly
x is a number approximately equal to 100
The pie is fruit
```

В последнем примере нечеткое множество могло бы быть определено следующим образом для указания на то, какого рода компоненты торта рассматриваются как фрукты:

$$\text{FRUIT} = 1/\text{apples} + 1/\text{oranges}$$

Обратите внимание на то, что теория нечетких множеств позволяет определить даже осмысленный способ описания фруктового торта, состоящего из яблок и апельсин.

Теперь определим нечеткое ограничение $R_1(x)$. Например, нечеткое множество HEAVY может быть определено следующим образом:

$$R_1(x) = \text{HEAVY} = .6/140 + .8/150 + 1/160$$

Правило композиции операций логического вывода определяет нечеткое ограничение на значения y в таком виде:

$$R_3(y) = R_1(x) \circ R_2(x, y)$$

В этом выражении операция композиции $\circ$ представляет собой операцию max-min:

$$\max_x \min(\mu_1(x), \mu_2(x, y))$$

Еще один способ, позволяющий представить значение $R_3(y)$, состоит в том, что это значение можно рассматривать как решение следующих реляционных уравнений относительно $R_3(y)$:

$$R_1(x)$$
$$R_2(x, y)$$

Это означает, что если дано нечеткое ограничение, распространяющееся на значение x, и нечеткое ограничение на x и y, то можно определить значение нечеткого ограничения на y дедуктивным методом. Дедукции, подобные этой, могут осуществляться в рамках **исчисления нечетких ограничений**, которое является основой приближенных рассуждений.

Использование данных определений позволяет вычислить значение $R_3(y)$ следующим образом:

$$R_3(y) = R_1(x) \circ R_2(x, y)$$

$$R_3(y) = \begin{bmatrix} 0.0 & 0.0 & 0.6 & 0.8 & 1.0 \end{bmatrix} \circ \begin{bmatrix} 1.0 & 0.7 & 0.4 & 0.2 & 0.0 \\ 0.7 & 1.0 & 0.6 & 0.5 & 0.2 \\ 0.4 & 0.6 & 1.0 & 0.8 & 0.5 \\ 0.2 & 0.5 & 0.8 & 1.0 & 0.8 \\ 0.0 & 0.2 & 0.5 & 0.8 & 1.0 \end{bmatrix}$$

В этом выражении отношение $R_1(x)$ представлено как вектор строк. Ненулевые элементы отношения $R_3(y)$, отличные от нуля, вычисляются таким образом:

$$R_3(120) = \max\min[(.6, .4), (.8, .2)] = \max(.4, .2) = .4$$
$$R_3(130) = \max\min[(.6, .6), (.8, .5), (1, .2)] = \max(.6, .5, .2) = .6$$
$$R_3(140) = \max\min[(.6, 1), (.8, .8), (1, .5)] = \max(.6, .8, .5) = .8$$
$$R_3(150) = \max\min[(.6, .8), (.8, 1), (1, .8)] = \max(.6, .8, .8) = .8$$
$$R_3(160) = \max\min[(.6, .5), (.8, .8), (1, 1)] = \max(.5, .8, 1) = 1$$

Поэтому если отношение $R_1(x)$ представляет собой отношение HEAVY (Имеющий большую массу тела), то становится справедливым следующее соотношение:

$$R_3(y) = .4/120 + .6/130 + .8/140 + .8/150 + 1/160$$

Это отношение имеет такую грубую лингвистическую аппроксимацию: $R_3(y) -$ MORE OR LESS HEAVY (Имеет более или менее значительную массу тела).

Отношение $R_3(y)$ представляет собой грубую лингвистическую аппроксимацию, поскольку операция DIL со значением $\mu^{0.5}$, действующая на отношение HEAVY, фактически приводит к получению следующего значения, а термы, соответствующие значениям массы тела 120 и 130 фунтов, отсутствуют:

$$\text{DIL(HEAVY)} = .8/140 + .9/150 + 1/160$$

Но элементы с большими значениями степеней принадлежности, которые соответствуют условию $\mu \geqslant .8$, представлены хорошо, и поэтому утверждение, что

MORE OR LESS HEAVY — грубая аппроксимация, вполне оправдывается. Иными словами, композиционный логический вывод с помощью операций $\max$ и $\min$ показал, что является действительным следующее нечеткое лингвистическое отношение:

$$\text{MORE OR LESS HEAVY} = \text{HEAVY} \circ \text{APPROXIMATELY EQUAL}$$

Следует отметить, что эти отношения зависят от определений нечетких множеств и от толкования лингвистических меток, присвоенных множествам. Это означает, что приведенное выше отношение не является истинным в абсолютном смысле, поскольку зависит от определений нечетких множеств, отношений и меток. Но после того как приняты эти основные определения, теория нечетких множеств предоставляет механизм для манипулирования этими выражениями с помощью формального и единообразного способа. Это очень важно, поскольку таким образом лингвистические манипуляции и толкования смысла ложатся на надежный теоретический фундамент и больше не зависят от произвольных или интуитивных представлений какого-то отдельного человека.

Метод максимума и метод моментов

Метод, в котором для определения истинности консеквентов правила осуществляется выбор элемента с максимальной степенью принадлежности, называется **методом максимума**. В альтернативном методе, который называется **методом моментов**, значения истинности консеквентам правил присваиваются с применением способа, аналогичного способу вычисления первого момента инерции материального объекта в физике. Основная идея метода моментов состоит в том, что в нем рассматриваются консеквенты всех правил, применяемых для принятия решения, а не лишь один максимальный элемент. Как уже упоминалось ранее, даже метод максимума может привести к неоднозначности, если консеквенты нескольких правил имеют одно и то же максимальное значение.

В качестве простого примера применения метода моментов вначале рассмотрим следующее множество нечетких продукционных правил изготовления бетона:

R_1 : IF MIX IS TOO WET
THEN ADD SAND AND COARSE AGGREGATE
R_2 : IF MIX IS WORKABLE
THEN LEAVE ALONE
R_3 : IF MIX IS TOO STIFF
THEN DECREASE SAND AND COARSE AGGREGATE

Бетон состоит из цемента, воды, песка и крупного заполнителя, такого как гравий. Все эти компоненты смешиваются в нужных пропорциях. Пробное ко-

личество бетонной смеси, которое изготавливается для определения пропорций, оптимальных с точки зрения намеченного назначения бетонной смеси, называется **замесом**. Разработаны общие рекомендации по определению пропорций в смеси с учетом требуемой прочности застывшего бетона. Но на практике наблюдается изменчивость результирующих характеристик из-за особенностей используемых местных материалов, отклонений в размерах частей крупного заполнителя, условий окружающей среды и других факторов. Поэтому рекомендуется проводить испытание путем изготовления замеса, прежде чем приступать к строительству здания стоимостью 10 миллионов долларов.

Одним из распространенных методов определения того, является ли замес составленным правильно, т.е. практически применимым, является **проверка на растекание**. Пробная порция бетона помещается в конусообразную форму, после чего эта форма удаляется. Площадь, на которой растекается замес после удаления формы, позволяет судить о состоянии материала. Бетон, предназначенный для изготовления обычных горизонтальных и вертикальных балок, должен иметь минимальный и максимальный диаметры растекания, соответственно 4 и 8 дюймов. На рис. 5.18 показаны нечеткие множества, которые иллюстрируют возможные определения для антецедентов нечетких продукционных правил изготовления бетонных замесов. Такие нечеткие множества могут быть также определены с помощью таблицы или заданы в виде S- и $\prod$-функций.



Рис. 5.18. Антецеденты нечетких продукционных правил, применяемых в управлении процессом изготовления бетонной смеси

Нечеткие множества, используемые для определения консеквентов нечетких правил, показаны на рис. 5.19. Как и в случае антецедентов, обнаруживаются пересечения множеств, соответствующих действиям нечетких консеквентов, с помощью которых создаются замесы, применимые на практике. Следует отметить,

что пределы изменений, равные -20 и 20%, для песка и крупного заполнителя определены произвольным образом. Как и антецеденты в нечетком множестве, эти консеквенты также могут быть определены с помощью S- и $\prod$-функций.

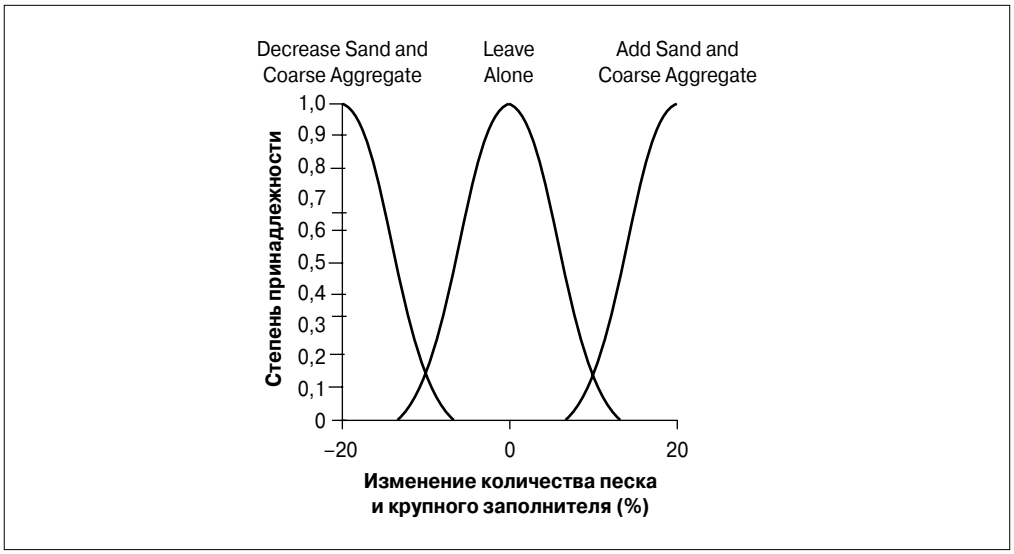


Рис. 5.19. Консеквенты нечетких продукционных правил, применяемых в управлении процессом изготовления бетонной смеси

В качестве примера применения этого множества нечетких продукционных правил рассмотрим вариант, в котором бетонный замес после растекания образовал круг диаметром 6 дюймов. Согласно рис. 5.18, степень принадлежности, или истинностное значение антецедента для каждого правила, определяется следующим образом:

$$\mu_{\text{TOO STIFF}}(6) = 0$$
$$\mu_{\text{WORKABLE}}(6) = 1$$
$$\mu_{\text{TOO WET}}(6) = 0$$

Это означает, что единственным правилом, антецедент которого выполняется, является правило R_2. Это правило активизируется и запускается с получением нечеткого консеквента LEAVE ALONE (Не вносить изменения).

Применение правила композиции операций логического вывода приводит к получению такого результата:

$$\mu_{\text{LEAVE ALONE}} = \max[\min(\mu_{\text{WORKABLE}}(6)] = \max[\min(1)] = 1$$

Согласно данным, приведенным на рис. 5.19, этот результат соответствует внесению изменений в долю песка и крупного заполнителя, равных 0%.

Теперь предположим, что диаметр растекания равен 4,8 дюйма. Согласно данным, приведенным на рис. 5.18, получаем следующее:

$$\mu_{\text{TOO STIFF}}(4.8) = .05$$
$$\mu_{\text{WORKABLE}}(4.8) = .2$$
$$\mu_{\text{TOO WET}}(4.8) = 0$$

Правила R_1 и R_2 имеют антецеденты, выполняемые частично. Эта ситуация аналогична частичному выполнению свидетельств в антецедентах вероятностных правил системы PROSPECTOR. С гипотезами, согласно которым замес является слишком густым (TOO STIFF) или практически применимым (WORKABLE), связаны некоторые ненулевые истинностные значения, поэтому происходит активизация и запуск обоих правил, R_1 и R_2. Следует отметить, что в системе, основанной на использовании нечетких продукционных правил, запускается каждое правило с ненулевым истинностным значением антецедента, если не установлена пороговая величина истинностного значения антецедента. Установка пороговой величины может потребоваться для предотвращения неэффективных действий, связанных с тем, что становятся активизированными и запускаются слишком многие правила с низкими истинностными значениями. Напомним, что в системе MYCIN в целях повышения эффективности работы экспертной системы предусмотрено, что минимальный уровень достоверности правила, подлежащего запуску, должен быть больше 0.2. Аналогичная пороговая величина может применяться к истинностным значениям антецедентов нечеткого множества.

Итак, если диаметр растекания равен 4,8 дюйма, активизируются два правила. Применяя к правилам операцию композиции max-min, получаем следующее:

$$\begin{aligned}\mu_{\text{DECREASE SAND AND COARSE AGGREGATE}} &= \max[\min(\mu_{\text{TOO STIFF}}(4.8)] \\ &= .05 \\ \mu_{\text{LEAVE ALONE}} &= \max[\min(\mu_{\text{WORKABLE}}(4.8)] = .2\end{aligned}$$

Вполне очевидно, что в случае использования термов с одним антецедентом функции $\max$ и $\min$ не требуются.

Таким образом, на данном этапе обнаружены два правила с ненулевыми консеквентами, поэтому необходимо выбрать одно из управляющих воздействий. В методе максимума предусматривается просто выбор правила с максимальной степенью принадлежности. В данном случае выбирается действие LEAVE ALONE, поскольку для него степень равна 0.2, в отличие от значения 0.05, соответствующего другому правилу.

С другой стороны, в методе моментов по существу определяется **центр масс** правил с нечеткими консеквентами. Термин “центр масс” взят из физики. Центром масс в физике называется такая точка, что если бы в ней была сосредоточена

вся масса объекта, то полученная точечная масса вела бы себя под воздействием внешних сил так же, как сам исходный объект. Для определения центра масс применяется следующая формула, называемая также определением **первого момента инерции**, I:

$$I = \frac{\int m(x)x \; d(x)}{\int m(x) \; d(x)}$$

В этой формуле знак интеграла обозначает обычную операцию интегрирования.

На рис. 5.20 показаны нечеткие множества для двух правил, R_2 и R_3. Следует отметить, что значения степеней принадлежности нечетких множеств ограничиваются (усекаются) в зависимости от истинностных значений антецедентов правил, определяющих эти множества. Это соответствует правилу композиции операций логического вывода. Усечение выполняется в связи с тем, что даже интуитивные соображения подтверждают необходимость соблюдения требования, чтобы истинностные значения консеквентов не превышали истинностные значения соответствующих им антецедентов.

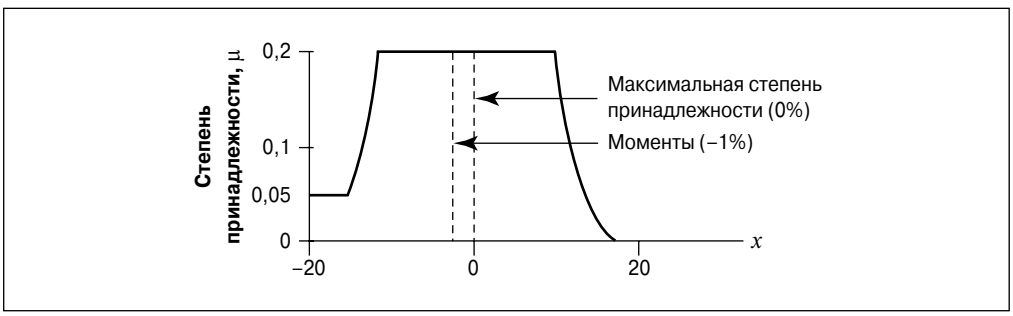


Рис. 5.20. Примеры применения метода максимума и метода моментов к нечетким правилам управления процессом изготовления бетона

Значения моментов для консеквентов вычисляются следующим образом и при использовании данных, приведенных на рис. 5.20, составляют приблизительно -1%:

$$I = \frac{\int m(x)x \; d(x)}{\int m(x) \; d(x)} \qquad \text{для непрерывных элементов}$$

или

$$I = \frac{\sum\limits_i \mu_i x}{\sum\limits_i \mu_i} \qquad \text{для дискретных элементов}$$

Безусловно, эти значения довольно близки к значению 0%, полученному с помощью метода максимума, но подобные различия могут стать более существенными

при использовании нечетких множеств, которые определены со значительной долей пересечения. В одном из контроллеров, действующих на основе нечеткой логики, который предназначен для распознавания типов воздушных судов, были реализованы и метод максимума, и метод моментов. В частности, в этом контроллере метод максимума применялся для расчета арифметического среднего всех максимумов для нечетких консеквентов. Таким образом, даже при наличии нескольких элементов с одним и тем же значением максимума все еще обеспечивалась возможность вычислить одно четкое управляющее значение.

Для решения такой **задачи перехода от нечеткого представления к четкому** (defuzzification problem), кроме метода максимума и метода моментов, применяются другие подходы, которые сводятся к преобразованию значения степени принадлежности в четкое управляющее значение, или к осуществлению лингвистической аппроксимации описания управляющей переменной. Но задача описания множества значений (нечеткого множества) с помощью одного числа или одного словесного выражения (кроме тех случаев, когда ваш автомобиль кто-то резко подрезает во время движения) является довольно сложной.

Возможность и вероятность

В теории нечетких множеств термин **возможность** имеет особый смысл. По существу, когда речь идет о возможности, подразумеваются допустимые значения. Например, предположим, что определено следующее высказывание, касающееся результатов броска двух игральных костей в универсуме $\mathcal{U}$ событий получения суммы очков на этих костях:

$$p = X - \text{целое число в } \mathcal{U}$$
$$\mathcal{U} = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

В терминологии нечетких множеств для любого целого числа i справедливы следующие соотношения:

$$\text{Poss}\{X = i\} = 1 \qquad \text{для } 2 \leqslant i \leqslant 12$$
$$\text{Poss}\{X = i\} = 0 \qquad \text{в противном случае}$$

В этих формулах терм **Poss** $\{X = i\}$ является сокращением для выражения "возможность того, что X может принять значение i". Возможность того, что сумма очков на игральных костях составит значение от 2 до 12, весьма отличается от вероятности любого значения i. Иными словами, **распределение возможностей** — это не то же самое, что **распределение вероятностей**. Распределение вероятностей выпадения определенной суммы на игральных костях характеризуется частотой ожидаемого появления **случайной переменной** X, которая обозначает сумму очков. Например, сумма очков, равная 7, может появиться в результате сложения

количества очков $1+6$, $2+5$ и $3+4$. Поэтому вероятность выпадения суммы 7 равна:

$$\frac{2 \cdot 3}{36} = \frac{1}{6}$$

а вероятность выпадения суммы 2 является таковой:

$$\frac{1}{36}$$

В отличие от этого, распределение возможностей представляет собой постоянное значение, равное 1, для правильных игральных костей применительно ко всем целым числам от 2 до 12. Принято применять такую формулировку, что из высказывания p логически следует распределение возможностей $\prod_X$. Таким образом, если дано нечеткое утверждение p, основанное на нечетком множестве F и лингвистической переменной X, то имеет место следующее высказывание:

$$p = X \text{ is } F$$

Если рассматриваемое высказывание выражено таким образом, его называют высказыванием в **канонической форме**; под термином "канонический" подразумевается "стандартный". Нечеткое множество F, в отличие от предикатов обычной логики, определяет **нечеткий предикат**. Множество F может также представлять собой нечеткое отношение. Распределение возможностей, логически выведенное на основании высказывания p, равно F и определяется с помощью следующего **уравнения присваивания значения возможности**:

$$\prod_X = F$$

Под этим подразумевается, что для всех значений x в универсуме $\mathcal{U}$ справедливо такое соотношение:

$$\text{Poss}(X = x\} = \mu_F(x) \quad x \in \mathcal{U}$$

Например, если дано следующее высказывание:

$$p = \text{John is tall}$$

то на его основе можно определить лингвистическую переменную Height со значением John. Следующая каноническая форма:

$$X \text{ is } F$$

может быть представлена в терминах переменной Height следующим образом:

$$\text{Height(John) is TALL}$$

а также с помощью формулы

$$\text{Poss}\{\text{Height}(\text{John}) = x\} = \mu_{\text{TALL}}(x)$$

Высказывание p может быть записано как распределение возможностей следующим образом:

$$\text{John is tall} \rightarrow \prod_{\text{Height(John)}} = \text{TALL}$$

В этой формуле символ стрелки означает "преобразуется в", обозначение Height представляет собой лингвистическую переменную, а TALL обозначает нечеткое множество. Обратите внимание на то, что John — не лингвистическая переменная.

Безусловно, любому распределению возможностей можно присвоить нечеткое множество, как в формуле $\prod_X = F$, но фактически эти два понятия не совпадают. В качестве примера, позволяющего проиллюстрировать различия между ними, рассмотрим следующее нечеткое множество, определяющее результаты броска игральных костей:

$$\text{ROLL}(1) = 1/3 + 1/4$$

Это множество определено как описание результата конкретного броска игральных костей, Roll 1, в котором на одной игральной кости выпало 3, а на другой — 4 очка. В отличие от этого, следующее распределение возможностей означает, что бросок привел к получению количества очков 3 или 4, а само слово "или" в последнем выражении фактически равнозначно операции "исключительное ИЛИ", представляющей неопределенность наших знаний о результатах броска:

$$\prod_{\text{ROLL}} (1) = 1/3 + 1/4$$

Существует единичная возможность, что результатом стали 3 очка, и единичная возможность, что было получено 4 очка. В этом нечетком множестве достоверно лишь то, что сумма очков на игральных костях может составить и 3, и 4. Это распределение возможностей означает, что игральные кости допустимо считать правильными с точки зрения того, может ли на них выпасть значение 3 или 4. Нечеткое множество сообщает, какие значения обнаруживаются после броска.

В качестве еще одного примера рассмотрим такое высказывание, что "Ганс съел на завтрак X яиц", где X — любое значение в универсуме $X = \{1, 2, \ldots 8\}$, как показано в табл. 5.18.

Таблица 5.18. Определение условий анализа утверждения "Ганс съел на завтрак X яиц"

$\boldsymbol{X}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$\prod_{\textbf{ATE(Hans)}}(\boldsymbol{X})$	1.0	1.0	1.0	1.0	0.8	0.6	0.4	0.2
$\boldsymbol{P(X)}$	0.1	0.8	0.1	0.0	0.0	0.0	0.0	0.0

Распределение возможностей $\prod_{\text{ATE(Hans)}}(X)$ интерпретируется как характеристика того, насколько легко Ганс может съесть X яиц. А распределение вероятностей $P(X)$ можно определить опытным путем, предварительно спросив Ганса, позволит ли он вам присутствовать во время его завтрака в течение года для проведения научного исследования.

В данном случае важнее всего понять, что определение концепции возможности — не статистическое, а определение концепции вероятности — статистическое. Например, безусловно, Ганс способен съесть восемь яиц, но эмпирическое исследование показало, что за весь период наблюдения он ни разу не съел больше трех яиц (и всегда старался в этом сдерживаться). В таком смысле возможность равнозначна способности, или продуктивности. Но высокая степень возможности не обязательно означает высокую степень вероятности. Таким образом, корреляция между возможностью и вероятностью может отсутствовать. Например, возможность наличия оружия массового уничтожения не означает, что при отсутствии конкретного свидетельства можно утверждать о вероятности наличия оружия массового уничтожения.

В теории нечетких множеств обычное определение вероятности дополнено и сформулировано понятие **нечеткой вероятности**, которое описывает вероятности, известные, но лишь неточно. В качестве некоторых примеров нечетких вероятностей можно назвать **нечеткие спецификаторы** Very Likely (Весьма вероятно), Unlikely (Невероятно), Not Very Likely (Не очень вероятно) и т.д. Пример нечеткого высказывания с нечеткой вероятностью приведен ниже.

the battery is BAD is Very Likely

Правила преобразования

Определение нечеткой вероятности реализовано в нечеткой логике, основанной на логике L_1 Лукашевича, которую принято сокращенно обозначать как **FL** (fuzzy logic). Одним из основных компонентов нечеткой логики FL является группа **правил преобразования**, которые указывают, как создаются модифицированные или сложные высказывания на основе содержащихся в них элементарных высказываний.

Правила преобразования подразделяются на четыре описанных ниже категории.

- Правила типа I — **правила модификации**. Примеры таких правил приведены ниже.

  X is very large

  John is much taller than Mike

- Правила типа II — **правила композиции**. Примеры таких правил приведены ниже.

условная композиция

If X is TALL then Y is SHORT

конъюнктивная композиция

X is TALL and Y is SHORT

дизъюнктивная композиция

X is TALL or Y is SHORT

условная и конъюнктивная композиция

If X is TALL then Y is SHORT
else Y is Rather SHORT

- Правила типа III — **правила квантификации**. Примеры таких правил приведены ниже.

  Most desserts are WONDERFUL
  Too Much nutritious food is FATTENING

- Правила типа IV — **правила оценки**. Примеры таких правил приведены ниже.

  истинностная оценка

  chocolate pie is DELICIOUS is Very True

  оценка вероятности

  chocolate pie is served SOON is Very Likely

  оценка возможности

  chocolate pie is BAD for you is Impossible

Правилами оценки (qualification rule) называются правила, относящиеся к нечетким вероятностям, а правила квантификации (quantification rule) предназначены для использования с нечеткими кванторами, такими как `Most` (Большинство), которые не могут быть определены с использованием классических кванторов всеобщности и существования.

Ниже приведен пример преобразования с использованием правила типа I.

$$X \text{ is } F \rightarrow \prod_X = F$$

Преобразованное высказывание выглядит следующим образом:

$$X \text{ is } mF \rightarrow \prod_X = F^+$$

В этом выражении m представляет собой модификатор, такой как Not (Нет), Very (Очень), More Or Less (Более или менее) и т.д. Терм F^+ представляет собой результат модификации высказывания F с помощью модификатора m. В табл. 5.5 представлены некоторые применяемые по умолчанию определения для m и F^+. Эти термы определяются так же, как лингвистические барьеры, которые рассматривались выше. Могут также использоваться другие определения для m и F^+.

Таблица 5.19. Значения параметров преобразования для некоторых правил типа I; следует отметить, что все интегралы берутся по универсуму

$\boldsymbol{m}$	$\boldsymbol{F^+}$
Not	$F' = \int [1 - \mu_F(x)]/x$
Very	$F^2 = \int \mu_{F^2}(x)/x$
More Or Less	$\sqrt{F} = \int \sqrt{\mu_F(x)}/x$

В качестве примера определим нечеткое множество TALL (Высокий) следующим образом:

$$\text{TALL} = .2/5 + .6/6 + 1/7$$

В таком случае могут быть выполнены следующие преобразования:

$$\text{John is not tall} \rightarrow .8/5 + .4/6 + 0/7$$
$$\text{John is very tall} \rightarrow .04/5 + .36/6 + 1/7$$
$$\text{John is more or less tall} \rightarrow .45/5 + .77/6 + 1/7$$

Переменная X не обязательно должна быть унарной переменной. Вместо этого в качестве X может быть, вообще говоря, выбрано бинарное или N-арное отношение. Например, такое высказывание, как "Y и Z есть F", обобщается высказыванием "X есть F". Определим CLOSE как нечеткое бинарное отношение в декартовом произведении двух универсумов, $\mathcal{U} \times \mathcal{U}$:

$$\text{CLOSE} = 1/(1,1) + .5/(1,2) + .5/(2,1)$$

В этом случае могут быть выполнены следующие преобразования:

$$\begin{aligned} &X \text{ and } Y \text{ are close} \rightarrow \prod\nolimits_{(X,Y)} = \text{CLOSE} \\ &X \text{ and } Y \text{ are very close} \rightarrow \prod\nolimits_{(X,Y)} \\ &\quad = \text{CLOSE}^2 \\ &\quad = 1/(1,1) + .25/(1,2) + .25/(2,1) \end{aligned}$$

Ниже приведен пример применения правила типа II.

$$\text{IF } X \text{ is } F \text{ then } Y \text{ is } G \rightarrow \prod\nolimits_{(X,Y)} = \overline{F}' \oplus \overline{G}$$

В этом выражении термы $\overline{F}$ и $\overline{G}$ представляют собой следующие расширения множеств F и G в их универсумах, а $\oplus$ обозначает ограниченную сумму:

$$\overline{F} = F \times V \quad \overline{G} = U \times G$$

Для этого правила имеет место также следующее соотношение, а функция $\min$ выражается как $\wedge$:

$$\mu_{\overline{F}' \oplus \overline{G}}(x, y) = 1 \wedge [1 - \mu_F(x) + \mu_G(y)]$$

Это определение совместимо с операцией импликации в логике L_1, тогда как другие определения, возможно, окажутся несовместимыми. В качестве примера применения правила типа II определим следующее:

$$\begin{aligned} &U = V = \{4, 5, 6, 7\} \\ &F = \text{TALL} = .2/5 + .6/6 + 1/7 \\ &G = \text{SHORT} = 1/4 + .2/5 \\ &\text{IF } X \text{ is tall then } Y \text{ is short} \rightarrow \prod\nolimits_{(X,Y)} \\ &\quad = 1/(5,4) + 1/(5,5) + 1/(6,4) + \\ &\quad\quad .6/(6,5) + 1/(7,4) + .2(7,5) \end{aligned}$$

В данных выражениях значения степени принадлежности элемента, такие как $(5, 4)$, вычисляются таким образом:

$$\mu_{\overline{F}' \oplus \overline{G}}(5, 4) = 1 \wedge [1 - .2 + 1] = 1 \wedge [1.8] = 1$$

Неопределенность в нечетких экспертных системах

При использовании нечетких вероятностей в экспертных системах приходится учитывать некоторые отличия по сравнению с обычным вероятностным логическим выводом. Рассмотрим такое каноническое нечеткое правило:

$$\text{If } X \text{ is } F \text{ then } Y \text{ is } G(\text{с вероятностью } \beta)$$

Это правило может быть записано как условная вероятность:

$$P(Y \text{ is } G \mid X \text{ is } F) = \beta$$

В таком случае в обычной экспертной системе, основанной на использовании классической теории вероятностей, было бы принято следующее предположение:

$$P(Y \text{ is not } G \mid X \text{ is } F) = 1 - \beta$$

Но в теории нечетких множеств, если F — нечеткое множество, это предположение было бы неправильным. В действительности результат, соответствующий теории нечетких множеств, является более слабым, поскольку он позволяет задавать лишь более низкие пределы значений вероятностей, которые могут быть определены как нечеткие числа:

$$P(Y \text{ is not } G \mid X \text{ is } F) + P(Y \text{ is } G \mid X \text{ is } F) = 1$$

Вообще говоря, для нечетких систем имеет место следующее соотношение:

$$P(H \mid E) \text{ is not necessarily equal to } 1 - P(H' \mid E)$$

В нечетких экспертных системах нечеткость может наблюдаться в трех описанных ниже областях.

Во-первых, в антецедентах и (или) консеквентах правил, подобных следующему:

$$\text{If } X \text{ is } F \text{ then } Y \text{ is } G$$
$$\text{If } X \text{ is } F \text{ then } Y \text{ is } G \text{ with } CF = \alpha$$

В этом правиле CF обозначает коэффициент достоверности, а α представляет собой числовое значение, такое как 0.5.

Во-вторых, в частичном соответствии между антецедентом и фактами, которые согласуются с шаблонами антецедента. В экспертных системах, отличных от нечетких, правило не становится активизированным, если шаблоны не согласуются точно с фактами. С другой стороны, в нечеткой экспертной системе все события зависят от степени проявления определенных характеристик, поэтому

всегда существует возможность активизации любых правил, если не установлено пороговое значение.

В-третьих, в нечетких кванторах, таких как Most (Большинство), и нечетких спецификаторах, подобных Very Likely (Весьма вероятно), Quite True (Полностью истинно), Definitely Possible (Безусловно возможно) и т.д. Высказывания часто содержат неявные и (или) явные нечеткие кванторы. В качестве примера рассмотрим следующее **разъяснение**:

$$d = \text{desserts are WONDERFUL}$$

Термин "разъяснение" обозначает высказывание, которое обычно является истинным. Разъяснения имеют следующую каноническую форму, в которой Usually обозначает подразумеваемый нечеткий квантор, а R представляет собой **ограничивающее отношение**, воздействующее на ограничиваемую переменную X, лимитируя значения, которые она может принимать:

$$\text{Usually}(X \text{ is } R)$$

Разъяснениями являются по сути многие эвристические правила, которыми пользуются люди. В действительности **знания, основанные на здравом смысле**, являются в сущности коллекцией разъяснений, относящихся к реальному миру.

Разъяснение может быть преобразовано в другие явные пропозициональные формы:

$$p = \text{Usually desserts are wonderful}$$
$$p = \text{Most desserts are wonderful}$$

В свою очередь, эти формы могут быть выражены как эвристические правила:

$$r = \text{If } x \text{ is a dessert}$$
$$\text{then it is likely that } x \text{ is wonderful}$$

Ниже приведены некоторые примеры правил вывода, применяемых в нечетких системах.

- **Принцип следования**

$$\frac{X \text{ is } F}{F \subset G}$$
$$X \text{ is } G$$

- **Распорядительное следование** ограничивается случаями, в которых квантор Usually (Обычно) становится квантором Always (Всегда)

$$\text{Usually}(X \text{ is } F)$$
$$\frac{F \subset G}{\text{Usually}(X \text{ is } G)}$$

- Композиционное правило

$$\begin{array}{l} X \text{ is } F \\ \underline{(X,Y) \text{ is } R} \\ Y \text{ is } F \circ R \end{array}$$

В этих формулах R — бинарное отношение по бинарной переменной (X, Y), а выражение $F \circ R$ представляет собой следующее:

$$\mu_{F \circ R}(y) = \sup_x [\mu_F(x) \wedge \mu_R(x, y)]$$

Кроме того, в рассматриваемом композиционном правиле неявно применяется операция определения **супремума** (символически обозначаемая как **sup**), которая представляет собой операцию определения **наименьшей верхней грани**. Вообще говоря, операция определения супремума по своему назначению аналогична функции $\max$. Различие между ними возникает в тех случаях, когда максимальное значение отсутствует, например, рассматривается интервал вещественных чисел меньше 0. Поскольку максимальное вещественное число, которое было бы меньше 0, не существует, используется операция определения супремума, позволяющая принять в качестве наименьшей верхней грани значение 0.

- Обобщенное правило модус поненс

$$\begin{array}{l} X \text{ is } F \\ \underline{Y \text{ is } G \text{ if } X \text{ is } H} \\ Y \text{ is } F \circ (H' \oplus G) \end{array}$$

В последней формуле H' обозначает нечеткое отрицание H, а ограниченная сумма определяется следующим образом:

$$\mu_{H' \oplus G}(x, y) = 1 \wedge [1 - \mu_H(x) + \mu_G(y)]$$

При использовании обобщенного правила модус поненс не требуется, чтобы антецедент "X есть H" был идентичен посылке "X есть F". Обратите внимание на то, что в этом состоит существенное расхождение с классической логикой, которая требует, чтобы подобные термы точно совпадали. Обобщенное правило модус поненс фактически представляет собой частный случай правила композиции операций логического вывода. В обычных экспертных системах основным правилом логического вывода является модус поненс, но, в отличие от них, в нечетких экспертных системах основным правилом служит правило композиции операций логического вывода.

В экспертных системах, действующих на основе приближенных рассуждений, может использоваться целый ряд различных методов, таких как ограничение истинностных значений и композиционный логический вывод. В обзоре одиннадцати нечетких экспертных систем, подготовленном Вейленом (Whalen), показано, что почти во всех этих системах используется композиционный логический вывод. Обычно трудно предсказать заранее, какой метод окажется наиболее приемлемым, поэтому для определения метода, в наибольшей степени подходящего для обработки имеющихся данных, применяется эмпирическое моделирование. Это положение не столь безнадежно, каким кажется на первый взгляд, поскольку, например, в статистике в первую очередь предпринимается попытка использовать линейную регрессию, так как этот метод наиболее прост, а предположение о гауссовом характере распределения вероятностей в совокупности обычно оправдывается. Если этот метод не подходит, то предпринимается попытка применения более сложной модели, такой как подбор многочлена, и т.д. Эти попытки осуществляются до тех пор, пока отклонения не станут приемлемыми.

5.6 Состояние неопределенности

Боже мой! Если вы прочитали каждое слово в первых пяти главах, вывели каждое уравнение и проработали каждое задание, то вы либо имеете неутолимую жажду знаний, либо страдаете от бессонницы. Но в любом случае после прочтения оставшейся части данной книги вы сможете решить любую задачу, стоящую перед вами. Но на данном этапе важнее всего понять, сумеем ли мы, блуждая между деревьями в лесу, состоящем из разнообразных и конкурирующих теорий, увидеть возвышающиеся над ним горы? Как обрести уверенность в том, что мы знаем правильный способ создания экспертных систем?

Вдалеке стоят две горы, которые возвышаются над всеми деревьями и лесами. Одна из этих гор поднимается очень высоко и видна очень четко. Это — гора Логики, на которой должны формироваться все экспертные системы. Экспертная система, достигая действительных заключений после получения действительных посылок, должна действовать подобно человеку. Экспертная система обязана вырабатывать действительные заключения, если, во-первых, ей даны правильно сформулированные правила, и, во-вторых, факты, на основании которых машина логического вывода экспертной системы должна выработать действительные заключения, являются истинными. Обратите внимание на то, что речь не идет об использовании обоснованных фактов, в том смысле, что эти факты являются истинными в реальном мире. Кроме того, не следует рассчитывать на получение от машины логического вывода непротиворечивых заключений, если в рассуждениях эксперта, заложенных в системе, есть противоречия, а факты недействительны. Экспертная система позволяет лишь моделировать экспертные знания человека

в ограниченной области знаний, а люди не всегда рассуждают рационально. Но по меньшей мере мы можем рассчитывать на то, что экспертная система придет к действительным заключениям, если ей будут даны действительные факты. (В противном случае нам останется утешать себя лишь тем, что впервые удалось призвать на службу программу вместо человека.)

Второй горой, которая поднимается очень высоко, но видна, как в тумане, является гора Неопределенности. Странно то, что эту гору не удается разглядеть более четко, сколь близко бы вы к ней не подходили. В действительности гора Неопределенности изрезана глубокими и сложными трещинами, поэтому изучение ее определенного аспекта, т.е. конкретной теории, приводит к открытию новых разновидностей теории, которые, в свою очередь, включают другие разновидности. Поэтому наилучший способ изучения этой горы состоит в том, чтобы учесть опыт освоивших ее альпинистов — моделировать экспертные знания специалиста или же попытаться применить более одного подхода к учету неопределенности, после чего организовать соревнование между различными основанными на них методами. Такой эволюционный принцип основан на классической **архитектуре классной доски**, в которой различные агенты одновременно работают над одной задачей, атакуя ее с разных сторон. Все агенты отправляют результаты выполненного ими анализа фрагментов головоломки на общедоступную классную доску для общего обозрения, в надежде на то, что внезапно удастся сложить вместе разные фрагменты. Управляющая программа постоянно наблюдает за появлением всех небольших фрагментов и пытается сформировать общую картину. Если бы в ландшафте неопределенности существовала конкретная теория, то неопределенность уже не была бы неопределенностью.

В настоящее время для решения задач, связанных с неопределенностью, наиболее часто используют нечеткую логику и байесовскую теорию. Но обе эти теории имеют много разновидностей, которые также применялись при осуществлении попыток создания работоспособных приложений. Поэтому важно выбрать такое программное обеспечение, которое позволяет воспользоваться наиболее широкими возможностями выбора. Как обычно, наибольшее количество вариантов предоставляет коммерческое программное обеспечение, но его стоимость также достаточно высока. Однако во многих случаях возможно приобретение коммерческого программного обеспечения по ценам, предназначенным для академических учреждений, или получение испытательных версий, которые рассчитаны на эксплуатацию в течение некоторого времени.

В связи с тем что теперь стали широко доступными мощные настольные компьютеры и появилась возможность связывать тысячи и даже миллионы компьютеров с помощью сети Grid, могут успешно проводиться испытания различных моделей неопределенности на очень больших наборах данных в течение короткого времени. Но при использовании всей этой массовой мультикомпьютерной среды необходимо соблюдать осторожность в связи с тем, что может возникнуть про-

блема чрезмерно тщательной подгонки данных с применением такого большого количества параметров, что будет достигнуто 100%-ное согласование с проверочным набором данных. Дело в том, что 100%-ное согласование является таким же неприемлемым, как и 0%-ное (или, наоборот, приемлемым, в зависимости от того, как к этому относится исследователь), поскольку оно означает, что имеет место какое-то упущение. Ведь из статистики известно, что в данных всегда должны быть случайные флуктуации из-за наличия шума или конечного размера выборки.

Например, если имеется N точек данных, то всегда можно начертить ряд из $N - 1$ отрезков прямой, проходящий от первой точки до последней. Такая модель идеально подходит для интерполяции данных, лежащих между двумя точками, но фактически становится совершенно бесполезной для экстраполяции данных, выходящих за пределы существующих. От экспертной системы в действительности требуется, чтобы она обладала способностью предсказывать и экстраполировать, переходя от известных случаев к новым, как это делает человек.

Программные инструментальные средства позволяют воспользоваться очень многими методами, в том числе основанными на классической байесовской теории, **байесовской теории с выпуклыми множествами**, теории Демпстера–Шефера, **теории Кайберга** (Kyburg) и теории возможностей, разработанной в рамках нечеткой логики. Поэтому исследователю может потребоваться гораздо больше времени на опробование различных моделей, чем на сам анализ. При использовании подхода к созданию байесовской теории на основе выпуклых множеств доверительное состояние не рассматривается как единственная функция, в отличие от классического байесовского метода. Вместо этого доверительное состояние характеризуется с помощью множества **выпуклых функций**. Это означает, что любая функция может быть представлена как линейная комбинация двух других функций.

С другой стороны, что касается теории Демпстера–Шефера, то вслед за первоначальным признанием ее как обобщения классической теории вероятностей были опубликованы опровержения. Например, в статье Кайберга содержатся утверждения, что фактически не теория Демпстера–Шефера является обобщением классической теории вероятностей, а последняя является обобщением первой. Кайберг считает, что теория Демпстера–Шефера укладывается в рамки классической теории вероятностей и что интервалы Демпстера–Шефера могут быть описаны в раках подхода к созданию байесовской теории на основе выпуклых множеств. Кроме того, создается впечатление, что теория Демпстера–Шефера сталкивается с затруднениями при решении задач, в которых значения степени доверия близки к нулю. Использование степеней доверия, равных нулю, приводит к получению результатов, весьма отличных от тех, которые соответствуют степеням доверия, близким к нулю. Еще одна проблема, связанная с использованием теории Демпстера–Шефера, обусловлена тем, что происходит экспоненциальный взрыв

объема вычислений по мере возрастания количества возможных ответов на диагностическую задачу.

Безусловно, с экспоненциальным взрывом позволяет справиться аппроксимация Гордона (Gordon) и Шортлиффа (Shortliffe), но применение этого метода может привести к некачественным результатам в случае использования в значительной степени конфликтующих свидетельств. Предложен также альтернативный подход, в котором не требуется аппроксимация и достигаются приемлемые результаты для иерархических свидетельств без комбинаторного взрыва. Опубликовано также множество других статей, в которых предпринимается попытка преодолеть проблему комбинаторного взрыва с помощью введения различных обобщений теории Демпстера–Шефера.

Наиболее важным положительным итогом всей этой работы стал полный пересмотр оснований теории вероятностей и расширение заинтересованности в использовании методов, позволяющих справляться с неопределенностью. Кроме того, разработан целый ряд гибридных подходов, в которых нечеткая логика используется в сочетании с искусственными нейронными системами, и многие эти подходы увенчались успехом [71]. Важно понять, что попытка заранее выбрать для своей работы какую-то конкретную модель неопределенности равносильна стремлению всегда использовать одну и ту же конкретную структуру данных обычного языка программирования, например, массив, связный список, дерево, очередь, стек и т.д. Всегда выбирайте модель, которая лучше всего подходит для решения рассматриваемой задачи.

5.7 Некоторые коммерческие приложения нечеткой логики

Количество коммерческих приложений нечеткой логики весьма велико. В их число входят многие бытовые приборы, начиная с видеокамер и заканчивая стиральными машинами. В приложении Ж приведены многочисленные ссылки с указанием информационных ресурсов, посвященных приложениям нечеткой логики.

- Автоматическое управление затворами плотин на **гидроэлектрических силовых установках** (Tokyo Electric Power).
- Упрощенное управление **роботами** (Hirota, Fuji Electric, Toshiba, Omron).
- **Наведение видеокамеры** во время телевизионной трансляции спортивных событий (Omron).
- Замена экспертов по **оценке активности фондовой биржи** (Yamaichi, Hitachi).
- Предотвращение нежелательных температурных изменений в **системах кондиционирования воздуха** (Mitsubishi, Sharp).

- Эффективное и стабильное управление **двигателем автомобиля** (Nissan).
- Автоматическое регулирование скорости **автомобиля** (Nissan, Subaru).
- Повышение эффективности и оптимизация функционирования **приложений управления производственными процессами** (Aptronix, Omron, Meiden, Sha, Micom, Mitsubishi, Nisshin-Denki, Oku-Electronics).
- Позиционирование степперов подложек в **производстве полупроводников** (Canon).
- Оптимизированное планирование **расписаний движения автобусов** (Toshiba, Nippon-System, Keihan-Express).
- Система архивирования **документов** (Mitsubishi Electric).
- **Система прогнозирования** для раннего распознавания землетрясений (Institute of Seismology Bureau of Metrology).
- **Медицинская технология** — диагностика раковых заболеваний (Kawasaki Medical School).
- Комбинирование нечеткой логики и **нейронных сетей** (Matsushita).
- Распознавание рукописных символов с помощью **карманных компьютеров** (Sony).
- Компенсация движения в видеокамерах (Canon, Minolta).
- Автоматическое управление двигателем **пылесоса** с применением средств распознавания состояния поверхности и степени загрязнения (Matsushita).
- Управление задней подсветкой **видеокамеры** (Sanyo).
- Компенсация колебаний **видеокамеры** (Matsushita).
- Управление **стиральными машинами** с помощью одной кнопки (Matsushita, Hitatchi).
- **Распознавание** почерка, объектов, голоса (CSK, Hitachi, Hosai University, Ricoh).
- Оказание помощи пилоту в управлении **вертолетом** (Sugeno).
- Моделирование **процессуальных действий** (Meihi Gakuin University, Nagoya University).
- **Проектирование программного обеспечения** для производственных процессов (Aptronix, Harima, Ishikawajima-OC Engeneering).
- Управление скоростью обработки и температурой на **сталелитейных заводах** (Kawasaki Steel, New-Nippon Steel, NKK).
- Управление **системами подземного транспорта** в целях улучшения комфортности поездки, повышения точности торможения и экономии электроэнергии (Hitachi).

- Улучшение регулирования расхода топлива в **автомобиле** (NOK, Nippon Denki Tools).
- Повышение чувствительности и эффективности системы **управления лифтом** (Fujitec, Hitachi, Toshiba).
- Повышение безопасности **ядерных реакторов** (Hitachi, Bernard, Nuclear Fuel Div.).

5.8 Резюме

В настоящей главе рассматривались неклассические вероятностные теории неопределенности. Для учета неопределенностей в экспертных системах применяется целый ряд методов, в том числе основанных на использовании коэффициентов достоверности, теории Демпстера–Шефера и теории нечетких множеств. Методы на основе коэффициентов достоверности несложны в реализации и успешно используются в таких экспертных системах, как MYCIN, в которых цепи логического вывода имеют небольшую длину. Но теория коэффициентов достоверности не базируется на надежном научном фундаменте, поэтому, вообще говоря, при использовании более длинных цепей логического вывода становится неприменимой.

Теория Демпстера–Шефера основана на строгом научном фундаменте и широко используется в экспертных системах.

Теория нечетких множеств — наиболее общая из всех теорий неопределенности, сформулированных до сих пор. Эта теория имеет очень широкую область применения благодаря возможности использовать принцип расширения. Теория нечетких множеств была впервые представлена в классической статье Заде в 1965 году и в дальнейшем стала основой многих новых научных направлений. В приложении Ж приведены многочисленные ссылки на дополнительную литературу по этой теме.

Задачи

5.1. Приняв предположение, что к первоначальному свидетельству E_1 добавляется свидетельство E_2, докажите следующее:

$$P(D_i \mid E_1 \cap E_2) = \frac{P(E_2 \mid D_i \cap E_1)P(D_i \mid E_1)}{\sum_j P(E_2 \mid D_j \cap E_1)P(D_j \mid E_1)}$$

Подсказка. Используйте результаты задачи 4.8, *б*.

5.2. Докажите следующее:

$$CF(H, E) + CF(H', E) = 0$$

5.3. Предположим, что даны следующие правила:

$$\begin{aligned}&\text{IF } E_1 \text{ AND } E_2 \text{ AND } E_3\\ &\quad\text{THEN } H(CF_1)\\ &\text{IF } E_4 \text{ OR } E_5\\ &\quad\text{THEN } H(CF_2)\end{aligned}$$

В этих правилах:

$$\begin{aligned}&CF_1(E_1, e) = 1 \qquad CF_1(E_2, e) = 0.5 \qquad CF_1(E_3, e) = 0.3\\ &CF_2(E_4, e) = 0.7 \qquad CF_2(E_5, e) = 0.2\\ &CF_1(H, E) = 0.5 \qquad CF_2(H, E) = 0.9\end{aligned}$$

а) Начертите дерево, которое показывает, каким образом эти правила обосновывают гипотезу H.

б) Вычислите коэффициенты достоверности $CF_1(H, e)$ и $CF_2(H, e)$.

в) Вычислите значение $CF_{\text{COMBINE}}[CF_1(H, e), CF_2(H, e)]$.

5.4. Выполните перечисленные ниже задания.

а) Приняв предположение, что на рис. 5.7, *в* заданы следующие значения:

$$\begin{aligned}m(X) &= 0.2\\ m(Y) &= 0.3\\ m(Z) &= 0.5\end{aligned}$$

найдите интервалы проявления свидетельств X, Y и Z с использованием теории Демпстера–Шефера.

б) Приняв предположение, что на рис. 5.7, *г* заданы следующие значения:

$$\begin{aligned}m(X) &= 0.4\\ m(Y) &= 0.6\end{aligned}$$

найдите интервалы проявления свидетельства для таких выражений:

$$\begin{aligned}&X\\ &X \cap Y\\ &Y\\ &X \cap Y'\\ &X' \cap Y\end{aligned}$$

5.5. Предположим, что даны следующие правила:

$$\text{Rule 1: IF } E \text{ THEN } H$$
$$\text{Rule 2: IF } E \text{ THEN } H'$$

а также предположим, что

$$\Theta = \{H, H'\}$$
$$m_1(H) = 0.5 \qquad m_1(\Theta) = 0.5 \quad \text{для правила Rule 1}$$
$$m_2(H') = 0.3 \qquad m_2(\Theta) = 0.7 \quad \text{для правила Rule 2}$$

а) Составьте таблицу Демпстера–Шефера, в которой показаны комбинации свидетельств, и вычислите комбинированные доверительные функции.

б) Вычислите значения степеней правдоподобия.

в) Вычислите интервалы проявления свидетельства.

г) Вычислите значения степеней сомнительности.

д) Вычислите значения степеней незнания.

5.6. В табл. 6 приведены значения степеней доверия, полученные от датчиков разных типов в среде распознавания типов летательных средств, в которой рассматриваются авиалайнеры (airliner — A), бомбардировщики (bomber — B) и истребители (fighter — F).

Таблица 5.20. Данные, полученные от датчиков в среде распознавания типов летательных средств

Фокальные элементы	**Датчик 1 (m_1)**	**Датчик 2 (m_2)**
$\varnothing$	0.15	0.2
A, B	0.3	0.1
A, F	0.1	0.05
B, F	0.1	0.1
A	0.05	0.3
B	0.2	0.05
F	0.1	0.2

а) Вычислите начальные доверительные функции, значения степеней правдоподобия, интервалы проявления свидетельства, значения степеней сомнительности и незнания.

б) Вычислите те же самые параметры после комбинирования свидетельств.

5.7. Полицейский останавливает автомобилиста за превышение скорости. Направленная радарная установка полицейского и спидометр автомобиля допускают ошибки, поэтому доверительные функции имеет вид, показанный в табл. 7.

Таблица 5.21. Значения доверительных функций

Радар полицейского	Спидометр автомобиля
$m_1(57) = 0.3$	$m_2(56) = 0.2$
$m_1(56) = 0.5$	$m_2(55) = 0.6$
$m_1(55) = 0.2$	$m_2(54) = 0.2$

а) Вычислите доверительные функции, значения степеней правдоподобия, интервалы проявления свидетельства, значения степеней сомнительности и незнания для каждого из заинтересованных лиц.

б) Вычислите эти параметры после комбинирования свидетельств.

в) Объясните, почему на основании этих параметров можно утверждать, что автомобилист не превысил скорость.

5.8. Предположим, что даны следующие нечеткие множества:

$$A = .1/1 + .2/2 + .3/3 B = .2/1 + .3/2 + .4/3$$

Для получения ответов на следующие вопросы проведите необходимые вычисления и дайте свои объяснения.

а) Равны ли эти множества? Объясните.

б) Определите результат операции дополнения множеств.

в) Определите результат операции объединения множеств.

г) Определите результат операции пересечения множеств.

д) Соблюдается ли для каждого множества закон исключенного третьего? Объясните.

е) Определите результат операции произведения множеств.

ж) Определите результат операции возведения каждого множества во вторую степень.

з) Определите результат операции вероятностной суммы.

и) Определите результат операции ограниченной суммы.

к) Определите результат операции ограниченного произведения.

л) Определите результат операции ограниченной разности.

м) Определите результат операции концентрации.

н) Определите результат операции растворения.

о) Определите результат операции интенсификации.

п) Определите результат операции нормализации.

5.9. Предположим, что даны следующие нечеткие множества:

$$Q = \begin{bmatrix} 0.2 & 0.3 \\ 0.4 & 1.0 \end{bmatrix} \text{ определено на } U_1 \times U_2$$

$$P = \begin{bmatrix} 0.1 & 0.5 & 0.3 \\ 0.2 & 0.0 & 0.4 \end{bmatrix} \text{ определено на } U_2 \times U_3$$

а) Вычислите первую, вторую и общую проекции для каждого множества.

б) Вычислите цилиндрические расширения для каждого множества.

в) Покажите, что имеет место следующее:

$$Q \circ P = \text{proj}(\overline{Q} \cap \overline{P}; U_1 \times U_3)$$

5.10. Рассмотрим лингвистическую переменную Person как нечеткое множество порядка 3.

а) Определите Person как три нечетких множества порядка 2.

б) Определите каждое из множеств второго порядка в терминах трех нечетких множеств порядка 1.

в) Определите три из нечетких множеств первого порядка в терминах S- и (или) $\prod$-функций.

5.11. Выполните перечисленные ниже задания.

а) Определите пять лингвистических значений для лингвистической переменной Uncertainty.

б) Составьте соответствующие функции для этих значений и объясните, чем обусловлен ваш выбор.

в) Составьте нечеткие множества для следующих случаев, приняв предположение, что множество TRUE можно представить с помощью S-функции:

Not TRUE

More Or Less TRUE

Sort Of TRUE

Pretty TRUE

Rather TRUE

TRUE

Каковы пределы нечеткого множества TRUE? Объясните.

5.12. Выполните перечисленные ниже задания.

а) Определите по меньшей мере шесть значений для лингвистической переменной Water Temperature (Температура воды).

б) Нарисуйте на одной диаграмме графики соответствующих функций для значений нечеткого множества.

в) Задайте три функции барьерных нечетких множеств на основе понятия FREEZING (Замерзание).

5.13. Выполните перечисленные ниже задания.

а) Покажите с помощью истинностной таблицы для $N = 2$ и $N = 3$ значения примитивных операций L-логики, приведенных в табл. 5.14.

б) Определите значение $x \leftrightarrow y$ в терминах абсолютных значений x и y.

5.14. Предположим, что даны следующие числовые истинностные значения:

$$x(A) = .2/.1 + .6/.5 + 1/.9$$
$$x(B) = .1/.1 + .3/.5 + 1/.9$$

Вычислите истинностные значения в нечеткой логике для указанных ниже выражений.

а) NOT A.

б) A AND B.

в) A OR B.

г) $A \rightarrow B$.

д) $B \rightarrow A$.

5.15. Определите с помощью барьерных первичных термов и словосочетаний, задающих область значений, такую нечеткую грамматику, которая позволяла бы вырабатывать продукционные правила, приведенные ниже.

IF PRESSURE IS HIGH THEN TURN VALVE LOWER
IF PRESSURE IS VERY HIGH THEN TURN VALVE MUCH LOWER
IF PRESSURE IS VERY VERY HIGH THEN TURN VALVE MUCH MUCH LOWER
IF PRESSURE IS LOW TO MEDIUM THEN TURN VALVE HIGHER

Примите следующие предположения: во-первых, первичными термами для множества PRESSURE (Давление) являются только LOW (Низкое) и HIGH

(Высокое); во-вторых, словосочетания, задающие области значений, в которых применяется слово ТО (От ... до ...), встречаются только в антецеденте, и, в-третьих, первичными термами для множества VALVE (Вентиль), применяемого в заключении, могут быть только LOWER (Ниже) и HIGHER (Выше).

Глава 6

Проектирование экспертных систем

6.1 Введение

Предыдущие главы были посвящены рассмотрению наиболее важных концепций, а также изложению теоретических основ экспертных систем и других интеллектуальных систем принятия решений. Общие сведения о свойствах интеллектуальных систем приведены во многих статьях. Такие системы подразделяются на несколько типов, и каждый из них характеризуется своими преимуществами и недостатками [3]. В настоящей главе приведены основные рекомендации по созданию практически применимых экспертных систем, предназначенных для реализации реальных приложений, а не исследовательских прототипов. Кроме того, описана **методология разработки программного обеспечения**, благодаря использованию которой экспертная система может стать высококачественным продуктом, разработанным в заданные сроки с применением экономически эффективных методов.

Проектирование экспертных систем рассматривается в многочисленных книгах и статьях, в которых все аспекты проектирования описаны очень подробно. Кроме того, весьма велико количество программных инструментальных средств, предназначенных для проектирования. Поэтому невозможно сразу же стать экспертом, прочитав только одну главу. (Тем не менее, не подлежит сомнению, что предварительное прочтение данной главы является обязательным первым шагом к указанной цели.) Однако понимание принципов, которые могут служить объяснением того, почему экспертные системы проектируются именно так, а не иначе, позволит более полно воспользоваться преимуществами сложных инструментальных средств и методологий. Проектирование экспертных систем фактически явля-

ется частью более общей задачи, называемой **управлением знаниями** (Knowledge Management — KM). Эта задача относится к управлению всеми активами знаний, доступными конкретной организации [2].

Управление знаниями связано с управлением информацией (Information Management — IM). Последнее научное направление связано со средствами обработки информации (Information Processing — IP), которые связаны с информационными системами (Information Systems — IS), связанными, в свою очередь, с информационной технологией (Information Technology — IT). (Закрадывается мысль, что тот, кто сможет три раза подряд произнести, не сбиваясь, слова "управление знаниями, управление информацией, обработка информации, информационная технология", уже может считать себя экспертом!) Управление знаниями является основной задачей, касающейся всевозможных типов информационных ресурсов для многих разных аудиторий, таких как управленческие работники; пользователи Web; люди, желающие получить ответы на часто задаваемые вопросы (Frequently Asked Questions — FAQ); люди, обращающиеся в справочные бюро (оснащенные кадровым персоналом или автоматические); пользователи электронной почты; абоненты факсимильной связи, абоненты телефонной связи; потребители товаров; программисты; разработчики; руководители; конечные пользователи и вся широкая общественность в целом.

Крупные компании используют собственные весьма детализированные методы, документы, книги и программы, предназначенные для создания, управления, сопровождения и реализации активов знаний, поскольку эта область является весьма прибыльной [17]. Это направление деятельности особенно оправдывается, если создаются такие интеллектуальные инструментальные средства, которые позволяют значительно сократить затраты на наем рабочей силы, поскольку при этом удается снижать издержки даже больше чем при использовании глобального аутсорсинга.

Анализ затрат времени типичного пользователя, работающего на компьютере, показывает, что при управлении всеми документами, созданными с помощью компьютеров, невозможно обойтись без компьютеров (невольно приходится прибегать к тавтологиям). Экспертные системы широко используются на деловых предприятиях именно потому, что применение этой технологии является крайне важным в условиях стремительного увеличения объема информации и знаний, доступных в Web [70]. Как обычно, ссылки на программное обеспечение и другие оперативные ресурсы для данной главы перечислены в приложении Ж.

6.2 Выбор соответствующей задачи

Для создания системы, основанной на знаниях, или экспертной системы можно воспользоваться многими способами и ресурсами. Но непродуктивные усилия

в этом направлении — один из лучших способов напрасно израсходовать свое время и деньги (не считая, впрочем, предоставления дотаций “Фонду голодающих авторов и технических редакторов”; кстати, эта организация является коммерческой, т.е. ей удается успешно зарабатывать деньги). Однако **четвертый закон успеха** гласит, что нельзя отправляться в путешествие, не зная, куда вы собираетесь прибыть.

Как было описано в разделе 1.6, прежде чем приступить к созданию экспертной системы, необходимо выбрать для нее подходящую задачу. Как и при реализации любого программного проекта, необходимо рассмотреть целый ряд общих соображений, и только после этого брать на себя существенные расходы по созданию предлагаемой экспертной системы, связанные с трудозатратами, потреблением ресурсов и времени. Такие же общие соображения, как правило, возникают в ходе руководства проектом по созданию обычных программ, но реализация проектов экспертных систем связана с учетом дополнительных требований. На рис. 6.1 представлена весьма обобщенная схема действий, осуществляемых при управлении проектом. Как показано на этом рисунке, общая задача управления проектированием подразделяется на три общих этапа — управление текущей деятельностью, управление настройкой конфигурации программного продук-



Рис. 6.1. Задачи руководства проектированием

та и управление ресурсами, и каждый из этих этапов предъявляет свои конкретные требования. В настоящей главе задачи, решаемые на каждом из указанных этапов, будут обсуждаться в форме вопросов и ответов в целях предоставления общих рекомендаций по созданию проектов экспертных систем.

Выбор наиболее приемлемого подхода

На этом этапе осуществляется поиск ответа на вопрос, для чего предназначена экспертная система.

По-видимому, это — наиболее важный вопрос, на который необходимо найти ответ, прежде чем приступать к реализации любого проекта. Наиболее способствующим успеху ответом является то, что создание экспертной системы осуществляется по желанию председателя совета директоров компании. Если же дело обстоит иначе, то необходимо определить, можно ли в данном случае надеяться на реализацию общих преимуществ экспертных систем, которые описаны в разделе 1.2 главы 1. Самое главное, не следует забывать, что лишь руководители компании имеют право выделить для создания системы необходимые ресурсы и технический персонал. Если же вы примете решение создать такую систему за счет своего личного времени, чтобы доказать свою правоту руководству компании, а затем поймете, что у вас получился великолепный программный продукт, и захотите уволиться и образовать собственную компанию, то не забудьте о небольшом нюансе — о подписанном вами **соглашении по охране интеллектуальной собственности** (Intellectual Property Agreement — IPA). В большинстве соглашений по охране интеллектуальной собственности указано, что все придуманное вами, будь то связанное или не связанное с вашими должностными обязанностями (т.е. охваченное периодом времени $24 \times 7 \times 52$ — 24 часа, 7 дней, 52 недели), принадлежит компании. В частности, ответ на вопрос о том, для чего создается экспертная система, должен быть в конечном итоге предоставлен владельцам или акционерам, от которых зависит финансирование всех разработок. Еще до начала работы эти заинтересованные лица должны ознакомиться с результатами четкого определения задачи, выбора эксперта и состава пользователей.

Обратите внимание на то, что речь идет об эксперте, указанном в единственном числе, а не об экспертах. Известно, что лучший способ обеспечить себе неприятности — вступить в брачные отношения одновременно с несколькими людьми. Привлечение к работе сразу нескольких экспертов также полностью гарантирует неприятности. Даже сам господин председатель совета директоров компании посещает одновременно только одного врача, причем это не связано с тем, что у него не хватает денег. Дело в том, что не рекомендуется пытаться моделировать экспертные знания нескольких экспертов в одной системе. Но попытка промоделировать экспертные знания нескольких экспертов в нескольких системах с использованием архитектуры классной доски для выработки по меньшей мере

общего мнения большинства вполне оправдана. Однако и в этом случае приходится задумываться над тем, что означает нечеткий термин “оправданная попытка”. Если девять из десяти врачей скажут, что пациент умрет, этот несчастный непременно выберет врача, предлагающего такое лечение, которое, вероятно, позволит остаться ему в живых (этого не произойдет, только если пациент действительно беден, а лечение стоит больших денег). Задача, связанная с осуществлением попытки решить, какую значимость следует придать мнениям многочисленных экспертов, является весьма утомительной. Обычно при этом приходится сталкиваться с еще бóльшими затруднениями, чем при создании одной экспертной системы для одного эксперта. Поэтому в настоящей главе мы будем придерживаться подхода “одна система и один эксперт” и оставим разработку на основе подхода “несколько экспертов и несколько систем” в качестве учебного проекта.

Выигрыш

На данном этапе необходимо определить, какой должен быть достигнут *выигрыш*.

Текущий вопрос связан с предыдущим вопросом, но является более прагматичным, поскольку в рассматриваемом случае поиск ответа требует определения конкретной отдачи от вложенных трудозатрат, ресурсов, времени и денег. Выигрыш может измеряться в деньгах, оцениваться в показателях повышения эффективности или устанавливаться с помощью любых других критериев, определяющих преимущества экспертных систем, которые описаны в главе 1. Важно также помнить, что если система не будет эксплуатироваться, то не будет и никакого выигрыша. Поиск ответа на указанный вопрос применительно к экспертным системам является более сложным, чем по отношению к обычному программированию, поскольку между экспертными системами и обычными программами имеются существенные различия.

Инструментальные средства

На данном этапе необходимо найти ответ на вопрос о том, какие инструментальные средства создания систем находятся в вашем распоряжении.

Как показано в приложении Ж, в настоящее время имеется много инструментальных средств экспертных систем, обладающих своими преимуществами и недостатками. Но указанная информация должна рассматриваться только в качестве общего руководства, поскольку программные инструментальные средства развиваются слишком быстро. В целом можно вполне рассчитывать на то, что через каждые два-три года будет происходить усовершенствование каждого инструментального средства, а примерно через пять лет будут существенно пересматриваться его возможности. Безусловно, больше денег можно заработать, выпуская доработанное программное обеспечение под новым именем в качестве новой вер-

сии, чем вносить существенные исправления, связанные с устранением ошибок, в старые программы. Те, кто сомневаются в истинности этого утверждения, могут посчитать количество новых имен для одной и той же операционной системы, которые появились на рынке в течение последних десяти лет.

Наилучшая рекомендация по выбору инструментального средства для экспертной системы следует из **третьего закона успеха**: если вы не в состоянии сделать выбор сами, обратитесь за помощью к другим. Проведите поиск в Web, чтобы узнать, какие приложения были созданы с использованием рассматриваемого программного обеспечения. Постарайтесь найти примеры не только успешного создания систем, но и примеры, в которых такая попытка окончилась неудачей. Безусловно, трудно подготовить точные статистические данные, но согласно большинству оценок относительное количество успешных программных реализаций достигает лишь 20–30%. Проблема создания системы дополнительно усложняется, если вы вынуждены работать с очень занятым экспертом, а сами вы не знакомы с терминами рассматриваемой проблемной области.

Поэтому перейдем к рассмотрению **второго закона успеха**: прежде чем обратиться к кому-либо за помощью, изучите язык, на котором вам сообщат ответ. Несмотря на большое количество доступных переводчиков с иностранного языка (людей и программ), трудность состоит не в том, чтобы просто перевести с одного языка на другой, а в том, чтобы правильно передать термины, относящиеся к данной области знаний.

Но недостаточно лишь изучить термины или запомнить словарные определения ключевых слов, чтобы создать семантическую сеть отношений, на которой базируется экспертная система, основанная на правилах. Безусловно, каждое правило представляет собой определенный фрагмент знаний, но нельзя забывать о том, что между правилами в базе знаний могут существовать и сильные, и слабые связи. Если запуск одного правила является гарантией запуска другого правила, то имеет место **сильная связь**, или, иными словами, в цепи логического вывода, которая ведет от действительных фактов к действительному заключению, имеется сильное звено. Если же запуск одного правила приводит к обнаружению нескольких правил, доступных для активизации машиной логического вывода, то имеет место **слабая связь**. Это означает, что цепь логического вывода становится менее прочной; нельзя со всей уверенностью сказать, по какому пути будут развиваться дальнейшие события, поскольку таких путей много.

Наличие большего количества правил, которые в принципе могут быть активизированы, показывает, что цепь логического вывода является не такой прочной, как при наличии сильной связи. Но такая ситуация отнюдь не является неблагоприятной. Если каждое правило действительно сильно связано с другим правилом, то не нужна и машина логического вывода или экспертная система, поскольку для реализации подобных связей идеально подходит любой процедурный язык программирования. Программы на процедурном языке программирования

выполняются последовательно, оператор за оператором, в порядке их ввода, до тех пор, пока не обнаруживаются такие структуры, предназначенные для принятия управленческих решений, как проверки условий IF. Если имеется алгоритм, позволяющий эффективно решать какие-то задачи, то нет необходимости решать эти задачи с помощью искусственного интеллекта. В идеале экспертная система должна представлять собой сбалансированную смесь правил, обеспечивающих создание сильных и слабых связей, по аналогии с тем, как люди для достижения обоснованного заключения используют дедуктивные, индуктивные, вероятностные и другие методы.

Рассмотрим следующий **семантический список** терминов из некоторых областей человеческой деятельности. В данном контексте используется термин "семантический список", поскольку рассматриваемые термины невозможно соединить с помощью графа. Вместо этого наблюдается такая картина, что одни понятия естественным образом следуют из других. (Следует отметить, что с возрастом люди начинают все лучше понимать смысл терминов из этого общего списка!) Стоит только представить себе, какой жизненный опыт нужно накопить, чтобы не только полностью понимать следующие выражения и описания ситуаций, но и оценивать их подтекст: фильм, который должен был выйти раньше; запоздалые известия; штрафы за просрочку; платежи со счета прекращены; счет закрыт; агентство по взысканию долгов; назойливые телефонные звонки ("Вы дома? Ответьте нам. Мы знаем, что вы — дома, поскольку вас нет на работе"); резкое увеличение количества спама, поступающего по электронной почте; плохая кредитная история; низкая кредитоспособность; отсутствие кредита; предложения, от которых не следует ждать хорошего ("Мы хотим вас финансировать"); наличие задолженности; невыплаченные налоги; налоги плюс проценты по налогам; судебные издержки; потеря права выкупа; распродажа имущества с молотка; лоббист; содействие в проведении избирательной кампании; поправка к финансовому законопроекту; извинения от судебных властей; извинения от руководителей компании; вознаграждение за моральный ущерб; покупка контрольного пакета акций компании видеопроката с помощью кредита.

Семантические списки часто ассоциируются с другими **знаками**. Знаки и их смысловые значения изучаются в области **семиотики**. Приведем пример, знакомый всем, кто в детстве распевал "В лесу родилась елочка. . . ". Слова этой песенки неразрывно связаны с ее музыкальным сопровождением, поскольку в песнях воплощается семиотическое отношение между стихами и музыкой. Семиотические отношения постоянно используют рекламодатели, повторяя одни и те же известные телевизионные музыкальные позывные. Некоторые создатели музыкальных позывных настолько преуспели в своем деле, что от составленных ими простеньких мелодий буквально невозможно отвязаться целыми сутками; несмотря на все усилия, эти бессмысленные музыкальные фразы снова и снова повторяются у вас в голове.

Прежде чем получить интервью у эксперта, также необходимо применить безотказный способ — подготовить письменный протокол собеседования. В дальнейшем, читая заполненный протокол, вы сможете уловить семантические связи между терминами. Ваша задача состоит именно в этом. Термины, представляющие знания, не существуют отдельно; они образуют семантические кластеры, которые соединяются, образуя общую семантическую сеть знаний, представленную в уме эксперта. Удачным примером ассоциативной семиотической памяти является само мышление, поскольку в процессе мышления каждое отдельное слово, запах, прикосновение, зрительный или звуковой образ могут вызвать переход от одних воспоминаний к другим.

Классические инструментальные средства извлечения знаний с помощью стандартных интервью и результаты важных исследований, проведенных в данной области, описаны в [76]. В действительности методы проведения собеседований обычно дополняются приемами вычерчивания схем, позволяющих уточнить рассматриваемые понятия и отношения. Графические схемы служат в качестве полезных визуальных метафор, поскольку и разработчики, и эксперты могут одновременно рассматривать представленные на них отношения и анализировать переданный тем самым смысл, достигая согласия в отношении того, что и как должно быть представлено на этих схемах. Кроме того, имеются такие программные инструментальные средства, которые позволяют представить сразу всю онтологию. Как было описано в главе 1, онтология — это полное описание проблемной области, представленное с помощью формализованного способа.

Одним из инструментальных средств, результаты применения которого оказались очень успешными, является Protégé — (редактор онтологий и баз знаний). Это чрезвычайно сложное инструментальное средство предоставляется бесплатно Станфордским университетом. На Web-узле указанного университета можно получить качественную поддержку этого инструментального средства; там же имеются учебники, с помощью которых онтологию может создать даже начинающий, а также приведены многие другие материалы. Система Protégé используется вместе с TixClips — интегрированной средой разработки для языка CLIPS. При создании очень крупных систем, состоящих из тысяч правил, практически невозможно обойтись без онтологий.

Стоимость

На этом этапе необходимо определить возможный объем расходов.

Такая задача является не менее сложной по сравнению с поиском ответа на вопрос о том, во сколько может вам обойтись расторжение брака. Это — далеко не те заранее указанные расходы, которыми сразу же стремятся вас заинтересовать авторы поступающего по электронной почте спама, где речь идет о том, как добиться развода без помощи адвоката или сделать самому себе операцию без по-

мощи хирурга. Дело в том, что расходы связаны не только с созданием экспертной системы, но и с ее эксплуатацией. На этапе создания экспертной системы приходится нести издержки, зависящие от того, сколько людей, ресурсов и времени фактически выделено на решение этой задачи. Кроме того, как и при эксплуатации любого другого программного или аппаратного компонента, необходимо учитывать стоимость сопровождения. Тем самым мы приходим к **первому закону успеха**: если владелец автомобиля не хочет беспокоиться о том, во сколько ему обойдется бензин, то он должен приобрести собственную нефтяную скважину. С этим же связан **нулевой закон успеха**: лучше всего иметь возможность пользоваться чужим автомобилем и бензином. Кроме того, если больше нет возможности прибегать к услугам того эксперта, с помощью которого создана экспертная система, значительно усложняется и сопровождение самой экспертной системы.

Привлечение другого эксперта не окажет значительной помощи, поскольку эксперты весьма предвзято относятся к мнению друг друга. Поэтому гораздо проще создать такую систему, основанную на знаниях, в которой знания получены от многих людей, из документов, а также из других общедоступных источников. Такие системы были созданы во многих компаниях, поскольку практика показала, что подобные системы являются полезным инструментальным средством предварительного консультирования, которое позволяет избавиться от 90% проблем, найдя решение с помощью всего лишь нескольких вопросов. Гораздо проще предоставить клиентам возможность работать с подобной системой, чем содержать целый телефонный центр предоставления консультаций.

Безусловно, для эксплуатации инструментария экспертных систем требуются аппаратные и программные средства, но компаниям приходится также брать на себя существенные затраты, связанные с обучением. Если персонал компании имеет мало опыта работы с каким-то инструментальным средством или вообще не имеет такого опыта, то обучение может потребовать существенных расходов. Профессиональные курсы обучения, предназначенные для освоения любого программного обеспечения, часто проводятся с учетом затрат порядка 2500 долларов в неделю на одного человека. Но, разумеется, больше никто не рассматривает такую необходимость просто как обузу. Теперь компании могут зарабатывать намного больше денег, чем раньше, предоставляя возможность пройти обучение для работы с их программными продуктами с отрывом и без отрыва от производства; сдать экзамены на получение документа об образовании; пройти подготовку к экзаменам на получение документа об образовании; стать инструктором для проведения экзаменов на получение документа об образовании; а также издавая книги, проводя семинары и практикумы, наконец, предоставляя обслуживание на дому. В наши дни выпуск программного продукта, который является мощным, удобным и не содержащим ошибок, — это путь к экономическому краху.

6.3 Общее описание процесса разработки экспертной системы

На данном этапе необходимо найти ответ на вопрос о том, как должна разрабатываться экспертная система.

Сам ход разработки экспертной системы в значительной степени зависит от того, какие для этого предусмотрены ресурсы. Тем не менее, как и при реализации любого проекта, успех разработки зависит также от правильной организации и управления процессом. Полезным руководством, материалы которого основаны на результатах разработки экспертной системы для Федеральной администрации скоростных дорог (Federal Highway Administration), в которой использовалось свыше 300 правил, является [93]. Если вы уже знакомы с каким-то стандартным инструментальным средством планирования проектов, допустим Microsoft Project, то вам не потребуется искать другое инструментальное средство. К тому же, если вы будете применять подобный планировщик, то вам будет проще объяснить руководству, как идут дела и почему проект немного отстает от графика (поскольку в вашем распоряжении в любой момент будут новейшие данные).

Руководство проектом

Предполагается, что стандартные методы управления проектом и программные инструментальные средства должны обеспечивать решение перечисленных ниже задач.

Управление текущей деятельностью

- Планирование:
  - определение состава действий;
  - назначение действиям приоритетов;
  - выявление требований к ресурсам;
  - выделение основных этапов;
  - определение значений продолжительности этапов;
  - распределение обязанностей.
- Составление графика работ:
  - определение значений времени начала и окончания работ;
  - разрешение конфликтов при включении в график задач с равным приоритетом.
- Текущий контроль:
  - контроль за ходом выполнения проекта.

- Анализ:
  - анализ планов, графиков и результатов осуществления проекта.

Управление конфигурацией программного продукта

- Управление обновлением программного продукта:
  - организация выпуска различных версий программного продукта.
- Управление процессом внесения изменений:
  - контроль над предложениями о внесении изменений и учет возможных последствий реализации предлагаемых изменений;
  - назначение ответственных за проведение изменений;
  - инсталляция новых версий программного продукта.

Управление ресурсами

- Прогнозирование потребности в ресурсах.
- Мобилизация ресурсов:
  - поиск наиболее подходящего эксперта или базы знаний;
  - корректировка графика с учетом пожеланий эксперта.
- Распределение обязанностей по оптимизации использования ресурсов.
- Предоставление ресурсов, необходимых для устранения узких мест.

В частности, следует отметить, что в отношении приобретения ресурсов отмечена необходимость согласовывать свое расписание работы с расписанием эксперта, но не требовать этого от эксперта (если только эксперт не привлечен к работе на постоянной основе). У экспертов есть другие обязанности, и их время ценится очень высоко.

На рис. 6.2 показано идеализированное и обобщенное представление перечня работ, требуемых для создания экспертной системы. Эти работы описаны в составе этапов, через которые должна пройти система в процессе своего развития.

Согласно этому идеализированному представлению, конечный продукт не передается в распоряжение конечных пользователей до тех пор, пока не будут устранены все ошибки. Такое требование действительно оправдывает себя, поскольку, например, если экспертная система предназначена для использования в военном деле, то допущенные ею ошибки чаще всего невозможно исправить (скажем, вернуть в исходное положение ракету с ядерной боеголовкой после ее пуска).

Правительственные контракты допускают возможность регулярной и гарантированной выплаты денег, а в мире частного предпринимательства в случае задержки при выполнении проекта смета может быть исчерпана. Поэтому коммерческие предприятия в настоящее время придерживаются такого подхода — в течение



Рис. 6.2. Общие этапы разработки экспертной системы

каждого квартала затрачивают определенные средства на рекламу будущего продукта, даже если окончательный выпуск этого продукта произойдет лишь через несколько лет. Такой подход позволяет достичь двух целей. Во-первых, частные компании, занимающиеся разработками, которые стремятся использовать будущий продукт в создаваемых ими продуктах, получают возможность приобретать предварительные версии даже при наличии в них известных ошибок. В результате этого появляется постоянный источник дохода от разрабатываемого продукта, поэтому компании не приходится самой нести все расходы на разработку в течение нескольких лет. Во-вторых, компания получает значительное преимущество, связанное с тем, что необходимость внутрифирменного тестирования продукта намного уменьшается (или вообще отпадает).

В этом случае компании, занимающиеся разработками, могут предусмотреть такую возможность, чтобы после каждого сбоя в работе создаваемого программного продукта поступал запрос к пользователю, желает ли он автоматически отправить отчет с описанием ошибки. Если пользователь щелкнет на кнопке OK, будет автоматически передано сообщение с диагностической информацией, касающейся того, где возникло нарушение в работе. А автоматизированная программа электронной почты в компании-разработчике сохранит полученное сообщение об ошибке в базе данных. Затем можно применить стандартные инструментальные средства анализа скрытых закономерностей в данных для поиска необходимой информации и передачи извещений конкретной группе разработчиков, ответственной за определенный аспект функционирования продукта. Такой подход позволяет устанавливать различные уровни реагирования, для того чтобы извещения передавались в группы разработчиков только после регистрации количества ошибок, превышающего установленный уровень реагирования. Может оказаться так, что некоторым группам поручена более важная работа, чем другим, поэтому уровни реагирования не обязательно должны быть одинаковыми. Таким образом, появляется возможность выделить требуемые ресурсы для исправления выявленных ошибок ко времени следующего ежеквартального выпуска, с учетом количества и серьезности ошибок. Для сопровождения предварительных версий должен быть предусмотрен специальный Web-узел, на котором пользователь мог бы вносить плату за подписку для получения доступа к каждой новой ежеквартальной предварительной версии.

При использовании рассматриваемого подхода необходимо учитывать еще одно, более важное требование, состоящее в том, что объявления о новом ежеквартальном выпуске должны быть согласованы по времени с ежеквартальными отчетами перед акционерами (владельцами привилегированных акций) согласно требованиям Комиссии по ценным бумагам и биржам (Securities and Exchange Commission — SEC) к открытым акционерным обществам. Даже если компания — разработчик все еще является закрытым акционерным обществом и не выпустила изначальное открытое предложение (Initial Public Offering — IPO) по приобретению акций, такой шаг позволит привлечь венчурный капитал в компанию, сопровождающую Web-узел с анонсами продукта, и увеличить стоимость изначального открытого предложения.

Проблема доставки

На этом этапе необходимо найти ответ на вопрос о том, как должна осуществляться доставка готовой системы пользователю.

Если на основе разрабатываемого инструментального средства должно быть создано большое количество экспертных систем, то важным фактором разработки может стать **проблема доставки** разрабатываемых систем. Поэтому проблеме

доставки необходимо уделять внимание даже на самых ранних этапах разработки. В XX столетии системы, применявшиеся для разработки, стоили значительно больше по сравнению с системами, на которых должен был установлен продукт, предоставляемый конечным пользователям. А в наши дни стоимость компьютерной техники значительно уменьшилась, поэтому данный аспект проблемы доставки разрешен.

Тем не менее остается актуальной другая проблема — необходимость обеспечить эксплуатацию программного продукта с использованием весьма разнообразного перечня аппаратных средств. Один из подходов к решению этой проблемы воплотился в создании языка Java, который сразу же приобрел широкую известность благодаря реализованному в нем принципу "обеспечения эксплуатации единожды созданной программы в любой среде"; этот принцип гарантировал также возможность использования аплетов Java только в "песочнице" (т.е. в защищенной и ограниченной среде). Проблема развертывания программного обеспечения с применением аппаратных средств различных типов все еще остается актуальной, если не предусматривается возможность использования интерпретатора или установки виртуальной машины в каждой отдельной разновидности аппаратных средств. С того времени, как впервые были предложены интерпретаторы байт-кода для языка Pascal в 1970-х годах, позволяющие эксплуатировать программы Pascal на любом микрокомпьютере, этот подход постоянно себя оправдывал, поэтому был реализован и в языке Java.

В языке CLIPS не предусмотрена выработка байт-кодов, но предоставляется возможность создания готовой версии экспертной системы на стандартном языке C. Готовая версия может быть откомпилирована с помощью стандартного компилятора C для конкретной аппаратной платформы, после чего полученный исполняемый файл развертывается для эксплуатации на данной конкретной платформе. Это означает, что какая-либо "виртуальная машина CLIPS" не требуется. Одной из первоначальных целей проекта создания языка CLIPS была реализация возможности использования этого языка для формирования экспертных систем, которые могут быть легко развернуты с применением любых новых аппаратных средств, которые когда-либо будут созданы. Как указано в предисловии, если основная версия CLIPS не позволяет справиться с этой задачей, то можно воспользоваться другими специально разработанными версиями, такими как Jess. Язык Jess написан на языке Java и обладает всеми преимуществами Java, поскольку легко обеспечивает возможность развертывания с использованием различных аппаратных средств (хотя и не поддерживает множественное наследование).

Во многих случаях возникает необходимость интегрировать экспертную систему с другими существующими программами. Это означает, что приходится учитывать необходимость обеспечения связи и координации входных и выходных потоков данных экспертной системы с указанными программами. Может также потребоваться, чтобы существовала возможность вызывать экспертную систе-

му как процедуру из программы на обычном языке программирования, поэтому система должна поддерживать подобную возможность. Язык CLIPS разработан таким образом, чтобы программы на этом языке можно было вызывать из программ на **базовом языке**, таком как C++. В таком случае вызываемые программы должны выполнить свои функции, а затем возвратить управление в программу на базовом языке. Создание подобных гибридных систем является также одним из основных подходов к разработке систем на языке CLIPS, поскольку производительность экспертной системы в значительной степени зависит от количества правил, заданных в системе, и от объема работы по согласованию с шаблонами. Безусловно, rete-алгоритм позволяет успешно решать многие задачи, но никакая интерактивная экспертная система не может сравниться по своему быстродействию с оттранслированной программой.

Разумеется, с помощью программы на языке CLIPS можно промоделировать работу калькулятора, электронной таблицы или даже текстового процессора, но производительность такой программы всегда будет ниже по сравнению с программой, написанной на языке третьего поколения, таком как C, C++ или C#. Например, может быть создано интеллектуальное инструментальное средство анализа скрытых закономерностей в данных, в котором программа на языке CLIPS вызывается из стандартной базы данных наподобие Oracle или MySQL, а эта программа выдает рекомендации пользователю базы данных. В СУБД Oracle для обеспечения возможности использования данных в других приложениях и последующего возврата управления в Oracle применяются курсоры. Программы на языке CLIPS допускают полноценную эксплуатацию в качестве вызывающих или вызываемых программных модулей на основе таких же принципов, как и любые другие программы. Еще один вариант состоит в том, что можно использовать или модифицировать исходный код на языке CLIPS и встроить этот код в свое приложение, а затем откомпилировать полученное программное обеспечение и подготовить для развертывания новое приложение.

Сопровождение и развитие

На данном этапе необходимо найти ответ на вопрос о том, как должна сопровождаться и развиваться готовая система.

Сопровождение и развитие любой экспертной системы — это задача, требующая еще более творческого подхода по сравнению с обычными программами. Дело в том, что функционирование экспертных систем не основано на использовании алгоритмов, поэтому производительность таких систем зависит от воплощенных в них обычных и экспертных знаний. В идеальном случае по мере приобретения новых знаний и модификации существующих знаний производительность экспертной системы должна оставаться постоянной.

Безусловно, при эксплуатации экспертных систем возникает проблема, не присущая обычным программам, основанным на алгоритмах, поскольку в обычных программах не бывает конфликтов между правилами, но таких конфликтов не следует опасаться. В действительности машина логического вывода как раз и предназначена для разрешения указанных конфликтов. По существу возникновение конфликта между правилами свидетельствует о том, что произошло согласование события с шаблоном в левой части больше чем одного правила, поэтому появились предпосылки запуска сразу нескольких правил. А люди постоянно совершают свою мыслительную деятельность именно по такому принципу.

Например, рассмотрим следующий **парадокс игры**. Допустим, что вы сидите в кресле и следите за футбольным матчем по телевизору, держа в одной руке бокал с пивом, а в другой — горсть чипсов. Что вы отправите прежде всего к себе в рот? Одно правило гласит: “Если у вас есть пища, то ешьте”, а другое правило говорит о том, что “Если у вас есть пиво, то пейте”. Поскольку в вашем распоряжении есть и еда, и напиток, то возникает конфликт правил.

К счастью, читатели этой книги могут взять на вооружение **минус первый закон успеха**: если у вас нет намерения отправиться по какому-то конкретному адресу, то отправляйтесь туда, куда вам вздумается, поскольку неизвестно, упустите ли вы что-либо, не отправившись в другое место. В данном случае очевидное решение парадокса игры состоит в том, чтобы обмакивать чипсы в пиво, т.е. одновременно и есть, и пить. (В конце концов, мир стал бы совсем другим, если бы граф Сэндвич не был настолько занят азартной игрой, что не мог выделить время для обеда, поэтому просто зажал кусок мяса между двумя ломтями хлеба, изобрел таким образом сэндвич и стал знаменитым!)

Кроме того, задача дальнейшего усовершенствования экспертной системы после ее доставки конечному пользователю является более значимой применительно к системам коммерческого назначения. Разработчики коммерческой системы стремятся к тому, чтобы созданный ими продукт позволил достичь финансового успеха. Это означает, что разработчики всегда готовы выслушать пожелания пользователей и, воспользовавшись предлагаемой оплатой, внести необходимые усовершенствования. Мир частного предпринимательства снова и снова предоставляет доказательства, что пользователи неизменно приобретают усовершенствованные версии программного обеспечения, руководствуясь неоправданной надеждой на то, что когда-либо количество ошибок в этих программах станет меньше.

Вполне естественно, что этого никогда не происходит, поскольку перечень имеющихся средств постоянно расширяется. В действительности стало общепринятой такая практика, что разработчики выпускают программные продукты с сотнями или даже тысячами известных ошибок. Фактически такой подход стал основой одного из способов борьбы с программным пиратством, поскольку законные пользователи имеют возможность выходить на Web-узел изготовителя, чтобы снова и снова загружать одну заплату за другой, затем инсталлировать сервисный

пакет, после чего опять устанавливать заплаты, пока не появится следующий сервисный пакет, и т.д., до тех пор, пока компания-разработчик не будет вынуждена выпустить новый продукт, чтобы не столкнуться с жесткими штрафами со стороны Комиссии по ценным бумагам и биржам за манипулирование с курсом акций. Может оказаться, что экспертная система в действительности так и не достигнет полного совершенства (как и любое подобное программное обеспечение); система лишь будет становиться все лучше и лучше.

6.4 Ошибки, возникающие на различных этапах разработки

Потенциальные крупные ошибки, возникающие в процессе разработки экспертной системы, могут классифицироваться с учетом тех этапов, на которых появление этих ошибок является наиболее вероятным, как показано на рис. 6.3. Ниже приведено подробное описание возможных ошибок.

Ошибки эксперта. Эксперт является **источником экспертных знаний** для экспертной системы. Если же знания эксперта ошибочны, то результаты реализации неправильных подходов могут отразиться на всем процессе разработки. Как было указано в главе 1, побочное преимущество создания экспертной системы состоит в том, что появляется возможность обнаружения ошибочных знаний, после того как знания эксперта становятся явно выраженными. Не следует рассчитывать на то, что возможность ошибки со стороны некоторого человека исключена лишь потому, что данный человек считается экспертом. Это утверждение является столь же истинным, как и констатация факта, что вероятность ошибки со стороны эксперта намного меньше по сравнению с тем, что может неправильно сообщить обычный человек, пытающийся разобраться в той же области экспертных знаний. Например, в принципе провести хирургическую операцию на головном мозге мог бы каждый, но пациент вряд ли выживет, если этим не будет заниматься опытный нейрохирург. С другой стороны, у всех на слуху телевизионные передачи, в которых сообщается о том, что в случае нанесения ущерба со стороны любого специалиста (или любого человека, обладающего деньгами), всегда найдется множество адвокатов, готовых помочь вам получить компенсацию.

Применительно к **ответственным** проектам, в которых жизнь и собственность человека подвергаются риску, может возникнуть необходимость предусмотреть формальные процедуры для сертификации знаний эксперта. Одним из подходов, который успешно использовался NASA для организации космических полетов, состоит в проведении **совещаний по разработке авиакосмической техники**, на



Рис. 6.3. Основные ошибки, возникающие в процессе разработки экспертной системы, и некоторые причины этих ошибок

которых регулярно рассматриваются решения предстоящих задач, а для разработки этих решений применяются методы анализа. В число участников совещания входят пользователи системы, независимые эксперты в данной проблемной области, разработчики системы и руководители, поэтому появляется возможность охватить все области, от которых зависит успех разработки.

Преимущество подхода, основанного на проведении совещаний группы специалистов, состоит в том, что знания эксперта подвергаются тщательной и скрупулезной проверке с самого начала процесса разработки, а это позволяет проще откорректировать ошибочные знания, ведь чем дольше ошибочные знания остаются необнаруженными, тем дороже обходится их корректировка. Если же проверка знаний эксперта не осуществляется с самого начала, то окончательной проверкой становится аттестация экспертной системы. Окончательная аттестация экспертной системы показывает, удовлетворяет ли система предъявленным требованиям, особенно в части правильности и полноты решений.

В крупных компаниях рабочие совещания проводятся постоянно. Иногда такие совещания именуются собраниями групп контроля производительности или обозначаются более нейтральными терминами. В 1970–1980-х годах проводились попытки использовать в разработке программного обеспечения одну из разновидностей совещаний по контролю за ходом выполнения работы; такие совещания позволяли программистам взаимно рецензировать код, разработанный каждым из участников совещания. К сожалению, этот подход быстро выродился и стал одним из самых неудачных инструментов контроля за ходом работ со стороны руководства, поскольку руководители начали оценивать производительность программистов с помощью таких показателей, как "количество безошибочных строк кода в сутки" или "количество ошибок в расчете на 100 строк кода". Безусловно, руководство не стремилось погружаться в детали этого процесса контроля со стороны товарищей по работе, рассчитывая на объективность самих программистов, но некоторые коллеги не проявляли столь же **товарищескую** непредвзятость, когда речь шла о повышениях в должности и премиях. Совещания могут стать полезным административным инструментом при соблюдении соответствующих предосторожностей, но при оценке надежности информации, полученной в ходе таких совещаний, необходимо всегда учитывать, что "товарищеская" критика может оказаться далеко не такой безобидной.

Еще одним полезным средством получения помощи могут оказаться **фокус-группы**. В частности, фокус-группы оказывают значительную помощь при осуществлении попытки узнать, какой продукт действительно найдет сбыт. В крупных компаниях обычно принято до начала разработки нового продукта нанимать людей для обсуждения с ними предстоящих проблем, чтобы в распоряжении компании был кто-то, способный выслушать непредвзятую оценку того, действительно ли людям требуется данный продукт, и какие характеристики он должен иметь. А недостатки подхода с использованием фокус-группы состоят в основном в том, что с этим связаны дополнительные затраты денег и времени. Тем не менее такие затраты часто окупаются благодаря повышению эффективности процесса разработки. Кроме того, фокус-группа может сберечь для компании миллионы долларов, если с ее помощью удастся определить, что продукт, намеченный к выпуску, не отвечает ожиданиям большинства потребителей.

Как правило, разработчикам системы не следует поручать выработку решений, касающихся маркетинга. Отступление от этого правила допускается только в том случае, если разработчики предназначают свой будущий продукт для людей, разделяющих общую с ними **культуру**. В данном случае термин "культура" используется в формальном смысле, как **корпоративная культура** или культура конечного пользователя; под культурой подразумевается общий основной набор ценностей, благодаря которым данная группа приобретает свои отличительные особенности. В частности, термин "корпоративная культура" закрепился в повседневном обиходе начиная с 1980-х годов, когда многие компании пытались опре-

делить, в чем состоят их отличительные особенности по сравнению с другими компаниями. Но в настоящее время понятие корпоративной культуры постепенно теряет свою значимость в связи с распространением глобального аутсорсинга.

Важность подхода, в котором учитывается общность культуры, выражается, например, в том, что наиболее успешные видеоигры создаются программистами, которые любят такие игры и постоянно участвуют в них, сидя за компьютером. Это означает, что преуспевающие разработчики видеоигр действительно знают свой рынок, а компании, в которых развита культура игры, неизменно нанимают программистов, не только достигших высокой квалификации в своей работе, но и любящих игры. Рассматриваемый пример может служить иллюстрацией к **минус второму закону успеха**: если вы собираетесь отправиться в путешествие, то оно пройдет более удачно при условии, что вы действительно стремитесь его совершить. В 1990-х годах многие компании потратили массу времени и денег на консультантов, чтобы они помогли сформировать корпоративную культуру, ввести всеобщий контроль качества и реализовать другие "тенденции десятилетия". Такая практика все еще продолжается, поскольку деловым предприятиям по-прежнему приходится развиваться и приспосабливаться к изменяющейся ситуации.

Но после того как компания становится открытой, для нее задача сохранения собственной корпоративной культуры значительно усложняется, поскольку теперь ей приходится выпускать ежеквартальные отчеты, привлекательные для фондового рынка. Это означает, что, возможно, президента компании, создавшего с нуля успешно действующую компьютерную компанию, придется заменить тем человеком, который раньше был президентом компании по выпуску безалкогольных напитков. (*Примечание.* Если читателя заставляет задуматься вопрос о том, почему здесь принята и положительная, и отрицательная нумерация законов успеха, отметим, что настоящая книга — приложение из области разработки философского интеллекта, а в этой области такая нумерация вполне допускается.)

> **Семантические ошибки**. Любая ошибка в семантике возникает, если не удается должным образом сообщить потребителю знаний смысл самих знаний. В качестве очень простого примера предположим, что эксперт сообщает такие сведения: "Огонь может быть погашен водой", а инженер по знаниям интерпретирует это утверждение как: "Любой огонь может быть погашен водой". Анализ этого примера показывает, что рекомендация для всех инженеров по знаниям научиться готовить пищу и самим попробовать потушить вспыхнувшее масло действительно оправдана (поскольку горящее масло невозможно потушить с помощью воды). Семантические ошибки возникают, если инженер по знаниям неправильно интерпретирует ответы эксперта, или эксперт неправильно интерпретирует вопросы инженера по знаниям, или и в том и в другом случае.

Одна из причин, по которой возникает неправильная интерпретация, обусловлена тем, что семиотические (знаковые) ассоциации могут вызвать неправильное понимание. Например, предположим, что рядом с вами посадили нового сотрудника и каждый раз после его прибытия на работу вы чувствуете неприятный для вас запах. У вас постепенно складывается о нем неблагоприятное впечатление, а однажды вы сталкиваетесь с ним вплотную и испытываете настоящий шок, услышав заявление: “Рад вас встряхнуть”, тогда как фактически он сказал: “Рад вас встретить” (и это зарегистрировала офисная видеокамера системы обеспечения безопасности). Безусловно, люди слышат сказанные (или читают написанные) слова, но другие чувства всегда оказывают свое влияние, и если в вашем сознании с кем-то ассоциируется что-то неприятное, то эти восприятия выражаются и в общении.

Эти факторы очень важно учитывать, подготавливаясь к собеседованию с экспертом. Специальная комната, предназначенная для проведения интервью, должна иметь стены нейтрального цвета и должна быть декорирована в мягких тонах, в нее необходимо подавать свежий воздух без запахов, должна быть установлена комфортная мебель (но не слишком располагающая к отдыху), а самое главное — в этой комнате должна быть ТИШИНА. Это позволит эксперту успокоиться и подсознательно установить ассоциацию между пребыванием в специальной комнате для интервью и подробным изложением знаний предметной области. А наихудшим местом для проведения собеседования является рабочее место инженера по знаниям. В этом случае эксперт чувствует дискомфорт, поскольку “играет на чужом поле”. Но столь же неудобно проводить собеседования на рабочем месте эксперта, так как и там постоянно звонят телефоны, раздаются сигналы, указывающие на поступление электронной почты, к вам подходят люди “только на минутку” и все вокруг напоминает о работе, которой придется заняться сразу после окончания собеседования. Вполне естественно, что эксперты стараются как можно быстрее завершить собеседование, чтобы снова вернуться к “настоящей работе”.

Еще один источник ошибок, возникающих при проведении собеседования, связан с когнитивными иллюзиями [78]. В общем иллюзия — это то, что вы наблюдаете, но знаете, что оно не существует, хотя и выглядит вполне реально, как мираж в пустыне. Люди обычно испытывают затруднения, пытаясь преодолеть когнитивные иллюзии, особенно если они сочетаются с соответствующими восприятиями. Например, рассматривая классический рисунок Мюллера–Лиера (Mueller–Lyer) и даже зная о том, что длина обоих отрезков, дополненных с двух сторон стрелками, является одинаковой (рис. 6.4), вы не сможете избавиться от ощущения, что эти отрезки не равны (даже после того, как измерите отрезки с помощью линейки).

Причина возникновения этой иллюзии связана с тем, что система зрительного восприятия человека работает независимо от той части мозга, которая отвечает за

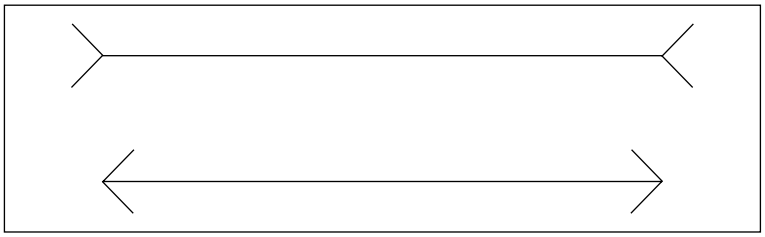


Рис. 6.4. Пример рисунка, вызывающего зрительную иллюзию

мышление и познание. Принято считать, что мозг функционирует на основе такой модели, как архитектура классной доски, в которой различные виды деятельности осуществляются специализированными участками нейронных кластеров. Такие кластеры могут рассматриваться как аппаратные компоненты, соединенные в мозгу связями, в отличие от привычных нам программных компонентов. Несомненно, система зрительного восприятия в процессе эволюции появилась раньше, чем когнитивные компоненты более высокого уровня, поскольку организм вообще не может выжить без развитого зрения.

Когнитивные иллюзии возникают постоянно и преодолеваются с большим трудом. Например, попытайтесь ответить на следующий вопрос, не глядя на карту США: “Какой город находится западнее — Лос-Анджелес, штат Калифорния, или Рено, штат Невада?” Большинство людей сразу же отвечают, что это — Лос-Анджелес, поскольку он находится на побережье Тихого океана, а Рено — на внутренней территории штата Невада. Это — хороший пример рассуждений на основе здравого смысла. К сожалению, он также является примером неправильных рассуждений. В действительности город Рено расположен западнее Лос-Анджелеса, в чем можно убедиться, проведя проверку по карте или сверив географические координаты этих городов, полученные в Web. В качестве еще одного примера предлагаем ответить на вопрос о том, какой город расположен севернее, Рим или Нью-Йорк? Мы привыкли видеть в фильмах солнечную Италию и заснеженный Нью-Йорк, поэтому здравый смысл подсказывает, что Нью-Йорк находится ближе к Северному полюсу. Но и этот ответ является неправильным, поскольку Нью-Йорк южнее Рима. Это — еще одна когнитивная иллюзия, причем настолько навязчивая, что даже зная правду, в нее нелегко поверить, как и преодолеть описанную выше иллюзию со стрелками.

Подобные когнитивные иллюзии с большим трудом поддаются обнаружению, особенно при проведении собеседования с экспертом. Достаточно сказать, что человек, являющийся экспертом в одной предметной области, не обязательно должен хорошо ориентироваться в любых других областях. Эксперты — тоже люди, поэтому подвержены когнитивным иллюзиям, как и все остальные. Опасность состоит в том, что лицо, проводящее собеседование, может принять предположе-

ние, что все сказанное экспертом является правильным, даже если эта информация выходит за пределы области его экспертных знаний. (Но если речь идет о вашем начальнике, то в этом, безусловно, нельзя сомневаться, и об этом нужно всегда ему сообщать, особенно в дни получения зарплаты.)

Синтаксические ошибки. Это — простые ошибки, возникающие в результате ввода правила или факта в неправильной форме. Инструментальные средства экспертной системы должны отмечать подобные ошибки и выдавать соответствующие сообщения. Другие ошибки, возникающие на этапе создания базы знаний, могут быть обусловлены наличием ошибок в источнике знаний, которые не были обнаружены на предыдущих этапах.

Ошибки машины логического вывода. Машина логического вывода реализована с помощью программы, а эта программа может содержать ошибки, как и любое другое программное обеспечение. Безусловно, ко времени выпуска инструментального средства создания экспертных систем для общего пользования все наиболее очевидные ошибки должны быть исправлены. Но могут существовать и такие ошибки, которые проявляются в редких случаях, например, если 1 апреля в рабочем списке правил накапливается 159 правил. А другие ошибки могут оказаться еще более трудноуловимыми и проявляться только в некоторых операциях согласования с шаблонами.

Вообще говоря, ошибки в программном обеспечении машины логического вывода могут проявляться при согласовании с шаблонами, разрешении конфликтов и выполнении действий. Если такие ошибки не обнаруживаются постоянно, то их выявление может оказаться очень сложным. Прежде чем воспользоваться каким-либо инструментальным средством для создания экспертной системы, применяемой в ответственном приложении, необходимо определить, как проводилась аттестация этого инструментального средства. Как правило, каждая новая версия инструментального средства создания экспертных систем должна проходить проверку с помощью тестового набора; такая проверка предусмотрена, в частности, для языка CLIPS. Проверка автоматизирована, поэтому позволяет получить объективные результаты, а ее проведение не происходит в спешке, поскольку никто не настаивает на скорейшем выпуске продукта под давлением внешних обстоятельств, чем обычно грешит руководство.

Простейшим методом проверки наличия ошибок в инструментальном средстве является классический метод опроса других пользователей, а также поставщиков этого инструментального средства. Любой поставщик программного обеспечения должен быть готов предоставить список заказчиков, отчеты об обнаруженных ошибках, внесенных исправлениях, а также сообщить, как долго это

инструментальное средство находится в эксплуатации. Еще одним превосходным источником информации может стать группа пользователей.

Ошибки в цепи логического вывода. Эти ошибки могут быть вызваны наличием ошибочных знаний, семантическими ошибками, ошибками в программном обеспечении машины логического вывода, неправильными спецификациями приоритетов правил и незапланированными взаимодействиями правил. Более сложные ошибки в цепи логического вывода возникают из-за неопределенности в правилах и свидетельствах, распространения неопределенности в цепи логического вывода, а также из-за немонотонности, обусловленной необходимостью извлекать неправильные факты, а также все другие факты, выработанные на основе неправильных фактов. Такое поведение машины логического вывода можно сравнить с тем, что человек, узнав об отмене собрания, не отправляется в пустой конференц-зал, безуспешно дожидаясь, пока в нем соберутся люди, а реализует другие планы (например, решает немного вздремнуть, так как вокруг больше никого нет).

Недостаточно просто выбрать какой-то метод учета неопределенности, поскольку никогда не удается автоматически разрешить все вопросы, связанные с неопределенностью. Например, даже прежде чем выбрать простой байесовский логический вывод, необходимо выяснить, достаточно ли для достижения успеха принять предположение об условной независимости.

Ошибки определения границ незнания. На всех этапах разработки приходится сталкиваться с одной и той же проблемой — определения границ незнания системы. Как было указано в главе 1, обычно можно рассчитывать на то, что эксперты-люди знают пределы своих знаний, поэтому надеяться, что сами эксперты смогут корректно показывать снижение качества своих рекомендаций при достижении границ своего незнания. Эксперты-люди должны быть достаточно честными, чтобы признать реальное увеличение неопределенности в своих выводах вблизи от этих границ. Но экспертная система может по-прежнему уверенно выдавать ответы, даже если цепи логического вывода и сами свидетельства становятся очень **хрупкими** и ненадежными (такой недостаток экспертных систем может быть преодолен, только если подобные системы специально запрограммированы для работы в условиях неопределенности). А если в надежность выводов системы необоснованно поверят люди, может случиться худшее.

6.5 Разработка программного обеспечения и экспертные системы

В предыдущих разделах были приведены общие соображения, которые касаются использования подхода, основанного на экспертных системах, а в этом разделе рассматриваются этапы разработки экспертных систем с более формальной точки зрения — с позиций инженера по знаниям, который фактически формирует систему.

С тех пор как экспертные системы вышли за пределы исследовательских лабораторий, появилась реальная потребность в создании для них высококачественного программного обеспечения на основе стандартов, не уступающих стандартам для обычного программного обеспечения. Общепринятой методологией разработки качественного программного обеспечения, соответствующего требованиям к коммерческим, промышленным и правительственным стандартам, является программотехника.

Необходимо тщательно придерживаться качественных стандартов разработки любого программного продукта, так как в противном случае не удастся достичь высокого качества такого продукта. А экспертные системы — это такие же программные продукты, как и любое другое программное обеспечение, будь то текстовый процессор, программа учета заработной платы или компьютерная игра.

Тем не менее по своему **назначению** экспертные системы значительно отличаются от обычных потребительских программных продуктов, таких как текстовые процессоры и видеоигры. Термин “назначение” служит для описания общей цели создания любого проекта или технологии. Обычно технология экспертных систем приобретает важное предназначение, которое связано с обеспечением доступа к экспертным знаниям в ситуациях, требующих высокой производительности, а также, возможно, связанных с опасностями, когда под угрозой находится жизнь и собственность человека. В этом и состоит определение тех ответственных приложений, о которых шла речь в предыдущем разделе. А системы, основанные на знаниях, не обязательно должны соответствовать таким высоким стандартам.

Ответственные приложения весьма отличаются по своему назначению от программ, эксплуатируемых в менее жестких условиях, например, текстовых процессоров и видеоигр, поскольку при разработке последних основное внимание уделяется повышению эффективности и обеспечению удобства в работе. В частности, в связи с наличием ошибки в программе текстового процессора или видеоигры жизнь человека не подвергается опасности (или, по крайней мере, этого не должно быть).

Экспертные системы представляют собой высокопроизводительные системы, которые должны обладать высоким качеством, так как в противном случае при их эксплуатации придется столкнуться с ошибками. Как показано на рис. 6.5,

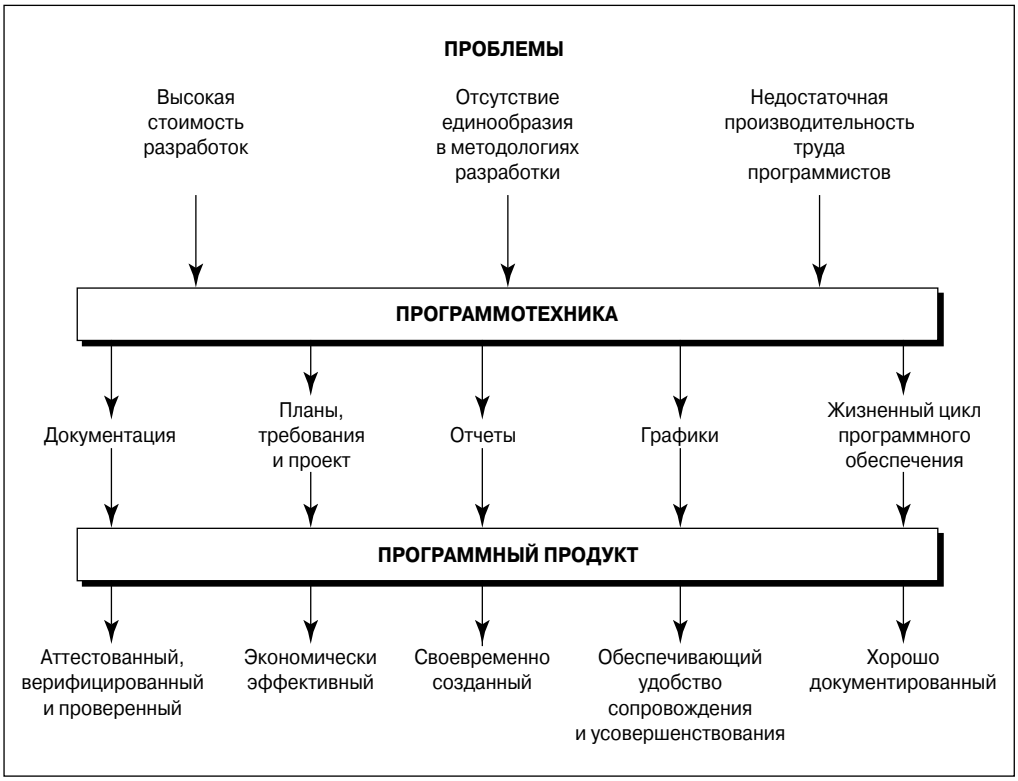


Рис. 6.5. Методология разработки программного обеспечения

в основе методологий создания качественного программного обеспечения лежит программотехника.

Термину “качество” нелегко дать общее определение, поскольку в разных областях человеческой деятельности он может приобретать различный смысл. Одним из способов определения **качества** является перечисление обязательных или желательных атрибутов некоторого объекта, измеряемых по некоторой шкале. В данном контексте термин “объект” используется для обозначения любых программных или аппаратных средств, таких как, допустим, рассматриваемый программный продукт. Атрибуты и их значения принято называть **метриками**, поскольку они используются как показатели измерения характеристик объекта. Например, надежность жесткого диска принято рассматривать как метрику, определяющую его качество. Одной из единиц измерения этого атрибута является **среднее время наработки на отказ** (Mean Time Between Failures — MTBF) жесткого диска. Надежный жесткий диск должен иметь показатель MTBF, составляющий не меньше 50 тысяч часов эксплуатации между отказами, а менее надежный, дешевый жесткий диск может иметь показатель MTBF, составляющий 10 тысяч

часов. В XX столетии часто приходилось наблюдать, как пользователи включают свои компьютеры всего на несколько часов дома или на работе, а теперь в основном принято оставлять компьютер включенным круглосуточно (по формуле $24 \times 7 \times 52$), поэтому общая продолжительность эксплуатации жесткого диска значительно увеличилась. Безусловно, современные операционные системы позволяют переводить в режим уменьшенного потребления электроэнергии не только мониторы, но и жесткие диски. Тем не менее, если продолжительность времени простоя, по истечении которого такое устройство переводится в режим ожидания, установлена слишком короткой, то останов и последующий запуск жесткого диска происходит слишком часто, что может стать источником новых рисков.

Производители жестких дисков стремятся не только повысить их быстродействие, но и сократить частоту возникновения ошибок при записи. Предположим, что согласно спецификации жесткий диск должен обеспечивать запись, допуская ошибки не больше чем в одном бите на каждый миллиард операций записи. В 1990-х годах, когда обычные жесткие диски имели объем порядка 20 Мбайт, это означало, что жесткий диск можно было заполнить больше 50 раз, ни разу не столкнувшись с ошибкой при записи. А в настоящее время появились жесткие диски с емкостью порядка 200 Гбайт, поэтому при том же показателе в одну ошибку на миллиард операций записи количество ошибок на жестком диске может достигнуть 200 (при условии, что частота ошибок не изменится).

Если ошибка произойдет при записи сжатого изображения, такого как `.jpg`, то, по всей видимости, она не будет обнаружена. Но если ошибка возникнет при записи строки кода, то искажение в одном бите может привести к тому, что оператор программы, допустим $X = A$, превратится в $X = B$, а это может повлечь за собой весьма неблагоприятные последствия. Современные жесткие диски имеют встроенный датчик, который непрерывно контролирует состояние поверхности диска с помощью средства, называемого SMART (Self-Monitoring, Analysis and Reporting Technology). Доступ к средствам **технологии SMART** может быть получен с помощью таких программ, как Norton System Utilities, например, путем вызова на выполнение программы Disk Doctor Utility. Такие программы позволяют получить заблаговременное предупреждение о большинстве возможных нарушений в работе аппаратного обеспечения жесткого диска. Однако, чтобы это предупреждение действительно не прошло даром, необходимо создавать резервные копии и заменять жесткий диск, пока еще не произошла потеря данных.

Технология SMART — это инструмент, который предсказывает возможное ухудшение состояния жесткого диска, но не позволяет его исправить, если проблема связана с износом и снижением качества поверхности магнитного носителя, поскольку ничто в мире не идеально. Но в каждом новом поколении жестких дисков применяются все более высокие значения частоты вращения пластин, а магнитное покрытие этих пластин неизменно становится все более совершенным и надежным.

В 1990-х годах стандартным показателем частоты вращения было значение 3600 мин$^{-1}$, затем это значение увеличилось до 5400 мин$^{-1}$ и теперь составляет 7200 мин$^{-1}$; появились даже такие выдающиеся модели жестких дисков, в которых частота вращения достигает 10000 мин$^{-1}$. Чтобы проще было представить себе эту скорость, рассмотрим такую аналогию. Колесо автомобиля имеет диаметр приблизительно 60 см и длину окружности около 190 см. Если бы колеса автомобиля вращались с частотой 3600 мин$^{-1}$, то автомобиль двигался бы со скоростью примерно 410 км/ч, а при частоте вращения 10000 мин$^{-1}$ скорость автомобиля достигла бы 1130 км/ч, т.е. приблизилась бы к скорости звука. Именно поэтому в любой компьютерной системе основным источником неисправностей являются механические компоненты, создающие вибрацию и вырабатывающие тепло.

Но после ввода в действие 64-битовых процессоров возник еще один первоисточник отказов, поскольку новые процессоры стали вырабатывать намного больше тепла по сравнению с 32-битовыми. Эта проблема оказалась настолько важной, что предназначенные для их установки системные платы начали выпускаться с температурными датчиками, прямо на самом процессоре стали устанавливаться радиаторы с вентиляторами, корпуса компьютеров для охлаждения оснащались шестью вентиляторами, а для останова системы в случае чрезмерного повышения температуры было создано специальное программное обеспечение. Кроме того, жесткий диск и память стали размещаться на максимально возможном расстоянии от процессора. К счастью, в новых версиях 64-битовых процессоров этот недостаток преодолен.

В табл. 6.1 приведен контрольный список некоторых показателей, которые могут использоваться в оценке качества экспертной системы. Эти показатели следует рассматривать только в качестве ориентировочных, поскольку, когда речь идет о конкретной экспертной системе, не все они могут оказаться в достаточной степени приемлемыми. Тем не менее всегда важно иметь в своем распоряжении перечень необходимых характеристик, который может использоваться при оценке качества.

Следует всегда руководствоваться заранее подготовленным списком требуемых показателей, поскольку он позволяет проще расположить показатели по приоритетам в случае обнаружения конфликтов между ними. Например, увеличение продолжительности проверки экспертной системы в целях обеспечения ее надежной аттестации приводит к увеличению издержек. Вообще говоря, решение о прекращении испытаний является очень сложным, требующим учета таких трех факторов, как степень выполнения графика работ, объем затрат и соответствие системы предъявляемым требованиям. В идеальном случае испытания должны быть завершены в условиях положительной оценки всех трех указанных факторов. Но на практике некоторые из них часто рассматриваются как более важные, чем другие, поэтому и ослабляются ограничения, касающиеся реализации всех факторов без исключения.

Таблица 6.1. Некоторые показатели оценки качества программного обеспечения для экспертных систем

Показатель оценки качества программного обеспечения
Правильные выходные данные при наличии правильных входных данных
Полный объем выходных данных при наличии правильных входных данных
Неизменные выходные данные при повторном поступлении одних и тех же входных данных
Достаточно высокая надежность, благодаря которой не происходит отказ из-за ошибок (или, по крайней мере, происходит редко)
Удобство в эксплуатации, а также, желательно, дружественность
Удобство в сопровождении
Расширяемость
Соответствие потребностям и запросам пользователя, подтвержденное путем проведения аттестации
Правильность и полнота, подтвержденные в результате проверки
Экономическая эффективность
Наличие кода, применимого для создания других приложений
Обеспечение переносимости в другую аппаратную и (или) программную среду
Возможность сопряжения с другим программным обеспечением
Удобство кода для чтения и понимания
Правильность
Точность
Корректное снижение качества результатов по мере приближения к границам незнания
Встроенные возможности взаимодействия с другими языками
Аттестованная база знаний
Наличие средства объяснения

6.6 Жизненный цикл экспертной системы

Одним из ключевых понятий программотехники является **жизненный цикл**. Жизненный цикл программного обеспечения — это период времени, который начинается с формулировки первоначальных концепций, лежащих в основе программного обеспечения, и заканчивается переводом программ в категорию непригодных для дальнейшего использования. Понятие жизненного цикла позволяет отказаться от отдельного рассмотрения этапов разработки и сопровождения и создать общий

контекст, непрерывно связывающий все этапы. При таком подходе появляется возможность запланировать работы, связанные с сопровождением и дальнейшим развитием, на ранних этапах жизненного цикла, что позволяет в последующем снизить расходы на осуществление указанных этапов.

Расходы на сопровождение

Расходы на сопровождение обычного программного обеспечения, достигшего коммерческого успеха, которое неизменно модифицируется и дополняется через каждые несколько лет, вполне могут превзойти в десятки раз первоначальные расходы на разработку, связанные с созданием первого выпуска данного программного продукта, и возрастать с появлением каждой новой версии. Безусловно, может показаться, что такие расходы со временем становятся чрезмерными, но, как было указано выше, само сопровождение программы может стать источником существенных доходов. Однако **минус третий закон успеха** гласит: если вы не любите путешествовать, то довольствуйтесь теми занятиями, которые вас устраивают на данный момент. Поэтому если вы неизменно продолжаете устранять ошибки в вашем программном продукте, то не заменяйте его название чем-то полностью новым. При этом целесообразно придерживаться такого принципа: варьировать старое название, но не вводить в это название неприятный оттенок, например, не называть новую версию “Doors BugFix No. 10” (Программа Doors, с исправлением ошибок номер 10), а назвать ее, скажем, “Doors 2010”.

В последнее время все чаще создаются экспертные системы, основанные не на экспертных знаниях человека, а на обычных знаниях. Из этого правила есть заметные исключения. К ним относятся интеллектуальные обучающие системы для такого специального персонала, как врачи или летчики. Экспертные системы весьма успешно применяются в качестве обучающих систем и позволяют значительно сократить затраты времени, на которые обычно приходится идти, если к преподаванию вступительного материала специалисты привлекаются в отрыве от основной деятельности. Но в принципе, если созданию экспертной системы уделено достаточно времени и ресурсов, то с ее помощью часто можно предоставить студенту такую подготовку, что он может немедленно приступать к реальной работе.

По мере того как убыстряется смена технологий, снижается потребность в сохранении в системе знаний людей, отправляющихся на пенсию. Но необходимость в сопровождении экспертных систем возрастает, поскольку относительный объем содержащихся в них эвристических знаний, основанных на опыте, возрастает. Еще бóльших расходов на сопровождение и доработку требуют экспертные системы, в которых осуществляется большой объем логического вывода в условиях неопределенности. Безусловно, в результате появления автоматизированных инструментальных средств формирования и импорта онтологий в экспертные си-

стемы задача сопровождения таких систем упрощается, но вместе с тем сами эти системы также становятся крупнее, как и любое программное обеспечение, развивающееся со временем, поэтому задача сопровождения отнюдь не становится проще.

Модель каскадного развития жизненного цикла

Для обычного программного обеспечения был разработан целый ряд различных моделей жизненного цикла. Классической моделью жизненного цикла является впервые разработанная модель такого типа — **каскадная модель**, которая показана на рис. 6.6.

Эта модель хорошо знакома программистам, занимающимся разработкой обычного программного обеспечения. В каскадной модели каждый этап производственной деятельности заканчивается верификацией и аттестацией (Verification and Validation — V & V), что позволяет свести к минимуму количество проблем, подлежащих решению после перехода на следующий этап. Кроме того, следует отметить, что, как показано на рис. 6.6, стрелки проходят в прямом и обратном направлениях одновременно только через один этап. Тем самым подчеркивается, что разработка осуществляется итеративно с охватом только двух смежных этапов. Это позволяет значительно уменьшить расходы по сравнению с таким подходом (требующим гораздо более значительных издержек), в котором итеративная разработка охватывает несколько этапов. Стоимость возврата к предыдущим этапам измеряется не простой линейной функцией, а определяется экспоненциальной зависимостью от сложности связанных с этим операций.

Для обозначения понятия жизненного цикла применяется еще один термин — **модель процесса**, поскольку это понятие связано с двумя перечисленными ниже фундаментальными вопросами разработки программного обеспечения.

1. Что должно быть сделано на следующем этапе?
2. Какую продолжительность должен иметь следующий этап?

Модель процесса фактически представляет собой **метаметодологию**, поскольку определяет порядок и продолжительность применения обычных программных методов. Ниже перечислены наиболее широко используемые методы (или методологии) разработки программного обеспечения.

Выбор конкретных методов для осуществления работ на определенной стадии, в том числе указанных ниже.

- Планирование.
- Определение требований.
- Приобретение знаний.
- Проверка.

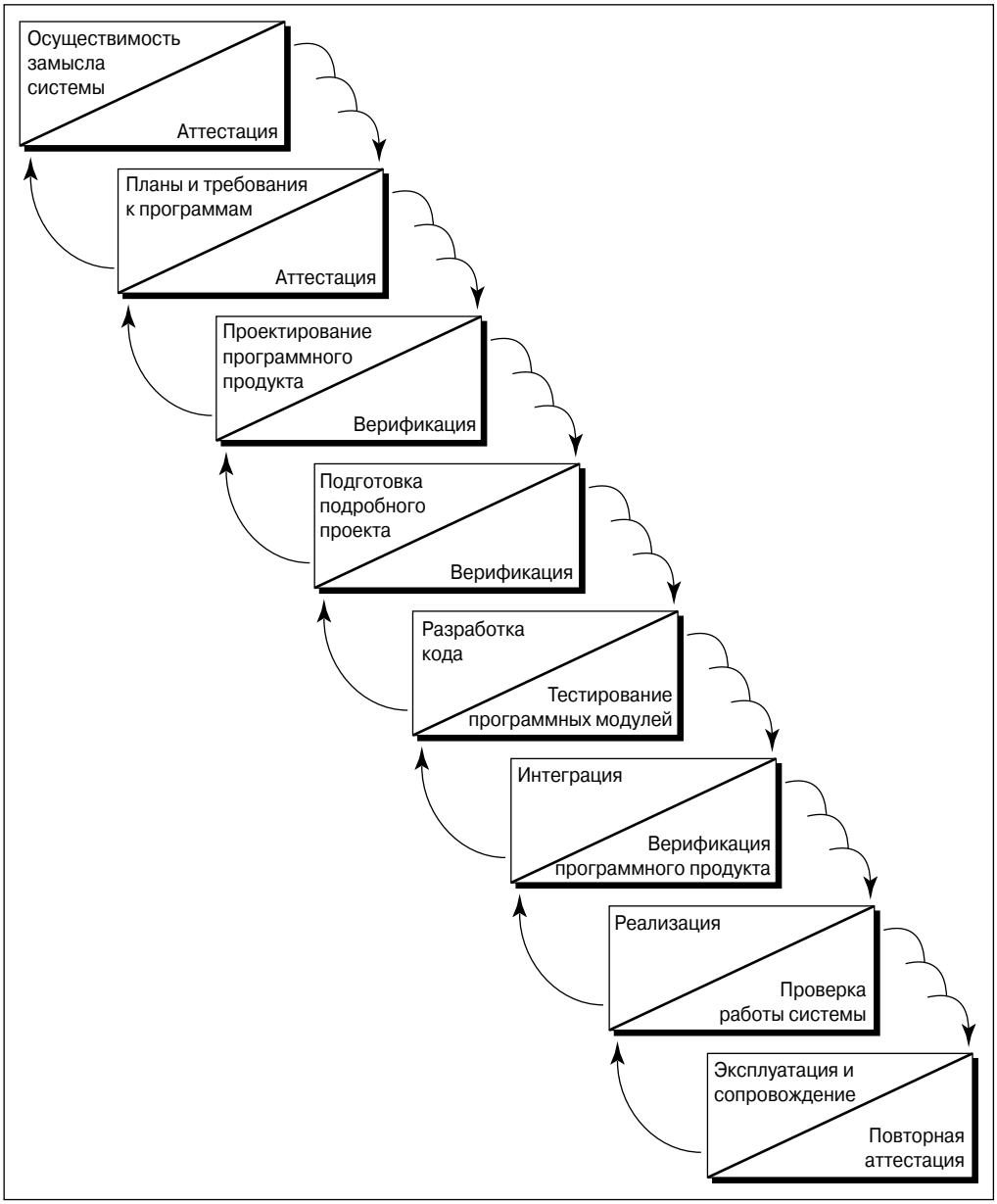


Рис. 6.6. Каскадная модель жизненного цикла программного обеспечения

Представление результатов выполнения работ на текущей стадии, как указано ниже.

- Составление документации.

- Разработка кода.
- Подготовка диаграмм.

Модель развития жизненного цикла на основе “кодирования и исправления”

Для разработки программного обеспечения применялся целый ряд моделей процесса. Одной из ранних моделей (впрочем, вряд ли даже имеющих право называться моделями) является печально знаменитая **модель разработки на основе “кодирования и исправления”**, в которой предусматривается написание некоторого кода и последующее исправление, если этот код действует неправильно. Как правило, именно такой метод выбирают студенты, впервые приступающие к освоению обычного программирования и программирования экспертных систем.

К 1970 году недостатки подхода на основе “кодирования и исправления” стали настолько очевидными, что была разработана каскадная модель, которая позволила наконец предоставить систематизированную методологию, оказавшуюся особенно удобной для крупных проектов. Но при использовании каскадной модели возникали сложности, поскольку в ней предполагалось, что известна вся информация, необходимая для реализации каждого этапа. Но на практике часто было невозможно сформулировать общие требования до того, как будет создан некоторый прототип. Такая ситуация привела к выработке концепции **двукратной разработки**, на основании которой создавался прототип, определялись требования, а затем создавался окончательный вариант системы.

Инкрементная модель жизненного цикла

Инкрементная каскадная модель представляет собой результат усовершенствования каскадного подхода и стандартного нисходящего подхода. Основная идея инкрементной разработки состоит в том, что программное обеспечение создается по принципу поэтапного наращивания функциональных возможностей. Применение инкрементной модели в крупных проектах разработки обычного программного обеспечения оказалось очень успешным. Инкрементная модель используется также для разработки экспертных систем; на ее основе происходит добавление правил, в результате чего возможности системы постепенно возрастают и система превращается вначале в помощника, затем в коллегу и наконец достигает уровня эксперта. Таким образом, в экспертной системе **крупными этапами усовершенствования** становится переход из статуса помощника в статус коллеги, и из статуса коллеги в статус эксперта.

Мелкие этапы усовершенствования соответствуют такому увеличению объема экспертных знаний на каждом уровне, которое может рассматриваться как заметное достижение. Наконец, **микроусовершенствование** — это такое изменение

в объеме экспертных знаний, которое соответствует добавлению или уточнению отдельного правила.

Основное преимущество инкрементной модели состоит в том, что задачи проверки, верификации и аттестации достижений в части расширения функциональных возможностей решаются проще по сравнению с каскадной моделью, поскольку последняя предусматривает решение этих задач на отдельных этапах применительно ко всему разрабатываемому продукту. Таким образом, инкрементная модель позволяет проводить с участием эксперта проверку, верификацию и аттестацию результатов каждого этапа усовершенствования функциональных возможностей сразу же после его завершения, а не ожидать проведения аттестации всей системы по окончании разработки. Благодаря этому снижается стоимость внесения исправлений в систему. По существу, инкрементная модель аналогична по своему назначению непрерывно дополняемому первоначальному прототипу, над которым проводится работа в течение всей разработки. Как уже было сказано, в подходе на основе двукратной разработки первоначальный прототип применяется только на начальных этапах для определения требований к системе, а в подходе на основе инкрементной модели как непрерывно развивающийся прототип рассматривается сама система.

Модель спирального развития жизненного цикла

Один из способов визуального представления инкрементной модели состоит в применении стандартной **спиральной модели** разработки программного обеспечения Барри Бома (Barry Boem), которая показана на рис. 6.7. После прохождения каждого цикла в спирали к системе добавляются определенные функциональные возможности. Обозначение конечной точки как “Системы, предоставляемой заказчику” фактически не становится концом спирали. Вместо этого закладывается новая спираль, в которую включены этапы сопровождения и развития системы. Саму спиральную модель можно также дополнительно усовершенствовать, определив более точно общие этапы приобретения знаний, разработки кода, проведения оценки и планирования.

6.7 Подробная модель жизненного цикла

Одной из моделей жизненного цикла, которая успешно использовалась во многих проектах экспертных систем, является **линейная модель**, показанная на рис. 6.8. Определяемый таким образом жизненный цикл состоит из ряда этапов, начинающихся с планирования и заканчивающихся оценкой готовой системы, и описывает разработку системы до некоторого момента, в который должна быть получена оценка функциональных возможностей системы. После этого в жизненном цикле повторяется та же последовательность от планирования до оценки

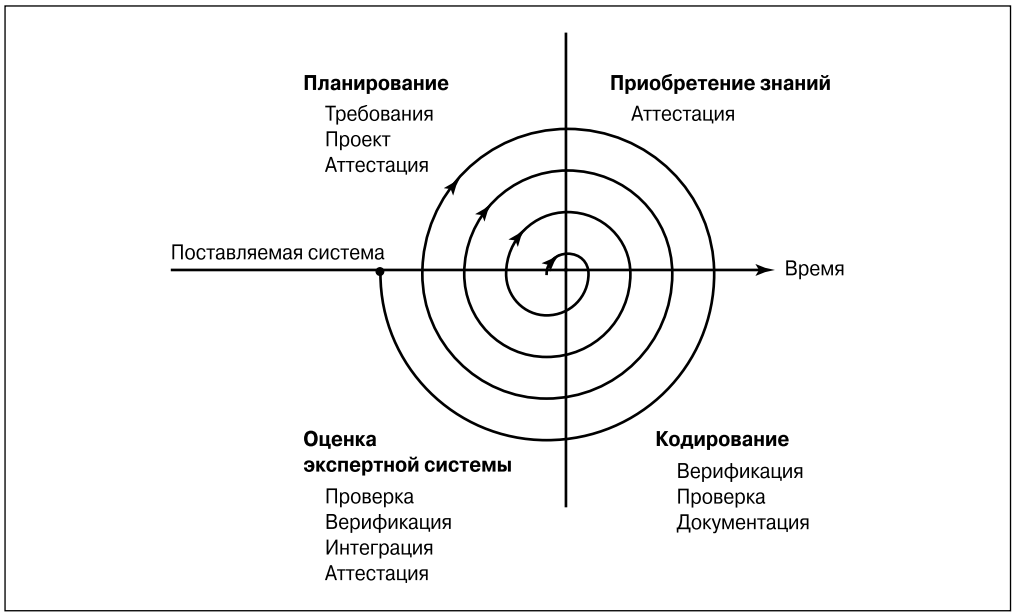


Рис. 6.7. Спиральная модель разработки экспертной системы

системы. Эти этапы разработки осуществляются до тех пор, пока система не будет передана в повседневную эксплуатацию. В дальнейшем это определение жизненного цикла применяется для осуществления последующих этапов сопровождения и развития системы. Безусловно, на схеме это явно не показано, но параллельно с осуществлением данных этапов происходит верификация и аттестация. Для обеспечения высокого качества экспертной системы необходимо не просто исправлять обнаруженные ошибки, но неуклонно придерживаться одной и той же последовательности этапов. Если какие-либо этапы пропускаются, пусть даже ради исправления единственной небольшой ошибки, общее качество системы снижается.

Жизненный цикл, показанный на рис. 6.8, может рассматриваться как один цикл спиральной модели. Каждый этап состоит из **задач**. На некотором этапе может не потребоваться выполнение всех задач, особенно после того как начинается сопровождение и развитие системы. Дело в том, что в рассматриваемом подходе понятие задачи формулируется таким образом, что каждая задача рассматривается как соединение всех задач, относящихся ко всему жизненному циклу, начиная от формулировки исходной концепции и заканчивая переводом программы в категорию подлежащих замене. Кроме того, формулировка задач зависит от того, какое именно приложение подлежит разработке, и поэтому рекомендуемый состав каждой задачи должен рассматриваться только в качестве руководящих, а не

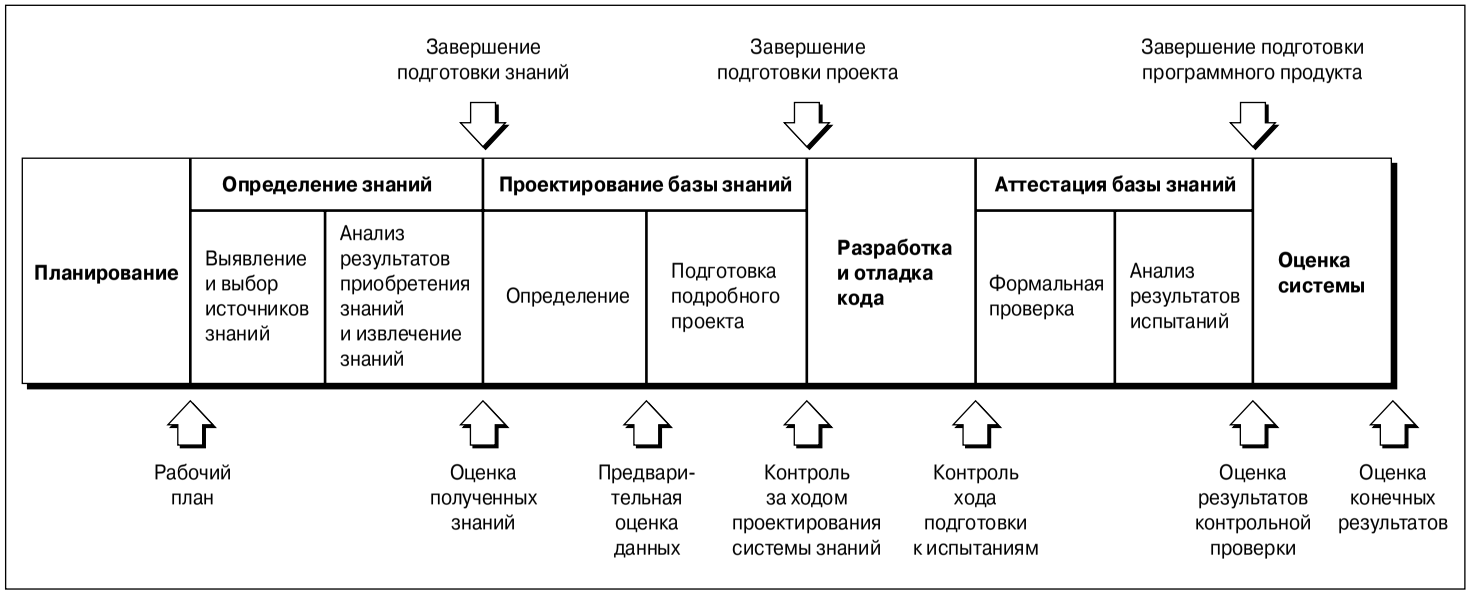


Рис. 6.8. Линейная модель жизненного цикла разработки экспертной системы

абсолютных требований, которые должны быть выполнены для успешного завершения каждого этапа.

Рассмотрим эту модель жизненного цикла более подробно, чтобы показать, как много факторов приходится рассматривать при создании крупной и качественной экспертной системы. С другой стороны, при разработке небольших экспериментальных прототипов, не предназначенных для общего пользования, нужны не все задачи и даже не все этапы. Но удивительно то, насколько большой объем программного обеспечения разрабатывался для личного пользования или ради эксперимента, после чего был передан в руки коллег, оказался весьма востребованным и наконец поступил в общее пользование.

Планирование

Назначение этапа планирования состоит в подготовке формального **рабочего плана** разработки экспертной системы. Рабочий план представляет собой набор документов, на основании которых должны осуществляться действия по руководству разработкой и оценке результатов. Отдельные действия, выполняемые на этом этапе, приведены в табл. 6.2.

Наиболее важной задачей, которая должна быть решена в процессе реализации жизненного цикла, является *оценка осуществимости*. При проведении этой

Таблица 6.2. Действия, выполняемые на этапе планирования

Действие	Назначение
Оценка осуществимости	Определение того, оправданы ли затраты по созданию системы, и, в случае положительного ответа, выяснение, должна ли для этого использоваться технология экспертных систем
Управление ресурсами	Оценка объема необходимых ресурсов, в том числе рабочей силы, времени и финансов, а также программного обеспечения и аппаратных средств. Приобретение и управление требуемыми ресурсами
Определение этапности решения задач	Выявление состава задач и определение поэтапного порядка их решения
Планирование	Составление планов, регламентирующих время начала и окончания этапов решения задач
Предварительная функциональная компоновка	Определение того, какие операции должна осуществлять система, путем указания функций системы высокого уровня. При осуществлении указанного действия на этапе планирования уточняется назначение системы
Определение требований высокого уровня	Подготовка составленного в общих терминах описания того, как должны выполняться функции системы

оценки необходимо найти ответы на вопрос о том, стоит ли заниматься реализацией данного проекта, а также на связанный с ним вопрос о том, является ли приемлемым подход, основанный на использовании экспертной системы. От ответов на эти два вопроса зависит решение, касающееся дальнейшей реализации рассматриваемого проекта на основе подхода, в котором применяется экспертная система. При оценке осуществимости приходится учитывать множество факторов. Как было описано в разделе 6.1, в число этих факторов входят выбор подходящей проблемной области экспертной системы, определение затрат, выигрыша и т.д.

Определение знаний

Целью **этапа определения знаний** является выяснение требований к знаниям, содержащимся в экспертной системе. Этап определения знаний состоит из двух основных задач, которые описаны ниже.

- Идентификация и выбор источника знаний.
- Приобретение, анализ и извлечение знаний.

Каждая из этих основных задач состоит из других задач. В табл. 6.3 описаны задачи, связанные с идентификацией и выбором источника знаний.

Таблица 6.3. Задачи идентификации и выбора источника знаний

Задача	Назначение
Идентификация источника знаний	Поиск ответа на вопрос о том, кто и что должно явиться источником знаний; при решении этой задачи проблема доступности источника знаний не рассматривается
Оценка важности источника знаний	Распределение источников знаний по приоритетам, отражающим их значимость для разработки
Оценка доступности источника знаний	Составление списка источников знаний, упорядоченного с учетом их доступности. Как правило, Web, книги и другие документы позволяют получить к ним доступ намного проще, чем эксперты-люди
Выбор источника знаний	Окончательный выбор источников знаний с учетом их важности и доступности

Задачи приобретения, анализа и извлечения знаний описаны в табл. 6.4.

Основной целью осуществления задач приобретения знаний, анализа знаний и извлечения знаний является подготовка и проверка знаний, требуемых в системе. К тому времени, когда будет собран требуемый объем знаний, все эти знания должны быть правильными и пригодными для осуществления следующего этапа проектирования знаний. Сбор знаний может выполняться не только с использованием общепринятого метода проведения собеседования с экспертами, но и с помощью многих программных инструментальных средств, позволяющих обеспечить

Таблица 6.4. Задачи приобретения, анализа и извлечения знаний

Задача	Назначение
Определение стратегии приобретения знаний	Регламентация процедур приобретения знаний с указанием методов получения интервью у экспертов, обработки документов, вывода правил методом индукции, создания устойчивых решеток (как способа представления знаний) и т.д.
Идентификация элементов знаний	Выборка из источников знаний конкретных знаний, которые потребуются на данной итерации жизненного цикла
Создание системы классификации знаний	Классификация и упорядочение знаний в целях упрощения операций верификации и усвоения знаний, выполняемых разработчиками. По возможности при решении этой задачи следует использовать способы классификации с помощью иерархических групп
Разработка подробной функциональной компоновки	Составление подробной спецификации функциональных возможностей системы. Этот уровень в большей степени относится к компетенции технических специалистов, тогда как задача разработки предварительной функциональной компоновки решается на уровне руководящих работников
Предварительное планирование процессов передачи управления	Описание общих этапов, реализуемых в ходе эксплуатации экспертной системы. Этапы соответствуют логическим группам правил, которые одновременно активизируются и (или) переходят в неактивное состояние, в связи с чем в системе осуществляется передача управления от одного компонента к другому
Предварительное составления руководства пользователя	Подготовка описания системы с точки зрения пользователя. Эта задача в составе задач создания системы часто игнорируется, но является чрезвычайно важной. Пользователей абсолютно необходимо привлекать к работе как можно раньше, чтобы можно было скорее ознакомиться с их откликами. Если систему никто не использует, она ничего не стоит
Определение требований к системе	Точное определение требований к системе. На основании этих требований должна проводиться аттестация экспертной системы
Определение базовой структуры знаний	Выявление того, какие знания являются для системы наиболее важными. После определения базовой структуры любые изменения в нее вносятся только по формальному запросу, поскольку это определение знаний высокого уровня теперь должны стать основой следующей стадии проектирования знаний

автоматическое приобретение знаний. Подобные инструментальные средства часто предоставляются поставщиками инструментальных средств. А что касается приобретения знаний инженером по знаниям, пытающимся выявить знания, которыми обладает эксперт, то для этого требуется довольно высокая квалификация; эта тема более подробно рассматривается в [43].

Проектирование знаний

Целью **этапа проектирования знаний** является подготовка подробного проекта для экспертной системы. Ниже указаны две основные задачи, из которых состоит этот этап.

- Определение знаний.
- Составление подробного проекта.

В табл. 6.5 описаны задачи, связанные с определением знаний.

Таблица 6.5. Задачи определения знаний

Задача	Назначение
Выбор способа представления знаний	Определение формы представления знаний, такой как правила, фреймы или логические формулы, с учетом того, какую форму представления поддерживает используемое инструментальное средство разработки экспертных систем
Подробная разработка структур управления	Создание трех общих структур управления: во-первых, структур, определяющих способ вызова системы, если код системы входит в состав процедурного кода; во-вторых, структур, обеспечивающих управление взаимосвязанными группами правил в ходе эксплуатации системы; в-третьих, структур метауровня, под управлением которых действуют правила
Регламентация внутренней структуры фактов	Определение единообразной внутренней структуры фактов, позволяющей упростить понимание смысла фактов и добиться применения хорошего стиля программирования
Предварительная подготовка пользовательского интерфейса	Определение предварительных требований к пользовательскому интерфейсу. Получение от пользователей отзывов об интерфейсе
Составление начального плана проверки	Определение способов проверки кода. Регламентация состава проверочных данных, подготовка вспомогательных программ, применяемых в процессе проверки, и выбор способа анализа результатов проверки

Внутренние структуры фактов, о которых речь идет в табл. 6.5, рассматриваются более подробно в главах данной книги, посвященных языку CLIPS. При

определении структур фактов необходимо прежде всего придерживаться хорошего стиля программирования. Например, такой факт, как “10”, сам по себе содержит мало смысла, поскольку трудно сказать, что означает “10”. Если же в определение факта включается дополнительная информация, такая как “price 10” (цена, равная 10) или, что еще лучше, “gold price 10” (цена на золото, равная 10), то становится ясно, что речь идет о цене на золото. Следует отметить, что в такой форме факт соответствует обычному шаблону “объект–атрибут–значение” и поэтому становится удобным для чтения и понимания людьми. В языке CLIPS возможность использования такой формы предоставляется с помощью конструкции deftemplate для правил, а также с помощью объектов.

В некоторых языках экспертных систем поля могут иметь **строгую типизацию**, что позволяет хранить в них только определенные значения. Если же предпринимается попытка задать в правиле недопустимое значение, то машина логического вывода отмечает такую ситуацию как ошибку. В табл. 6.6 показан состав задач, решаемых на **этапе подробного проектирования** знаний.

Таблица 6.6. Задачи этапа подробного проектирования знаний

Задача	Назначение
Разработка структуры проекта	Регламентация способов логической организации знания в базе знаний и определение того, что должно находиться в базе знаний
Выбор стратегии реализации	Определение того, как должна осуществляться реализация системы
Подробная разработка пользовательского интерфейса	Подробное определение пользовательского интерфейса на основании изучения отзывов пользователей в отношении предварительного проекта пользовательского интерфейса
Подготовка спецификаций проекта и составление отчетов	Документальное оформление результатов проекта
Составление подробного плана проверки	Точное определение того, как должна осуществляться проверка и аттестация кода

В результате выполнения этапа подробного проектирования создается конкретизированный проектный документ, на основании которого можно непосредственно приступать к разработке кода. Но до начала разработки кода этот конкретизированный проектный документ в качестве заключительной проверки подвергается рецензированию как проект системы знаний.

Разработка кода и отладка

В табл. 6.7 представлены задачи **этапа разработки кода и отладки**, с которого начинается фактическая реализация кода.

Таблица 6.7. Задачи разработки кода и отладки

Задача	Назначение
Разработка кода	Фактическое осуществление разработки кода
Проверка	Проверка кода с использованием проверочных данных, вспомогательных программ проверки и процедур анализа результатов проверки
Подготовка распечаток программ	Подготовка снабженного комментариями и документированного исходного кода
Подготовка пользовательской документации	Подготовка рабочей пользовательской документации, с помощью которой эксперты и пользователи могли бы быстрее осваивать систему
Подготовка руководств по инсталляции и эксплуатации	Документирование применяемых пользователями процедур инсталляции и эксплуатации системы
Составление документа с описанием системы	Общее описание функциональных возможностей, ограничений и задач, связанных с эксплуатацией экспертной системы

Этот этап завершается **проверкой готовности к испытаниям**, в ходе которой определяется готовность экспертной системы к проведению следующего этапа — верификации знаний.

Верификация знаний

Этап верификации знаний имеет своей целью определение правильности, полноты и непротиворечивости системы. Этот этап подразделяется на две основные задачи, приведенные ниже.

- Проведение формальных проверок.
- Анализ результатов испытаний.

В табл. 6.8 описан состав **задач формальной проверки** на этапе верификации знаний.

В табл. 6.9 показаны **задачи анализа результатов проверки**. В процессе анализа результатов проверки предпринимаются попытки выявить следующие основные недостатки:

- неправильные ответы;
- неполные ответы;

Таблица 6.8. Состав задач формальной проверки на этапе верификации знаний

Задача	Назначение
Осуществление процедур проверки	Реализация процедур формальной проверки
Подготовка отчетов по результатам проверки	Документирование результатов проверки

- несовместимые ответы.

Кроме того, в ходе решения этих задач необходимо определить, обусловлены ли эти недостатки ошибками в правилах, цепях логического вывода, в оценке неопределенности, или являются следствием воздействия некоторой комбинации этих трех факторов. Если причины обнаруженного недостатка невозможно выявить путем анализа самой экспертной системы, то необходимо проанализировать на наличие программных ошибок программное обеспечение инструментального средства разработки экспертных систем, но к такой мере следует прибегать только в крайнем случае. В отличие от других инструментальных средств для языка CLIPS предоставляется весь исходный код и все руководства, поэтому всегда есть возможность дойти до первопричины ошибки.

Таблица 6.9. Задачи анализа результатов проверки

Задача	Назначение
Оценка результатов	Анализ результатов проверки
Подготовка рекомендаций	Документальное оформление рекомендаций и заключений по результатам проверок

Оценка системы

Как показано в табл. 6.10, конечным этапом жизненного цикла разработки является **этап оценки системы**. Назначение этого этапа состоит в подведении итогов того, какие выводы были сделаны на основании рекомендаций по усовершенствованию и внесению исправлений.

Как уже было сказано, обычно разработка экспертной системы осуществляется путем последовательного наращивания ее возможностей, поэтому, как правило, отчет, составленный по результатам выполнения этапа оценки системы, становится промежуточным отчетом, в котором дано описание функциональных возможностей системы, возросших в результате введения новых знаний. Тем не менее новые возможности системы необходимо проверять не только отдельно, но и в сочетании со знаниями системы, которые были введены в нее ранее. Это означает, что и верификация системы должна выполняться с учетом всех знаний

Таблица 6.10. Задачи этапа оценки системы

Задача	Назначение
Оценка результатов	Подведение итогов по результатам проверки и верификации
Выработка рекомендаций	Подготовка рекомендаций по внесению изменений в систему
Аттестация	Проведение аттестации, позволяющей определить, что система действует правильно, а также соответствует потребностям и пожеланиям пользователей
Подготовка промежуточного или окончательного отчета	Если разработка системы завершена, то составляется окончательный отчет. В противном случае составляется промежуточный отчет и определяются потребности в дальнейшем финансировании

системы, а не только новых знаний. В идеальном случае после каждого перехода на данный этап должна также проводиться аттестация экспертной системы, чтобы не приходилось ожидать завершения последней итерации разработки. Но при таком подходе возникает опасность непроизводительного использования рабочего времени эксперта.

6.8 Резюме

В этой главе обсуждался программотехнический подход к разработке экспертных систем. Изложены основные принципы качественного проведения собеседований с экспертом. В наши дни технология экспертных систем широко используется для решения практических задач, поэтому к качеству экспертных систем предъявляются очень высокие требования, особенно в связи с тем, что многие из них используются в оборонительных системах и в кредитных агентствах.

При проектировании экспертной системы необходимо рассмотреть целый ряд факторов, таких как состав задач, ограничения по стоимости и вероятный выигрыш. Успешное создание системы становится невозможным, если не учитываются и управленческие, и технические аспекты [40].

Одним чрезвычайно полезным понятием, применяемым при разработке программного обеспечения, является понятие жизненного цикла экспертной системы. При использовании подхода, основанного на определении жизненного цикла, разработка программного обеспечения рассматривается как ряд стадий существования программного обеспечения, начинающихся с формулировки начальной концепции и заканчивающихся его переводом в категорию непригодных для дальнейшего использования. (Однако создается впечатление, что при современной организации бизнеса, когда основные доходы приносит исправление ошибок и (или)

выпуск усовершенствованных версий, более выгодно никогда не отказываться от дальнейшего сопровождения программного обеспечения в связи с его устареванием.) Практика показала, что неизменное соблюдение всех требований, связанных с осуществлением подхода на основе жизненного цикла, позволяет успешно создавать качественное программное обеспечение. Обсуждалось несколько разных моделей жизненного цикла, применимых для разработки экспертных систем, и приведено подробное описание каждой из этих моделей.

В рамках проекта RuleXML и инициативы по созданию языка RuleML (Rule Markup Language) ведутся работы по обеспечению машинного чтения правил (`http://www.ruleml.org`).

Задачи

6.1. Рассмотрите простую систему, основанную на знаниях, предназначенную для диагностирования неисправностей в автомобиле. Опишите эту предлагаемую систему в том виде, который подходит для использования на каждой стадии линейной модели. (Для этого не требуется детализация вплоть до отдельной задачи.) Исходите из того, что над проектом работает много людей, поэтому предусмотрите возможность координации их усилий. Дайте пояснения по поводу всех принятых вами предположений.

6.2. Составьте отчет о выполнении задачи 6.1 с использованием коммерческого автоматизированного инструментального средства для управления знаниями. Загрузите его испытательную версию, проведя поиск либо в приложении Ж, либо в Web. Перечислите основные проблемы, которые удалось обнаружить с его помощью, и укажите стоимость их устранения.

6.3. Объясните, в чем состоят изменения или различия в линейных моделях, используемых для разработки крупного проекта с участием многочисленных разработчиков или небольшого проекта, только с одним участником.

Глава 7

Введение в CLIPS

7.1 Введение

В начале данной главы приведено вводное описание практических понятий, необходимых для формирования экспертной системы. В главах 1–6 даны основные сведения, представлена история развития, изложены основные определения, описана терминология, расшифрованы основные понятия, а также перечислены наиболее характерные инструментальные средства и приложения экспертных систем. Короче говоря, в первых главах изложен материал, позволяющий получить основные сведения о том, что представляют собой экспертные системы и какие задачи они позволяют решать. Эта теоретическая инфраструктура понятий и алгоритмов необходима для создания экспертных систем. Однако в процессе создания экспертных систем приходится сталкиваться со многими практическими аспектами, которые необходимо изучать путем самостоятельного выполнения соответствующих действий. Процесс создания экспертной системы во многом напоминает процесс написания программы на процедурном языке. Ведь недостаточно знать, как действует алгоритм, чтобы быть способным написать процедурную программу для реализации этого алгоритма. Аналогичным образом, недостаточно овладеть знаниями эксперта, чтобы приобрести способность к созданию экспертной системы. По этой причине в процессе изучения всей тематики экспертных систем становится бесценным практический опыт использования инструментальных средств создания экспертных систем.

В оставшейся части данной книги для демонстрации различных концепций будет использоваться язык экспертных систем CLIPS. Язык CLIPS поддерживает подходы к программированию трех типов: основанные на правилах, объектно-ориентированные и процедурные. Программирование, основанное на правилах, рассматривается в главах 7–9. В главе 10 описано процедурное программирование.

Глава 11 посвящена объектно-ориентированному программированию. Наконец, в главе 12 обсуждается несколько примеров программ, которые демонстрируют возможности, представленные в предыдущих главах. В целом главы 7–12 отражают мнение авторов в отношении того, какие средства CLIPS являются наиболее полезными и в каком контексте они должны использоваться. Безусловно, эти главы и связанные с ними приложения могут применяться в качестве справочника по языку программирования CLIPS, но основным их назначением является обучение читателя написанию программ CLIPS. Поэтому оставшуюся часть книги не следует рассматривать как материал, в котором приведено исчерпывающее описание каждой подробности каждого средства, предусмотренного в языке CLIPS. А для использования в качестве авторитетного справочника по языку программирования CLIPS предназначен документ *CLIPS Basic Programming Guide*, представленный в электронном виде на компакт-диске, прилагаемом к данной книге. Этот документ полностью соответствует своему назначению как справочного руководства, поэтому может принести наибольшую пользу тем программистам, которые уже знают, как писать программы CLIPS, и нуждаются в более точной информации о конкретном средстве.

В настоящей главе описаны основные компоненты экспертной системы, основанной на правилах (см. главу 1), входящих в состав CLIPS. Эти основные компоненты перечислены ниже.

1. **Список фактов** содержит данные, на основании которых формируются логические выводы.
2. **База знаний** содержит все правила.
3. **Машина логического вывода** обеспечивает общее управление процессом выполнения программы.

Настоящая глава в основном посвящена описанию этих трех компонентов CLIPS. Особенно подробно рассматривается первый компонент — факты. Вначале описаны операции добавления, удаления, модификации, дублирования, отображения и трассировки фактов, а затем дано объяснение того, как правила взаимодействуют в программе CLIPS с фактами для обеспечения функционирования программы. После этого рассматривается тема использования переменных и подстановочных символов в операциях сопоставления с шаблонами. Наконец, для демонстрации взаимодействия многочисленных правил применяется программа “мира блоков”, основанная на использовании средств, которые описаны в данной главе.

7.2 Язык CLIPS

CLIPS — это язык программирования, позволяющий использовать целый ряд подходов, который обеспечивает поддержку программирования на основе пра-

вил, объектно-ориентированного и процедурного программирования. Возможности логического вывода и представления, предоставляемые основанным на правилах языком программирования CLIPS, аналогичны возможностям языка OPS5, но являются более мощными. По своей синтаксической структуре правила CLIPS весьма напоминают правила, применяемые в таких языках, как Eclipse, CLIPS/R2 и Jess, но CLIPS поддерживает только правила прямого логического вывода. Этот язык не обеспечивает обратный логический вывод.

Возможности объектно-ориентированного программирования языка CLIPS, упоминаемые под общим названием как язык COOL, представляют собой гибридную комбинацию средств, обнаруживаемых в других объектно-ориентированных языках, таких как CLOS (Common Lisp Object System — общая объектная система Lisp) и SmallTalk; кроме того, в языке COOL воплощены некоторые новые идеи. С другой стороны, средства языка процедурного программирования, предусмотренные в CLIPS, напоминают средства таких языков, как C, Ada и Pascal, а по своему синтаксису аналогичны языку LISP.

Язык CLIPS (название которого представляет собой сокращение от C Language Integrated Production System — продукционная система, интегрированная с языком C) был разработан с использованием языка программирования C в Космическом центре NASA/Джонсон. Перед разработчиками этого языка была поставлена конкретная задача — обеспечить полную переносимость, низкую стоимость и простую интеграцию с внешними системами. Но компоненты расшифровки аббревиатуры CLIPS не следует трактовать буквально. Дело в том, что первоначально CLIPS обеспечивал поддержку только программирования на основе правил (отсюда происходит часть его обозначения как “продукционной системы”). Но в версии 5.0 языка CLIPS введена поддержка процедурного и объектно-ориентированного программирования. Кроме того, может сбить с толку и часть этого сокращенного обозначения, в которой упоминается “язык C”, поскольку одна из версий CLIPS разработана полностью на языке Ada. К настоящей книге прилагается компакт-диск, содержащий исполняемые файлы версии CLIPS 6.2 для DOS, Windows 2000/XP и MacOS, документацию в электронном виде и исходный код на языке C для CLIPS.

Язык CLIPS действительно обеспечивает полную переносимость, поэтому может быть устанавливаться на компьютерах самых разных типов, начиная с персональных компьютеров и заканчивая суперкомпьютерами CRAY. Большинство примеров, приведенных в этой и последующих главах, должно работать на любом компьютере, на котором установлен язык CLIPS. Но рекомендуется, чтобы пользователи CLIPS обладали определенными знаниями об операционной системе того компьютера, на котором эксплуатируется CLIPS. Например, от типа операционной системы зависит метод обозначения имен файлов. Если рассматриваемые команды могут зависеть от типа компьютера или операционной системы, об этом дается соответствующее упоминание.

7.3 Система обозначений

В данной главе и в следующих главах для описания синтаксиса различных команд и конструкций, рассматриваемых в ходе изложения, используется одна и та же система обозначений. Эта система обозначений состоит из трех различных типов текста, подлежащего вводу.

Обозначения первого типа относятся к символам и знакам, которые должны быть введены точно так, как они показаны; к ним относятся любые текстовые надписи, не заключенные в пару знаков <>, [] или {}. Например, рассмотрим следующее описание синтаксиса:

```
(example)
```

Это описание синтаксиса означает, что конструкция `(example)` должна быть введена так, как показано. Точнее, вначале должен быть введен знак открывающей скобки `(`, затем буква `e`, после этого буквы `x`, `a`, `m`, `p`, `l`, `e` и, наконец, знак закрывающей скобки, `)`.

Квадратные скобки, `[]`, указывают, что содержимое в квадратных скобках является необязательным. Например, следующее описание синтаксиса показывает, что цифра `1`, находящаяся в квадратных скобках, может не указываться:

```
(example [1])
```

Таким образом, следующий результат ввода является совместимым с указанным выше синтаксическим определением:

```
(example)
```

как и такой результат ввода:

```
(example 1)
```

Знаки “меньше” и “больше”, вместе взятые, <>, указывают, что должна быть выполнена замена значением того типа, который обозначен содержимым, находящимся внутри знаков <>. Например, следующее описание синтаксиса, в котором используются знаки “меньше” и “больше”, показывает, что должна быть выполнена замена действительным целочисленным значением:

```
<integer>
```

Продолжая предыдущие примеры, укажем, что такое описание синтаксиса:

```
(example <integer>)
```

может быть заменено следующими результатами ввода:

```
(example 1)
```

или:

```
(example 5)
```

или:

```
(example -20)
```

или многими другими результатами ввода, в которых содержатся знаки "`(example `", за этими знаками следует целое число, а за ним — знак `)`. Важно отметить, что пробелы, показанные в описании синтаксиса, также должны быть включены в результат ввода.

Еще один вариант обозначения характеризуется использованием звездочки, `*`, которая следует за описанием. Такое обозначение показывает, что описание может быть заменено вхождениями указанного значения в количестве от нуля или больше. После каждого вхождения некоторого значения должны быть проставлены пробелы. Например, следующее описание синтаксиса:

```
<integer>*
```

может быть заменено таким результатом ввода:

```
1
```

или таким:

```
1 2
```

или таким:

```
1 2 3
```

или любым другим количеством разделенных пробелами целых чисел, или же вообще оставлено пустым.

Описание, за которым следует знак "плюс", `+`, указывает, что вместо этого описания синтаксиса должно быть введено одно или несколько значений, заданных этим описанием. Необходимо отметить, что следующее описание синтаксиса:

```
<integer>+
```

эквивалентно такому описанию синтаксиса:

```
<integer> <integer>*
```

Вертикальная черта, `|`, указывает на необходимость выбора одного или нескольких элементов, разделенных вертикальными чертами. Например, следующее описание синтаксиса:

```
all | none | some
```

может быть заменено таким результатом ввода:

```
all
```

или таким:

```
none
```

или таким:

```
some
```

7.4 Поля

В процессе создания базы знаний интерпретатор CLIPS считывает входные данные с клавиатуры и из файлов, чтобы выполнить команды и загрузить программу. По мере считывания отдельных знаков с клавиатуры или из файлов интерпретатор CLIPS группирует эти знаки в **лексемы**. Лексемы представляют собой группы знаков, имеющих в языке CLIPS особый смысл. Некоторые лексемы, такие как левая и правая круглые скобки, состоят только из одного знака.

Особое значение имеет группа лексем, называемых **полями**. В языке CLIPS предусмотрено восемь типов полей, которые принято также называть примитивными типами данных: **число с плавающей точкой**, **целое число**, **символ**, **строка**, **внешний адрес**, **адрес факта**, **имя экземпляра** и **адрес экземпляра**.

Поля первых двух типов, числа с плавающей точкой и целые числа, называются **числовыми полями**, или просто **числами**. Числовое поле может состоять из трех частей: знак, значение и показатель степени. Знак и показатель степени являются необязательными. В качестве знака может быть указан + или −. Значение состоит из одной или нескольких цифр, между которыми может находиться единственная необязательная десятичная точка. Показатель степени состоит из буквы e или E, за которой следует необязательный знак + или −, сопровождаемый одной или несколькими цифрами. Любое число, состоящее из необязательного знака + или −, за которым следуют только цифры, записывается в память в виде **целого числа**. Все другие числа хранятся как **числа с плавающей точкой**.

Ниже приведены примеры допустимых чисел с плавающей точкой CLIPS.

```
1.5
1.0
0.7
9e+1
3.5e10
```

А следующие примеры показывают допустимые целые числа CLIPS:

```
1
+3
-1
65
```

Символ — это поле, которое начинается с пригодного для печати знака кода ASCII и сопровождается другими пригодными для печати знаками в количестве от нуля или больше. Конец символа достигается при обнаружении **разграничителя**. К разграничителям относятся любые непечатаемые знаки кода ASCII (включая пробелы, знаки табуляции, знаки возврата каретки и перевода строки), знак " (двойная кавычка), знак ((открывающая круглая скобка), знак) (закрывающая круглая скобка), знак ; (точка с запятой), знак & (амперсанд), знак | (вертикальная

черта), знак ~ (тильда) и знак < (меньше). Символы не могут содержать разграничителей (за исключением знака <, который может быть первым знаком в символе). Кроме того, знак ? и последовательность знаков $? (вопросительный знак и знак доллара с вопросительным знаком) нельзя помещать в начало символа, поскольку они используются для обозначения *переменных* (см. раздел 7.19). К тому же как символ рассматривается последовательность знаков, которые не соответствуют точно формату числового поля.

Ниже приведены примеры допустимых символов.

```
Space-Station
February
fire
activate_sprinkler_system
notify-fire-department
shut-down-electrical-junction-387
!?#$^*
345B
346-95-6156
```

Обратите внимание на то, как используются знаки подчеркивания и дефисы для соединения символов друг с другом и преобразования их в одно поле.

Интерпретатор CLIPS сохраняет в неизменном виде прописные и строчные буквы, обнаруженные им в лексемах. Таким образом, язык CLIPS предусматривает проведение различий между буквами верхнего и нижнего регистров, поэтому называется **чувствительным к регистру**. Например, такие символы рассматриваются в языке CLIPS как различные:

```
case-sensitive
Case-Sensitive
CASE-SENSITIVE
```

Следующим типом поля является **строка**. Строка должна начинаться и заканчиваться двойными кавычками, которые являются частью поля. Между двойными кавычками могут находиться от нуля и больше знаков, включая те, которые обычно используются в языке CLIPS как разграничители. Ниже приведены примеры допустимых строк CLIPS.

```
"Activate the sprinkler system."
"Shut down electrical junction 387."
"!?#$^"
"John Q. Public"
```

Пробелы обычно используются в языке CLIPS как разграничители для разделения полей (таких как символы) и других лексем. Дополнительные пробелы, присутствующие между лексемами, отбрасываются. Но пробелы, входящие в со-

став строки, сохраняются. Например, в языке CLIPS следующие четыре строки рассматриваются как различные:

```
"spaces"
"spaces "
" spaces"
" spaces "
```

Если бы окружающие строку двойные кавычки были удалены, то интерпретатор CLIPS рассматривал бы каждую из этих строк ввода как содержащую один и тот же символ, поскольку все пробелы, отличные от тех, которые используются в качестве разграничителей, игнорируются.

Еще раз отметим, что *двойные кавычки* применяются для обозначения начала и конца строк, поэтому двойные кавычки нельзя непосредственно помещать в саму строку. Например, строка

```
" "three-tokens" "
```

обрабатывается интерпретатором CLIPS как три отдельные лексемы следующим образом, поскольку двойные кавычки рассматриваются и как обозначения строк, и как разграничители:

```
" "
three-tokens
" "
```

Двойные кавычки можно включать в строку с использованием знака *операции переключения* на другой режим обработки, т.е. обратной косой черты, \. Например, следующая строка:

```
"\"single-token\""
```

обрабатывается интерпретатором CLIPS как такое строковое поле:

```
""single-token""
```

При этом создается только одно поле, поскольку знак обратной косой черты исключает возможность обработки следующей за ним двойной кавычки как разграничителя. Сам знак обратной косой черты может быть задан в строке с использованием двух подряд идущих знаков обратной косой черты. Например, строка

```
"\\single-token\\"
```

обрабатывается интерпретатором CLIPS как такое строковое поле:

```
"\single-token\"
```

Во вступительном описании программных возможностей языка CLIPS поле следующего типа, **внешний адрес**, еще нельзя представить достаточно подробно. Поле внешнего адреса содержит адрес внешней структуры данных, возвращаемой **функцией, определяемой пользователем** (функция, написанная на таком языке,

как C или Ada, и присоединенная с помощью редактора связей к интерпретатору CLIPS в целях приобретения дополнительных функциональных возможностей). Безусловно, значение внешнего адреса невозможно задать с помощью последовательности знаков, образующих лексему, а основная немодифицированная версия CLIPS не содержит функций, возвращающих внешние адреса, поэтому в основной немодифицированной версии CLIPS невозможно создать поле такого типа.

Оставшимися тремя типами полей являются **адрес факта**, **адрес экземпляра** и **имя экземпляра**. Как будет вскоре описано, факты относятся к одному из тех сложных представлений данных, которые предусмотрены в языке CLIPS. Адрес факта используется для получения ссылки на конкретный факт. Как и внешний адрес, адрес факта не может быть задан с помощью последовательности знаков, образующей лексему. Но предусмотрена возможность получать адреса фактов с помощью правил в составе процесса сопоставления с шаблонами. Синтаксические конструкции, применяемые для обеспечения такого действия, рассмотрены в разделе 7.21.

Еще одним сложным представлением данным, предусмотренным в языке CLIPS, являются *экземпляры*, которые рассматриваются более подробно в главе 11. На экземпляры можно ссылаться с использованием либо адреса экземпляра, либо имени экземпляра. Адрес экземпляра подобен адресу факта, но ссылается на экземпляр, а не на факт. Предусмотрена также возможность ссылаться на экземпляр с помощью имени. *Имя экземпляра* — это символ специального типа, который заключен в левую и правую квадратные скобки. Например, `[pump-1]` представляет собой имя экземпляра.

Ряды, состоящие из полей, содержащихся вместе в одной синтаксической позиции в количестве от нуля и больше, называются **многозначным значением**. Многозначные значения обычно создаются путем вызова некоторой функции (как будет показано в следующих главах) или путем задания ряда полей. Во время вывода на внешнее устройство многозначное значение заключается в левую и правую круглые скобки. Например, многозначное значение нулевой длины выводится следующим образом:

```
()
```

а многозначное значение, содержащее символы `this` и `that`, выводится так:

```
(this that)
```

7.5 Вход и выход из системы CLIPS

В систему CLIPS можно войти путем выдачи соответствующей команды вызова интерпретатора на выполнение для компьютера, на котором установлена система CLIPS. После этого должно появиться приглашение CLIPS, которое выглядит следующим образом:

```
CLIPS>
```

С этого момента появляется возможность непосредственно вводить команды в систему CLIPS; этот режим называется режимом **верхнего уровня**.

Нормальный способ выхода из системы CLIPS состоит в применении **команды exit**. Эта команда имеет следующий синтаксис:

```
(exit)
```

Обратите внимание на то, что символ `exit` заключен в согласованные (парные) круглые скобки. Многие языки, основанные на правилах, происходят от языка LISP, в котором в качестве разграничителей применяются *круглые скобки*. А так как CLIPS основан на языке, первоначально разработанном с помощью машин LISP, в нем сохранились те же разграничители. Символ `exit` без охватывающих круглых скобок имеет совсем не такой смысл, как символ `exit` с охватывающими круглыми скобками. Круглые скобки вокруг слова `exit` указывают на то, что `exit` — *команда*, подлежащая выполнению, а не просто символ. Как будет показано ниже, круглые скобки служат в качестве важных разграничителей для команд.

А на данный момент важно запомнить лишь то, что каждая команда CLIPS должна иметь согласованное количество левых и правых круглых скобок.

Окончательный этап вызова на выполнение команды CLIPS после ее ввода со сбалансированным должным образом количеством круглых скобок состоит в нажатии *клавиши ввода*. Клавиша ввода может быть также нажата до или после ввода любой лексемы. Например, нажатие клавиши ввода после нажатия на клавиатуре знаков “`ex`”, но до нажатия знаков “`it`”, приводит к созданию двух лексем: лексемы, соответствующей символу `ex`, и лексемы для символа `it`.

Ниже приведена последовательность команд, которая демонстрирует *пример сеанса*, состоящего из этапов входа в систему CLIPS, обработки значения поля константы, вычисления функции, а затем выхода с помощью команды `exit`. Показанный пример относится к той ситуации, когда используется персональный компьютер класса IBM с операционной системой MS-DOS, на котором исполняемый файл CLIPS хранится на гибком диске, установленном в дисководе `A`, причем текущим дисководом также является `A`. Подразумевается, что исполняемый файл CLIPS имеет имя `CLIPSDOS`. Вывод, отображаемый операционной системой MS-DOS или системой CLIPS, показан обычным шрифтом. С другой стороны, все входные данные, которые должны быть набраны пользователем, показаны полужирным шрифтом, а нажатие клавиши ввода обозначено знаком ↵. Следует помнить, что язык CLIPS чувствителен к регистру, поэтому важно, чтобы буквы верхнего и нижнего регистров вводились так, как они показаны в примерах.

```
A:\>CLIPSDOS↵
         CLIPS (V6.22 06/15/04)
CLIPS> exit↵
```

```
exit
CLIPS> (+ 3 4)↵
7
CLIPS> (exit)↵
A:\>
```

Работая в режиме верхнего уровня, система CLIPS принимает входные данные от пользователя и осуществляет попытки обработать эти входные данные, чтобы правильно определить соответствующий ответ. Поля, введенные без какой-либо сопровождающей информации, рассматриваются как константы, а результатом обработки константы становится сама константа. Поэтому после ввода отдельно взятого символа `exit`, за чем следует нажатие клавиши ввода, интерпретатор CLIPS обрабатывает эти входные данные и в качестве результата отображается символ `exit`. А символ, заключенный в круглые скобки, рассматривается как команда или вызов функции. Таким образом, результатом ввода данных `(+ 3 4)` становится *вызов функции* `+`, которая осуществляет сложение. Параметрами этой функции являются значения `3` и `4`. Возвращаемым значением для данного вызова функции является значение `7` (возвращаемые значения будут рассматриваться более подробно в главе 8). Наконец, в результате ввода данных `(exit)` вызывается команда `exit`, выполнение которой приводит к выходу из системы CLIPS. Термины "функция" и "команда" следует рассматривать как взаимозаменяемые. Кроме того, во всей оставшейся части книги термин "функция" используется для указания на то, что происходит возврат некоторого значения, а термин "команда" — для того чтобы показать, что либо не происходит возврат значения, либо соответствующее действие обычно выполняется после ввода данных в приглашении верхнего уровня.

7.6 Факты

Для успешного решения задачи программе CLIPS должны быть предоставлены данные, или информация, на основании которых программа могла бы формировать рассуждения. "Фрагменты" информации, применяемые в языке CLIPS, называются **фактами**. Факты состоят из **имени отношения** (символического поля), за которым следует от нуля и больше **слотов** (также символических полей) и связанных с ними значений. Пример факта приведен ниже.

```
(person (name "John Q. Public")
        (age 23)
        (eye-color blue)
        (hair-color black))
```

Весь факт, как и каждый слот, ограничивается открывающей (левой) круглой скобкой и закрывающей (правой) круглой скобкой. Символ `person` представляет

собой имя отношения для рассматриваемого факта, а сам факт содержит четыре слота: name (имя), age (возраст), eye-color (цвет глаз) и hair-color (цвет волос). Значением слота name является "John Q. Public", значением слота age — число 23, значением слота eye-color — blue, а значением слота hair-color — black. Следует отметить, что порядок, в котором заданы слоты, не имеет значения. В частности, следующий факт рассматривается интерпретатором CLIPS как идентичный первому приведенному факту person:

```
(person (hair-color black)
        (name "John Q. Public")
        (eye-color blue)

        (age 23))
```

Конструкция deftemplate

Прежде чем появится возможность создать факт, интерпретатору CLIPS необходимо сообщить информацию о том, каковым является список допустимых слотов для отношения с указанным именем. Группы фактов, объединяемых под одним и тем же именем отношения и содержащих общую информацию, можно описать с использованием **конструкции deftemplate**. **Конструкции** образуют ядро программы CLIPS, поскольку вводят знания программиста в среду CLIPS, и этим отличаются от функций и команд. Конструкция deftemplate аналогично структуре записи в таких языках, как Pascal. Общий формат конструкции deftemplate является таковым:

```
(deftemplate <relation-name> [<optional-comment>]
   <slot-definition>*)
```

Описание синтаксиса <slot-definition> определяется следующим образом:

```
(slot <slot-name>) | (multislot <slot-name>)
```

С использованием этого синтаксиса факт person можно описать с помощью следующей конструкции deftemplate:

```
(deftemplate person "An example deftemplate"
   (slot name)
   (slot age)
   (slot eye-color)
   (slot hair-color))
```

Многозначные слоты

Допускается, чтобы слоты факта, которые были заданы с помощью ключевого слова `slot` в соответствующих им конструкциях `deftemplate`, содержали только одно значение (такие слоты называются *однозначными слотами*). Тем не менее часто возникает необходимость поместить в какой-то конкретный слот от нуля и больше полей. Допускается, чтобы слоты факта, которые были заданы с помощью **ключевого слова multislot** в соответствующих конструкциях `deftemplate`, содержали от нуля и больше значений (такие слоты называются *многозначными слотами*). Например, в слоте `name` конструкции `deftemplate person` имя данного лица хранится как единственное строковое значение. А если бы слот `name` был определен с использованием ключевого слова `multislot` вместо ключевого слова `slot`, то в этом слоте можно было бы хранить любое количество полей. Таким образом, следующий факт является недопустимым, если слот `name` однозначен, и допустимым, если слот `name` многозначен:

```
(person (name John Q. Public)
        (age 23)
        (eye-color blue)
        (hair-color brown))
```

Любая конструкция `deftemplate` может содержать любую комбинацию однозначных и многозначных слотов.

Упорядоченные факты

Факты с именем отношения, имеющие соответствующую конструкцию `deftemplate`, называются **фактами с конструкцией deftemplate**. С другой стороны, факты с именем отношения, не имеющие соответствующей конструкции `deftemplate`, называются **упорядоченными фактами**. Под этим подразумевается, что упорядоченные факты имеют единственный многозначный слот, используемый для хранения всех значений, следующих за именем отношения. В действительности при обнаружении интерпретатором CLIPS каждого упорядоченного факта автоматически создается **подразумеваемая конструкция deftemplate** для этого факта (в отличие от **явной конструкции deftemplate**, создаваемой с применением определения `deftemplate`). Любой упорядоченный факт имеет только один слот, поэтому при определении такого факта имя слота не требуется. Например, с помощью следующего факта может быть представлен список чисел:

```
(number-list 7 9 3 4 20)
```

По существу это определение эквивалентно определению такой конструкции `deftemplate`:

```
(deftemplate number-list (multislot values))
```

и последующему заданию факта таким образом:

```
(number-list (values 7 9 3 4 20))
```

Вообще говоря, при любой возможности следует использовать факты с конструкцией `deftemplate`, поскольку благодаря наличию имен слотов факты становятся более пригодными для чтения, а работа с ними облегчается. Тем не менее можно назвать два случая, в которых полезны и упорядоченные факты. Во-первых, факты, состоящие только из имени отношения, могут применяться в качестве *флажков* и выглядят одинаково, независимо от того, была для них определена конструкция `deftemplate` или нет. Например, следующий упорядоченный факт может использоваться в качестве флажка для указания на то, что все заказы обработаны:

```
(all-orders-processed)
```

Во-вторых, для факта, содержащего единственный слот, имя слота обычно становится синонимом имени отношения. Например, следующие факты:

```
(time 8:45)
(food-groups meat dairy bread fruits-and-vegetables)
```

несут в себе не меньше смысла, чем такие факты:

```
(time (value 8:45))
(food-groups (values meat dairy bread
                     fruits-and-vegetables))
```

Связи между терминами, рассматриваемыми в этом разделе, представлены графически на рис. 7.1.

7.7 Добавление и удаление фактов

Все факты, известные системе CLIPS, группируются и сохраняются в списке фактов. Этот список фактов позволяет добавлять и удалять факты, представляющие информацию. Новые факты могут быть добавлены к списку фактов с использованием **команды assert**. Команда `assert` имеет следующий синтаксис:

```
(assert <fact>+)
```

В качестве примера воспользуемся конструкцией `deftemplate` с именем `person` для описания некоторых людей с помощью фактов. Информацию о человеке с именем `John Q. Public` можно добавить к списку фактов с помощью следующих команд:

```
CLIPS>
(deftemplate person
   (slot name)
   (slot age)
```

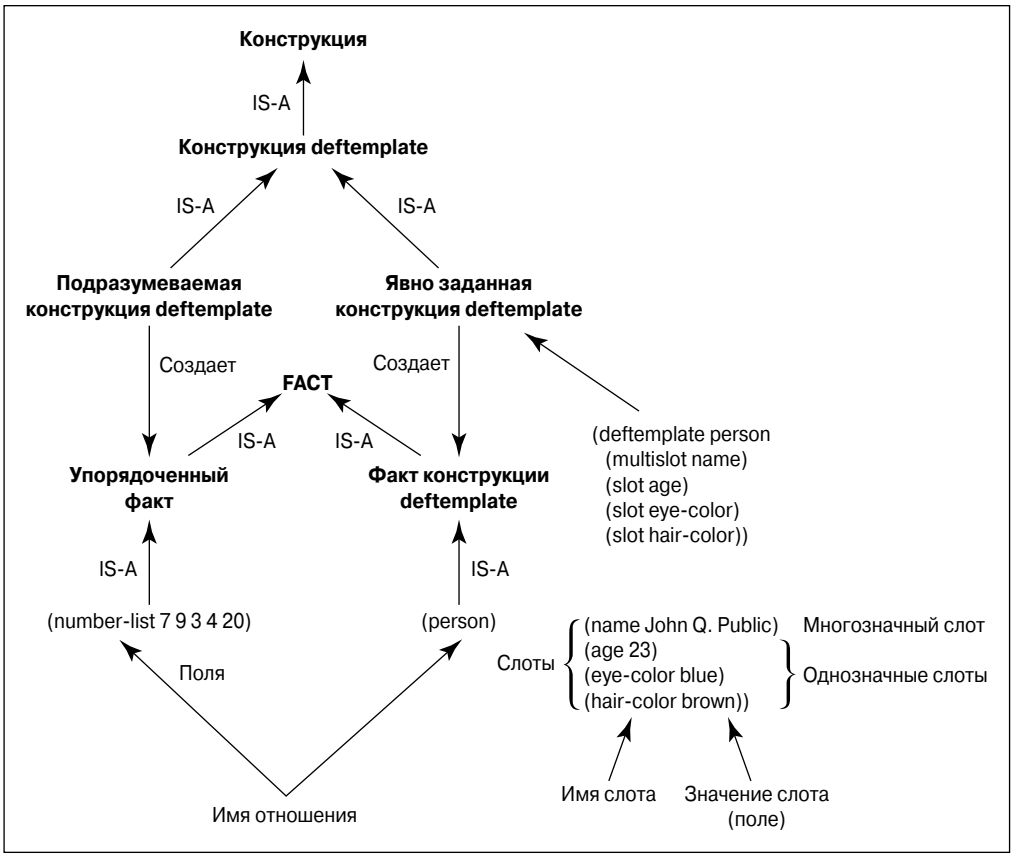


Рис. 7.1. Краткий обзор терминов, применяемых при описании конструкции `deftemplate`

```
   (slot eye-color)
   (slot hair-color)).↵
CLIPS>
(assert (person (name "John Q. Public")
                (age 23)
                (eye-color blue)
                (hair-color black))).↵
<Fact-0>
CLIPS>
```

Обратите внимание на то, что команда `assert` возвращает значение, `<Fact-0>`. Способы применения возвращаемых значений будут описаны в главе 8.

Для отображения фактов, находящихся в списке фактов, можно воспользоваться **командой facts**. Основной синтаксис команды `facts` является таковым:

```
(facts)
```

А ниже приведен пример применения этой команды.

```
CLIPS> (facts)↵
f-0     (person (name "John Q. Public")
                (age 23)
                (eye-color blue)
                (hair-color black))
For a total of 1 fact.
CLIPS>
```

Терм "f-0" представляет собой **идентификатор факта**, присвоенный этому факту системой CLIPS. Каждому факту, вставляемому в список фактов, присваивается уникальный идентификатор факта, начинающийся с буквы f и заканчивающийся целым числом, которое называется **индексом факта**. Индекс факта 0 был показан также в возвращаемом значении <Fact-0> показанной выше команды assert. Обратите внимание на то, что, как показано в этой книге, к выводу команды facts были добавлены дополнительные пробелы между слотами (age 23) и (eye-color blue) для повышения удобства чтения. Но обычно система CLIPS помещает только один пробел между слотами и переносит строку вывода с крайнего правого столбца на крайний левый столбец. В других примерах вывода CLIPS, приведенных в настоящей книге, к выводу CLIPS иногда также добавляются пробелы для повышения удобства чтения, при условии, что такое добавление пробелов не вызывает путаницы.

При обычных обстоятельствах система CLIPS не допускает ввод *дублирующихся фактов* (хотя такое поведение системы может быть изменено, как будет показано в разделе 12.2). Поэтому попытка поместить в список фактов второй факт "John Q. Public" с идентичными значениями слотов не приведет к значимому результату. Но, безусловно, к списку фактов можно легко добавлять другие факты, не являющиеся дубликатами существующих фактов, например, как показано ниже.

```
CLIPS>
(assert (person (name "Jane Q. Public")
                (age 36)
                (eye-color green)
                (hair-color red)))↵
<Fact-1>
CLIPS> (facts)↵
f-0 (person (name "John Q. Public")
            (age 23)
            (eye-color blue)
            (hair-color black))
```

```
f-1 (person (name "Jane Q. Public")
            (age 36)
            (eye-color green)
            (hair-color red))
For a total of 2 facts.
CLIPS>
```

Как показывает синтаксис команды `assert`, с помощью единственной команды `assert` можно ввести в систему несколько фактов одновременно. Например, следующая команда добавляет в список фактов сразу два факта:

```
(assert (person (name "John Q. Public")
                (age 23)
                (eye-color blue)
                (hair-color black))
        (person (name "Jane Q. Public")
                (age 36)
                (eye-color green)
                (hair-color red)))
```

Важно помнить, что идентификаторы фактов в списке фактов не обязательно должны быть последовательными. Кроме того, факты можно не только добавлять к списку фактов, но и удалять из этого списка с помощью определенного способа. После удаления фактов из списка фактов идентификаторы удаленных фактов становятся пропущенными в списке фактов, поэтому в ходе выполнения программы CLIPS может быть нарушена строгая последовательность индексов фактов.

Количество фактов, содержащихся в списке фактов, может стать весьма значительным, поэтому часто желательно иметь возможность рассматривать только часть списка фактов. Такая возможность обеспечивается с помощью задания дополнительных параметров команды `facts`. Полный синтаксис команды `facts` приведен ниже:

```
(facts [<start> [<end> [<maximum>]]])
```

В этой команде в качестве параметров `<start>`, `<end>` и `<maximum>` применяются положительные целые числа. Следует отметить, что синтаксис команды `facts` допускает использование от нуля до трех параметров. Если параметры не заданы, отображаются все факты. Если задан параметр `<start>`, то отображаются все факты с индексами фактов, большими или равными значению `<start>`. Если заданы параметры `<start>` и `<end>`, отображаются все факты с индексами фактов, большими или равными значению `<start>` и меньшими или равными значению `<end>`. Наконец, если наряду с параметрами `<start>` и `<end>` задан параметр `<maximum>`, то отображается количество фактов, не превышающее значение `<maximum>`.

Еще раз отметим, что существует возможность не только добавлять, но и удалять факты из списка фактов. Удаление фактов из списка фактов называется **извлечением** и выполняется с помощью **команды retract**. Команда `retract` имеет следующий синтаксис:

```
(retract <fact-index>+)
```

В качестве параметров команды `retract` должны быть указаны индексы фактов, относящиеся к одному или нескольким фактам. Например, информацию о человеке по имени `John Q. Public` можно удалить из списка фактов с помощью такой команды:

```
(retract 0)
```

Аналогичным образом, следующая команда позволяет извлечь из списка фактов тот факт, который относится к `Jane Q. Public`:

```
(retract 1)
```

Попытка извлечь несуществующий факт приводит к получению следующего сообщения об ошибке (в этом сообщении ключ `[PRNTUTIL1]` представляет собой ключ для поиска описания данного сообщения об ошибке в справочном руководстве *CLIPS Reference Manual*):

```
[PRNTUTIL1] Unable to find fact <fact-identifier>.
```

В качестве примера можно указать такой фрагмент диалога системы:

```
CLIPS> (retract 1)↵
CLIPS> (retract 1)↵
[PRNTUTIL1] Unable to find fact f-1.
CLIPS>
```

Единственная команда `retract` может использоваться для одновременного извлечения нескольких фактов. Например, следующая команда извлекает факты `f-0` и `f-1`:

```
(retract 0 1)
```

7.8 Модификация и дублирование фактов

Значения слотов фактов с конструкцией `deftemplate` могут модифицироваться с помощью **команды modify**. Синтаксис команды `modify` является таковым:

```
(modify <fact-index> <slot-modifier>+)
```

В этой команде параметр `<slot-modifier>` имеет следующий вид:

```
(<slot-name> <slot-value>)
```

Например, после наступления очередного дня рождения лица по имени John Q. Public можно изменить его возраст с 23 на 24 с помощью такой команды modify:

```
CLIPS> (modify 0 (age 24))↵
<Fact-2>
CLIPS> (facts)↵
f-2     (person (name "John Q. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
For a total of 1 fact.
CLIPS>
```

Команда modify действует по принципу извлечения первоначального факта и последующего добавления нового факта после модификации указанных значений слотов. В связи с этим для модифицированного факта вырабатывается новый индекс факта.

Команда duplicate действует по такому же принципу, за исключением того, что извлечение первоначального факта не происходит. Таким образом, если найдется давно пропавший брат-близнец Джона, Джек, то информацию о нем можно будет добавить к списку фактов, продублировав факт с информацией о Джоне и изменив значение слота name, как показано ниже.

```
CLIPS> (duplicate 2 (name "Jack S. Public"))↵
<Fact-3>
CLIPS> (facts)↵
f-2     (person (name "John Q. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
f-3     (person (name "Jack S. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
For a total of 2 facts.
CLIPS>
```

Команды modify и duplicate не могут использоваться для работы с упорядоченными фактами.

7.9 Команда watch

Команда watch является полезным средством отладки программ. Результаты применения команды watch для отслеживания фактов рассматриваются в этом разделе, а остальные компоненты команды watch обсуждаются в этой и последующих главах. Команда watch имеет такой синтаксис:

```
(watch <watch-item>)
```

В этом определении <watch-item> обозначает один из символов facts, rules, activations, statistics, compilations, focus, deffunctions, globals, generic-functions, methods, instances, slots, messages, message-handlers или all.

Отслеживание этих элементов может осуществляться в любых комбинациях для получения необходимого объема отладочной информации. Команда watch может быть вызвана на выполнение несколько раз для отслеживания больше чем одного аспекта функционирования системы CLIPS. Чтобы разрешить отслеживание с помощью команды watch всех аспектов функционирования системы CLIPS, можно применить ключевое слово all. По умолчанию сразу после запуска системы CLIPS отслеживаются действия по компиляции, compilations, а остальные элементы команды watch не отслеживаются.

Если осуществляется отслеживание фактов с помощью символа facts, система CLIPS автоматически выводит сообщение, указывающее на то, что в списке фактов произошло некоторое обновление, при выполнении каждой операции вставки или удаления фактов. Применение этой команды отладки иллюстрируется в следующем командном диалоге:

```
CLIPS> (facts 3 3)↵
f-3     (person (name "Jack S. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
For a total of 1 fact.
CLIPS> (watch facts)↵
CLIPS> (modify 3 (age 25))↵
<== f-3 (person (name "Jack S. Public")
        (age 24)
        (eye-color blue)
        (hair-color black))
==> f-4 (person (name "Jack S. Public")
        (age 25)
        (eye-color blue)
        (hair-color black))
```

```
<Fact-4>
CLIPS>
```

Последовательность знаков <== указывает, что происходит извлечение факта, а последовательность знаков ==> указывает, что происходит добавление факта.

Существует возможность обеспечить отслеживание только конкретных фактов, указав одно или несколько имен конструкций deftemplate в конце команды watch. Например, выполнение команды (watch facts person) приводит к тому, что сообщения, касающиеся фактов, отображаются только применительно к факту person.

Отображение результатов, предусмотренных командой watch, можно отменить с использованием соответствующей **команды unwatch**. Команда unwatch имеет такой синтаксис:

```
(unwatch <watch-item>)
```

7.10 Конструкция deffacts

Часто удобно иметь возможность автоматически вводить множество фактов, а не набирать похожие друг на друга команды в режиме работы верхнего уровня. Это особенно относится к тем фактам, которые заведомо являются истинными еще до выполнения программы (например, начальные знания). Еще одна ситуация, в которой удобно автоматически вводить группу фактов, относится к прогону тестовых наборов для отладки программы. Группы фактов, представляющих начальные знания, можно определить с помощью **конструкции deffacts**. Например, ниже приведен оператор deffacts, предоставляющий начальную информацию о некоторых лицах, которые уже упоминались в данной главе.

```
(deffacts people "Some people we know"
(person (name "John Q. Public") (age 24)
        (eye-color blue) (hair-color black))
(person (name "Jack S. Public") (age 24)
        (eye-color blue) (hair-color black))
(person (name "Jane Q. Public") (age 36)
        (eye-color green) (hair-color red)))
```

Конструкция deffacts имеет следующий общий формат:

```
(deffacts <deffacts name> [<optional comment>]
   <facts>*)
```

Вслед за ключевым словом deffacts находится обязательное имя этой конструкции deffacts. В качестве имени может использоваться любой допустимый символ. В данном случае для конструкции выбрано имя people. За именем следует необязательный комментарий в двойных кавычках. Как и необязательный

комментарий правила, этот комментарий сохраняется в определении конструкции `deffacts` после загрузки соответствующего определения системой CLIPS. Вслед за именем или комментарием находятся факты, которые вводятся в список фактов с помощью оператора `deffacts`.

Факты, заданные в операторе `deffacts`, вводятся с помощью **команды reset** языка CLIPS. Команда `reset` удаляет все факты из списка фактов, а затем вводит факты из существующего оператора `deffacts`. Синтаксис команды `reset` является таковым:

```
(reset)
```

Предположим, что введена конструкция `deffacts` с именем `people` (после конструкции `deftemplate` с именем `person`); в таком случае формируется следующий диалог, показывающий, как происходит ввод фактов в список фактов с помощью команды `reset`:

```
CLIPS> (unwatch facts)↵
CLIPS> (reset)↵
CLIPS> (facts)↵
f-0     (initial-fact)

f-1     (person (name "John Q. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
f-2     (person (name "Jack S. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
f-3     (person (name "Jane Q. Public")
                (age 36)
                (eye-color green)
                (hair-color red))
For a total of 4 facts.
CLIPS>
```

В этом выводе показаны факты из оператора `deffacts` и новый факт, выработанный командой `reset`, который имеет имя **initial-fact**. После запуска система CLIPS автоматически определяет следующие две конструкции:

```
(deftemplate initial-fact)
(deffacts initial-fact
   (initial-fact))
```

Таким образом, даже если в программе не определены какие-либо операторы `deffacts`, команда `reset` осуществляет ввод факта `(initial-fact)`. Факт `initial-fact` всегда имеет идентификатор факта `f-0`. Польза от применения факта `(initial-fact)` заключается в том, что с него начинается выполнение программы (как описано в следующем разделе).

7.11 Компоненты правила

Экспертная система сможет выполнять содержательную работу только в том случае, если в ней присутствуют не только факты, но и правила. После обсуждения того, как происходит добавление и извлечение фактов, мы можем приступить к рассмотрению принципов действия правила.

Правила могут быть введены непосредственно в систему CLIPS или загружены из файла с правилами, созданного с помощью текстового редактора (вопрос о том, как загружаются конструкции из файла, будет рассматриваться в разделе 7.17). По-видимому, для создания любых программ, кроме самых небольших, целесообразно использовать один из встроенных редакторов, предусмотренных в системе CLIPS. Информацию об исполняемых файлах редакторов для операционных систем Windows 2000/XP и MacOS можно найти в документе *Interfaces Guide*, который представлен в электронном виде на компакт-диске, прилагаемом к данной книге. А информация о редакторе EMACS, который может использоваться в такой среде, как Unix, представлена в документе *Basic Programming Guide*. Встроенные редакторы позволяют избирательно переопределять конструкции во время разработки программы, и эта возможность является чрезвычайно полезной. Например, если во время ввода в режиме верхнего уровня какой-то конструкции будет допущена опечатка, то потребуется заново набрать текст всей конструкции. Если же эта конструкция вначале введена в редакторе, то обеспечивается возможность исправить опечатку и переопределить конструкцию с помощью нескольких нажатий клавиш непосредственно из редактора. Но первоначально в качестве демонстрируемых примеров рассматриваются правила, введенные непосредственно в систему CLIPS из приглашения режима верхнего уровня.

В качестве примера рассмотрим, какие типы фактов и правил могут оказаться полезными в той ситуации, когда приходится осуществлять текущий контроль и реагировать на несколько возможных чрезвычайных ситуаций. Одной из таких чрезвычайных ситуаций может оказаться пожар, а другой — наводнение. Ниже приведен псевдокод одного из возможных правил в экспертной системе текущего контроля над промышленной установкой.

```
IF the emergency is a fire
THEN the response is to activate
     the sprinkler system
```

Прежде чем преобразовать этот псевдокод в правило, необходимо определить конструкции deftemplate для фактов такого типа, которые упоминаются в этом правиле. Чрезвычайная ситуация может быть представлена с помощью следующей конструкции deftemplate:

```
(deftemplate emergency (slot type))
```

В этой конструкции поле type факта emergency должно содержать такие символы, как fire (пожар), flood (наводнение) и power outage (прекращение подачи электроэнергии). Аналогичным образом, ответ экспертной системы, response, может быть представлен следующей конструкцией deftemplate:

```
(deftemplate response (slot action))
```

В данной конструкции поле action факта response указывает на то, какой ответ должен быть выработан системой.

Правило, выраженное с помощью синтаксиса CLIPS, приведено ниже. Это правило можно ввести, набирая его текст после приглашения CLIPS, но прежде чем появится возможность ввести правило, необходимо ввести такие конструкции deftemplate, как emergency и response. Однако, в свою очередь, перед вводом любых из этих конструкций необходимо набрать в приглашении режима верхнего уровня команду (clear), а затем нажать клавишу ввода. В результате этого будут удалены все конструкции deftemplate и deffacts, созданные в результате выполнения примеров предыдущего раздела. Полное описание команды clear приведено в разделе 7.13.

```
(defrule fire-emergency "An example rule"
   (emergency (type fire))
   =>
   (assert (response

      (action activate-sprinkler-system))))
```

Если правило будет введено безошибочно, в полном соответствии с тем, что показано в данном примере, снова появится приглашение CLIPS. В противном случае будет выведено сообщение об ошибке, которое с наибольшей вероятностью будет указывать на неправильно введенное ключевое слово или на несогласованную круглую скобку.

Ниже приведено то же правило, в котором добавлены комментарии, соответствующие отдельным частям правила. Комментарии начинаются с точки с запятой и продолжаются до конца строки, обозначенного символом возврата каретки. Комментарии игнорируются системой CLIPS; дополнительная информация о комментариях приведена в разделе 7.18.

```
; Заголовок правила
(defrule fire-emergency "An example rule"
```

```
; Шаблоны
(emergency (type fire))
; Стрелка THEN
=>
; Действия
(assert (response
   (action activate-sprinkler-system))))
```

Определение правила имеет следующий общий формат:

```
(defrule <rule name> [<comment>]
   <patterns>* ; Левая часть (Left-Hand Side - LHS)
      правила
   =>
   <actions>*) ; Правая часть (Right-Hand Side - RHS)
      правила
```

Все правило должно быть заключено в круглые скобки; кроме того, в круглые скобки необходимо заключить каждый из компонентов `<patterns>` (шаблоны) и `<actions>` (действия). Правило может иметь несколько шаблонов и действий. Если круглые скобки, окружающие шаблоны и действия, являются вложенными, то левые и правые круглые скобки должны быть правильно сбалансированы. В правиле `fire-emergency` имеются один шаблон и одно действие.

Заголовок правила состоит из трех частей. Правило должно начинаться с **ключевого слова defrule**, за которым следует имя правила. В качестве имени можно применить любой допустимый символ CLIPS. Если правило вводится с именем правила, которое является таким же, как и у существующего правила, то новое правило заменяет старое. В данном правиле именем правила является `fire-emergency`. За именем следует необязательная строка комментария. Для этого правила комментарием служит `"An example rule"` (Пример правила). Комментарий обычно используется для описания назначения правила или представления любой другой информации, которую пожелает ввести программист. В отличие от обычных комментариев, начинающихся с точки с запятой, комментарий, который следует за именем правила, не игнорируется и может быть выведен на внешнее устройство вместе с остальной частью правила (с помощью команды `ppdefrule`, описанной в разделе 7.13).

За заголовком правила следует от нуля и больше условных элементов (Conditional Element — CE). Простейшим типом условного элемента является **условный элемент шаблона**, или просто **шаблон**. Каждый шаблон состоит из одного или нескольких ограничений, которые предназначены для сопоставления с полями факта, заданного с помощью конструкции `deftemplate`. В правиле `fire-emergency` шаблоном является `(emergency (type fire))`. Ограничение для поля `type` указывает на то, что это правило удовлетворяется только

для фактов `emergency`, содержащих в своем поле `type` символ `fire`. Система CLIPS предпринимает попытки сопоставить шаблоны правил с фактами в списке фактов. Если все шаблоны правила согласуются с фактами, правило **активизируется** и помещается в **рабочий список правил** — в коллекцию активизированных правил. В рабочем списке правил может находиться от нуля и больше правил.

Символ =>, который следует за шаблонами в правилах, называется **стрелкой**. Этот символ образуется путем нажатия клавиши со знаком равенства, а затем клавиши со знаком "больше". Стрелка — это символ, представляющий начало части `THEN` правила `IF-THEN`. Часть правила, находящаяся перед стрелкой, называется левой частью (**Left-Hand Side — LHS**), а часть, находящаяся за стрелкой, называется правой частью (**Right-Hand Side — RHS**).

Если правило не имеет шаблонов, то в качестве шаблона для данного правила добавляется специальный шаблон `(initial-fact)`. Поскольку конструкции `deffacts` с именем `initial-fact` определяются автоматически, все правила, не имеющие шаблонов в своих левых частях, активизируются после выполнения любой команды `reset`, поскольку при этом в список фактов автоматически вводится факт `(initial-fact)`. Таким образом, после выполнения команды `reset` в рабочий список правил помещаются все правила, не имеющие шаблонов в левой части.

Последней частью правила является список действий, выполняемых после **запуска** правила. Правило может не иметь действий. Такой способ использования правил не особенно полезен, но иногда применяется. В рассматриваемом примере одно из действий предусматривает внесение факта `(response (action activate-sprinkler-system))` в список фактов. Термин "запуск" означает, что система CLIPS выполняет действия одного из правил, находящихся в рабочем списке правил. Обычно программа при отсутствии правил в рабочем списке правил прекращает свое выполнение. Если же в рабочем списке правил имеется несколько правил, то система CLIPS автоматически определяет, какое из правил является подходящим для запуска. Система CLIPS упорядочивает правила, находящиеся в рабочем списке правил, с учетом возрастания приоритета и запускает правило с наивысшим приоритетом. Приоритет правила представляет собой целочисленный атрибут, который принято называть **значимостью** (salience). Понятие значимости будет описано более подробно в главе 9.

7.12 Рабочий список правил и выполнение программы

Программа CLIPS может быть вызвана на выполнение с помощью **команды run**. Команда `run` имеет следующий синтаксис:

```
(run [<limit>])
```

В этом определении необязательный параметр `<limit>` указывает максимальное количество правил, подлежащих запуску. Если параметр `<limit>` не задан или значение параметра `<limit>` равно `-1`, то запуск правил происходит до тех пор, пока в рабочем списке правил не останется ни одного правила. В ином случае выполнение правил прекращается после запуска такого количества правил, которое определено значением `<limit>`.

В ходе выполнения программы CLIPS из рабочего списка правил происходит запуск правила, имеющего самую высокую значимость. Безусловно, если в рабочем списке правил имеется только одно правило, то запускается именно это правило. В данном случае условный элемент правила `fire-emergency` удовлетворяется под действием факта `(emergency (type fire))`, поэтому после вызова программы на выполнение должен произойти запуск правила `fire-emergency`.

Правило активизируется после того, как все шаблоны правила согласовываются с фактами. Процесс сопоставления с шаблонами всегда поддерживается в актуальном состоянии и происходит без учета того, произошел ли ввод фактов в список фактов до или после определения того или иного правила.

Безусловно, для вызова правил на выполнение требуются факты, поэтому основным методом запуска или перезапуска экспертной системы, написанной на языке CLIPS, является команда `reset`. Как правило, факты, введенные в список фактов с помощью команды `reset`, удовлетворяют шаблонам одного или нескольких правил, что приводит к передаче результатов активизации этих правил в рабочий список правил. Затем, после ввода команды `run`, начинается выполнение программы.

Отображение рабочего списка правил

Список правил, находящихся в рабочем списке правил, можно вывести на внешнее устройство с помощью **команды agenda**. Синтаксис команды `agenda` является таковым:

```
(agenda)
```

Если в рабочем списке правил активизированные правила отсутствуют, то после ввода команды `agenda` снова появляется приглашение CLIPS. А если правило `fire-emergency` было активизировано фактом `(emergency (type fire))` с индексом факта `1`, то выполнение команды `agenda` приводит к получению следующего вывода:

```
CLIPS> (reset)↵
CLIPS> (assert (emergency (type fire)))↵
<Fact-1>
CLIPS> (agenda)↵
```

```
0      fire-emergency: f-1
For a total of 1 activation.
CLIPS>
```

В этом выводе число 0 показывает значимость данного правила в рабочем списке правил. За обозначением значимости следует имя правила, после чего отображаются идентификаторы фактов, которые согласуются с шаблонами правила. В данном случае имеется только один идентификатор факта, `f-1`.

Правила и релаксация правил

Поскольку в данном случае в рабочем списке правил находится правило `fire-emergency`, выполнение команды `run` приведет к запуску этого правила. В качестве результата выполнения действия этого правила к списку фактов будет добавлен факт `(response (action activate-sprinkler-system))`, как показано в следующем выводе:

```
CLIPS> (run)↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (emergency (type fire))
f-2     (response
           (action activate-sprinkler-system))
For a total of 3 facts.
CLIPS>
```

Теперь возникает интересный вопрос: “Что произойдет, если снова будет вызвана команда `run`?” Напрашивается ответ, что имеется правило и налицо факт, который удовлетворяет этому правилу, поэтому должен снова произойти запуск правила. Но если сейчас будет предпринята попытка выполнить команду `run`, это не приведет к получению каких-либо результатов. Проверка рабочего списка правил покажет, что запуск правил не произошел, поскольку в рабочем списке правил отсутствовали правила.

Запуск правила не происходит снова потому, что именно так спроектирована система CLIPS. Правила в системе CLIPS демонстрируют свойство, называемое **релаксацией**; под этим подразумевается, что запуск правил применительно к какому-то конкретному множеству фактов не происходит больше одного раза. А если бы свойство релаксации не было предусмотрено, то экспертные системы всегда попадали бы в тривиальные циклы. Под этим подразумевается то, что после запуска какого-то правила этот запуск происходил бы применительно к одному и тому же факту снова и снова, тогда как в реальном мире стимул, вызвавший запуск рефлекса, в конечном итоге исчезает. Например, огонь был бы наконец потушен с помощью спринклерной системы, или пожар, все уничтожив, закончился бы сам

собой. Но в компьютерном мире после ввода факта в список фактов он остается в списке до тех пор, пока не будет удален явно.

В случае необходимости можно обеспечить повторный запуск правила путем извлечения факта `(emergency (type fire))` и повторного его ввода. По существу, система CLIPS запоминает идентификаторы фактов, вызвавших запуск правила, и не активизирует снова это правило с помощью точно повторяющейся комбинации идентификаторов фактов. Идентичные множества идентификаторов фактов должны совпадать взаимно однозначно, как по порядку расположения, так и по индексам фактов. Ниже приведены примеры, которые показывают, как может оказаться так, что один и тот же факт согласуется с шаблоном больше чем одним способом. В таком случае в рабочий список правил может быть помещено несколько активизированных правил с одним и тем же множеством идентификаторов фактов, относящихся к одному правилу, причем каждое из этих активизированных правил соответствует каждому отдельному согласованию.

Еще один вариант состоит в том, что для повторного запуска правила можно применить **команду refresh**. Команда `refresh` помещает в рабочий список правил все активизированные правила, соответствующие некоторому правилу, запуск которых уже произошел (с учетом того условия, что факты, вызвавшие активизацию, все еще присутствуют в списке фактов). Команда `refresh` имеет следующий синтаксис:

```
(refresh <rule-name>)
```

Ниже приведены команды, которые показывают, как можно воспользоваться командой `refresh` для повторной активизации правила `fire-emergency`.

```
CLIPS> (agenda)↵
CLIPS> (refresh fire-emergency)↵
CLIPS> (agenda)↵
0      fire-emergency: f-1
For a total of 1 activation.
CLIPS>
```

Отслеживание активизаций, правил и статистических данных

Если осуществляется отслеживание активизаций, то система CLIPS автоматически выводит сообщение при каждом добавлении или удалении активизированного правила из рабочего списка правил. Как и в случае фактов, последовательность знаков <== указывает, что активизированное правило удаляется из рабочего списка правил, а последовательность знаков ==> указывает, что активизированное правило добавляется к рабочему списку правил. Вслед за этой начальной последовательностью знаков отображается активизированное правило в том же формате,

который используется в команде agenda. Ниже приведена последовательность команд, которая показывает, как происходит отслеживание активизаций.

```
CLIPS> (reset)↵
CLIPS> (watch activations)↵
CLIPS> (assert (emergency (type fire)))↵
==> Activation 0       fire-emergency: f-1
<Fact-1>
CLIPS> (agenda)↵
0      fire-emergency: f-1
For a total of 1 activation.
CLIPS> (retract 1)↵
<== Activation 0       fire-emergency: f-1
CLIPS> (agenda)↵
CLIPS>
```

Если же осуществляется отслеживание правил, то система CLIPS выводит сообщение при запуске каждого правила. Приведенная ниже последовательность команд показывает, как происходит отслеживание активизаций и правил.

```
CLIPS> (reset)↵
CLIPS> (watch rules)↵
CLIPS> (assert (emergency (type fire)))↵
==> Activation 0        fire-emergency: f-1
<Fact-1>
CLIPS> (run)↵
FIRE 1                  fire-emergency: f-1
CLIPS> (agenda)↵
CLIPS>
```

Число, следующее за символом FIRE, показывает, сколько правил было запущено со времени выдачи команды run. Например, если бы после правила fire-emergency было запущено другое правило, то ему бы предшествовало обозначение “FIRE 2”. После вывода порядкового номера запуска выводится имя правила, а затем отображаются индексы фактов, согласованных с шаблонами этого правила. Следует отметить, что отслеживание активизации не вызывает отображения сообщений при запуске правил (и удалении в связи с этим из рабочего списка правил).

Существует возможность отслеживать только конкретные правила и активизации, задавая одно или несколько имен правил в конце команды watch. Например, после выдачи команды (watch activations fire-emergency) отображаются только сообщения об активизации, касающиеся правила fire-emergency.

Если осуществляется отслеживание статистических данных (с помощью параметра statistics), то система CLIPS в конце каждого прогона программы,

выполняемого после выдачи команды run, выводит информационные сообщения, подобные приведенным ниже.

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch statistics)↵
CLIPS> (assert (emergency (type fire)))↵
<Fact-1>
CLIPS> (run)↵
1 rules fired     Run time is 0.02 seconds
50.0 rules per second
3 mean number of facts (3 maximum)
1 mean number of instances (1 maximum)
1 mean number of activations (1 maximum)
CLIPS> (unwatch statistics)↵
CLIPS>
```

Таким образом, если происходит отслеживание статистики, то после выполнения команды run выводятся шесть наборов статистических данных. Вначале отображается общее количество активизированных правил, количество времени в секундах, затраченного на запуск правил, и среднее количество правил, запускаемых в секунду (результат деления первого статистического показателя на второй). Кроме того, для каждого цикла исполнения программы система CLIPS сохраняет статистические данные о количестве фактов, активизаций и экземпляров. Среднее количество фактов рассчитывается как сумма общего количества фактов в списке фактов после запуска каждого правила, деленное на количество запущенных правил. Число в круглых скобках, за которым следует слово maximum, показывает наибольшее количество фактов, содержавшихся в списке фактов, относящееся к запуску любого отдельного правила. Аналогичным образом, количества, обозначенные в статистических данных об активизациях как mean и maximum, показывают среднее количество активизаций в расчете на запуск каждого правила и наибольшее количество активизированных правил в рабочем списке правил, относящееся к запуску любого отдельного правила. Значения mean и maximum, относящиеся к экземплярам, показывают информацию о конструкциях языка COOL.

7.13 Команды, применяемые для манипулирования конструкциями

Отображение списка элементов указанной конструкции

Для отображения текущего списка правил, сопровождаемых системой CLIPS, используется **команда list-defrules**. Аналогичным образом **команды list-deftem-**

plates и **list-deffacts** могут применяться соответственно для отображения текущего списка конструкций `deftemplate` или текущего списка конструкций `deffacts`. Эти команды имеют следующий синтаксис:

```
(list-defrules)
(list-deftemplates)
(list-deffacts)
```

Ниже приведен пример применения этих команд.

```
CLIPS> (list-defrules)↵
fire-emergency
For a total of 1 rule.
CLIPS> (list-deftemplates)↵
initial-fact
emergency
response
For a total of 3 deftemplates.
CLIPS> (list-deffacts)↵
initial-fact
For a total of 1 deffacts.
CLIPS>
```

Отображение текстового представления указанного элемента конструкции

Для отображения текстовых представлений конструкций `defrule`, `deftemplate` и `deffacts` применяются соответственно **команда ppdefrule** (pretty print `defrule`), **команда ppdeftemplate** (pretty print `deftemplate`) и **команда ppdeffacts** (pretty print `deffacts`), которые имеют следующий синтаксис:

```
(ppdefrule <defrule-name>)
(ppdeftemplate <deftemplate-name>)
(ppdeffacts <deffacts-name>)
```

Единственный параметр каждой команды задает имя отображаемой конструкции `defrule`, `deftemplate` или `deffacts`. Во время вывода этой информации система CLIPS разбивает различные части конструкций на строки для удобства чтения, например, как показано ниже.

```
CLIPS> (ppdefrule fire-emergency)↵
(defrule MAIN::fire-emergency "An example rule"
   (emergency (type fire))
   =>
   (assert (response
```

```
              (action activate-sprinkler-system))))
CLIPS> (ppdeftemplate response)↵
(deftemplate MAIN::response
   (slot action))
CLIPS> (ppdeffacts initial-fact)↵
CLIPS> (deffacts start-fact (start-fact))↵
CLIPS> (ppdeffacts start-fact)↵
(deffacts MAIN::start-fact
   (start-fact))
CLIPS>
```

Символ `MAIN::`, предшествующий каждому имени конструкции, обозначает модуль, в который помещена эта конструкция. Модули предоставляют механизм секционирования базы знаний и рассматриваются более подробно в главе 9. Обратите внимание на то, что конструкция `deffacts` с именем `initial-fact` не имеет текстового представления (поскольку создается системой CLIPS автоматически). С другой стороны, конструкция `deffacts` с именем `start-fact`, введенная в программу, имеет текстовое представление.

Удаление указанного элемента конструкции

Для удаления конструкции `defrule`, `deftemplate` или `deffacts` используются соответственно **команда undefrule**, **команда undeftemplate** и **команда undeffacts**, которые имеют следующий синтаксис:

```
(undefrule <defrule-name>)
(undeftemplate <deftemplate-name>)
(undeffacts <deffacts-name>)
```

Каждая команда принимает единственный параметр, который задает имя конструкции `defrule`, `deftemplate` или `deffacts`, подлежащей удалению, например, как показано ниже.

```
CLIPS> (undeffacts start-fact)↵
CLIPS> (list-deffacts)↵
initial-fact
For a total of 1 deffacts.
CLIPS> (undefrule fire-emergency)↵
CLIPS> (list-defrules)↵
CLIPS>
```

Следует отметить, что конструкция `deffacts` с именем `initial-facts` и конструкция `deftemplate` с именем `initial-facts` могут быть удалены наряду с любыми другими конструкциями, определяемыми пользователем. А если

после этого будет выполнена команда `reset`, то к списку фактов не добавится факт `(initial-fact)`.

Если в качестве параметра любой из команд удаления конструкций указывается символ `*`, то удаляются все конструкции соответствующего типа. Например, команда `(undefrule *)` удаляет все конструкции `defrule`. Символ `*` может также использоваться в команде `retract` для удаления всех фактов.

Конструкции, на которые имеется ссылка в других конструкциях, не могут быть удалены до тех пор, пока не произошло удаление ссылающихся на них конструкций. Как показывает приведенный ниже диалог, конструкция `deftemplate` с именем `initial-fact` не может быть удалена до тех пор, пока не удалены конструкция `deffacts` с именем `initial-fact`, факт `(initial-fact)` и конструкция `defrule` с именем `example` (в которой используется применяемый по умолчанию шаблон `initial-fact`).

```
CLIPS> (defrule example =>)↵
CLIPS> (undeftemplate initial-fact)↵
[PRNTUTIL4] Unable to delete deftemplate
initial-fact.
CLIPS> (undeffacts initial-fact)↵
CLIPS> (undeftemplate initial-fact)↵
[PRNTUTIL4] Unable to delete deftemplate
initial-fact.
CLIPS> (undefrule example)↵
CLIPS> (undeftemplate initial-fact)↵
[PRNTUTIL4] Unable to delete deftemplate
initial-fact.
CLIPS> (retract *)↵
CLIPS> (undeftemplate initial-fact)↵
CLIPS>
```

Удаление всех конструкций из среды CLIPS (очистка среды)

Для удаления всей информации, содержащейся в среде CLIPS, может применяться **команда clear**. Эта команда удаляет все конструкции, содержащиеся в настоящее время в системе CLIPS, а также все факты из списка фактов. Синтаксис команды `clear` такой:

```
(clear)
```

После очистки среды CLIPS команда `clear` добавляет конструкцию `deffacts` с именем `initial-fact` к среде CLIPS, как показано ниже.

```
CLIPS> (list-deffacts)↵
CLIPS> (list-deftemplates)↵
emergency
response
start-fact
For a total of 3 deftemplates.
CLIPS> (clear)↵
CLIPS> (list-deffacts)↵
initial-fact
For a total of 1 deffacts.
CLIPS> (list-deftemplates)↵
initial-fact
For a total of 1 deftemplate.
CLIPS>
```

7.14 Команда printout

Правая часть правил может использоваться не только для ввода фактов в список фактов, но и для вывода информации. Для этого служит **команда printout**, которая имеет следующий синтаксис:

```
(printout <logical-name> <print-items>*)
```

В этом определении параметр <logical-name> показывает назначение вывода команды printout, а параметр <print-items>* содержит от нуля и больше элементов, которые должны быть выведены с помощью указанной команды.

Применение команды printout демонстрируется на примере следующего правила:

```
(defrule fire-emergency
(emergency (type fire))
=>
(printout t "Activate the sprinkler system" crlf))
```

Очень важно задать букву t после имени команды printout, поскольку этот параметр показывает назначение вывода. Такое назначение принято также называть **логическим именем устройства**. В данном случае логическое имя t сообщает системе CLIPS, что вывод должен быть направлен на **стандартное устройство вывода** компьютера (обычно таковым является терминал). Это логическое имя может быть переопределено таким образом, чтобы стандартным устройством вывода стало нечто иное, допустим, модем или принтер. Понятие логических имен будет подробно рассматриваться в разделе 8.6.

Параметры, которые следуют за логическим именем, представляют собой элементы данных, подлежащих выводу с помощью команды `printout`. В данном случае на терминал должна быть выведена следующая строка без окружающих ее кавычек:

```
"Activate the sprinkler system"
```

В команде `printout` **ключевое слово crlf** трактуется особым образом. Применение этого ключевого слова приводит к выводу знаков возврата каретки и перевода строки, благодаря чему внешний вид выходных данных улучшается в результате применения форматирования с разбивкой на строки.

7.15 Использование нескольких правил

До сих пор в этой главе рассматривались исключительно программы простейшего типа, состоящие лишь из одного правила. Но экспертная система, которая включает только одно правило, вряд ли может принести значительную пользу. Практически применимые экспертные системы могут состоять из сотен или тысяч правил. В частности, экспертная система текущего контроля за промышленной установкой кроме правила `fire-emergency` может включать правило для таких критических ситуаций, в которых произошло наводнение. Итак, дополним множество правил, которое теперь должно выглядеть следующим образом:

```
(defrule fire-emergency
   (emergency (type fire))
   =>
   (printout t "Activate the sprinkler system"
   crlf))
(defrule flood-emergency
   (emergency (type flood))
   =>
   (printout t "Shut down electrical equipment"
               crlf))
```

После ввода этих правил в систему CLIPS введение факта `(emergency (type fire))`, а затем выдача команды `run` приводят к формированию вывода `"Activate the sprinkler system"` (Активизировать спринклерную систему). А ввод факта `(emergency (type flood))` и выдача команды `run` приводят к получению вывода `"Shut down the electrical equipment"` (Отключить электрическое оборудование).

Отражение в правиле свойств реального мира

Если бы приходилось иметь дело только с такими двумя чрезвычайными ситуациями, как пожары и наводнения, то приведенных выше правил было бы достаточно. Но реальный мир не так прост. Например, не все пожары могут быть потушены с помощью воды. Для тушения некоторых пожаров требуются химические огнетушители. Кроме того, необходимо предусмотреть действия, осуществляемые в том случае, если при пожаре, допустим, выделяются ядовитые газы или происходят взрывы. Следует ли известить внешнее пожарное отделение, а не только собственное подразделение пожарной охраны? Имеет ли значение то, на каком этаже здания происходит пожар? Ведь ввод в действие водяной спринклерной системы на втором этаже может вызвать повреждения под действием воды и на втором, и на первом этажах. К тому же может потребоваться отключить подачу электропитания к оборудованию на обоих этажах. А пожар, происходящий на первом этаже, может потребовать отключения электропитания, поступающего к оборудованию, только на первом этаже. С другой стороны, если здание подвергается воздействию наводнения, то имеются ли в нем водонепроницаемые двери, которые можно закрыть, чтобы предотвратить ущерб? А какие действия необходимо предпринять в случае обнаружения незаконного вторжения в помещение промышленной установки? Наконец, как проверить, все ли варианты были учтены, после того как эти ситуации найдут свое отражение в правилах?

К сожалению, ответ на этот вопрос всегда отрицателен. В реальном мире события очень редко развиваются в соответствии с заранее продуманным сценарием. К тому же задача отражения в экспертной системе всех относящихся к делу знаний может оказаться весьма трудной. В наилучшем случае существует возможность определить лишь наиболее важные чрезвычайные ситуации и предусмотреть правила, позволяющие экспертной системе распознать такое положение, когда она не способна справиться с критической ситуацией.

Правила с несколькими шаблонами

Большинство эвристик, применяемых на практике, являются слишком сложными для того, чтобы их можно было выразить с помощью правил, имеющих только один шаблон. Например, решение активизировать спринклерную систему при обнаружении пожара любого типа может оказаться не только ошибочным, но и опасным. Пожары, в которых происходит горение таких обычных сгораемых материалов, как бумага, древесина и ткань (пожары категории A), могут быть потушены с помощью воды или огнетушителей на основе воды. А пожары, которые охватывают легко воспламеняемые и горючие жидкости, консистентные смазки и подобные материалы (пожары категории B) необходимо гасить другим способом, например, с помощью углекислотного огнетушителя. Для представле-

ния таких условий могут применяться правила больше чем с одним шаблоном, например, как показано ниже.

```
(deftemplate extinguisher-system
   (slot type)
   (slot status))
(defrule class-A-fire-emergency
   (emergency (type class-A-fire))
   (extinguisher-system (type water-sprinkler)
                        (status off))
   =>
   (printout t "Activate water sprinkler" crlf))
(defrule class-B-fire-emergency
   (emergency (type class-B-fire))
   (extinguisher-system (type carbon-dioxide)
                        (status off))
   =>
   (printout t "Use carbon dioxide extinguisher"
               crlf))
```

Оба этих правила имеют два шаблона. Первый шаблон определяет, существует ли критическая ситуация, связанная с пожаром, и относится ли пожар к категории A или к категории B. Второй шаблон определяет, включена ли соответствующая система пожаротушения. Для решения задачи отключения средств пожаротушения могут быть введены дополнительные правила (например, если уже включена водяная спринклерная система, но возник пожар категории B, то систему необходимо отключить; еще один вариант состоит в том, что все средства пожаротушения должны быть отключены после того, как пожар будет ликвидирован). А для определения того, приводит ли воспламенение какого-то конкретного горючего материала к развитию пожара категории A или категории B, могут быть введены другие дополнительные правила.

В любое правило может быть помещено любое количество шаблонов. Но следует учитывать важное условие, что правило помещается в рабочий список правил, только если имеющиеся факты удовлетворяют всем шаблонам. Ограничение этого типа принято называть **условным элементом and**. В условном элементе `and` неявно содержатся шаблоны всех правил, поэтому правило не активизируется, если факты удовлетворяют только одному из шаблонов. В системе должны появиться все необходимые факты, и только после этого удовлетворяется левая часть какого-то правила, которое помещается в рабочий список правил.

7.16 Команда set-break

В языке CLIPS предусмотрена команда отладки, называемая **командой set-break**, которая позволяет останавливать исполнение программы перед тем, как произойдет запуск любого правила из указанной группы правил. Правило, которое останавливает исполнение прежде, чем будет запущено, называется **точкой останова**. Команда set-break имеет следующий синтаксис:

```
(set-break <rule-name>)
```

В этом определении параметр <rule-name> представляет собой имя правила, для которого задается точка останова. В качестве примера рассмотрим следующие правила (обратите внимание на то, что в данном случае используются упорядоченные факты, как описано в разделе 7.6):

```
(defrule first
   =>
   (assert (fire second)))
(defrule second
   (fire second)
   =>
   (assert (fire third)))
(defrule third
   (fire third)
   =>)
```

Следующий пример диалогового выполнения команд показывает, как исполняются правила, для которых не заданы какие-либо точки останова:

```
CLIPS> (watch rules)↵
CLIPS> (reset)↵
CLIPS> (run)↵
FIRE 1 first: f-0
FIRE 2 second: f-1
FIRE 3 third: f-2
CLIPS>
```

В данном случае после выдачи команды run последовательно происходит запуск всех трех правил. А в следующем примере диалогового выполнения команд демонстрируется использование команды set-break для останова выполнения:

```
CLIPS> (set-break second)↵
CLIPS> (set-break third)↵
CLIPS> (reset)↵
CLIPS> (run)↵
FIRE 1 first: f-0
```

```
Breaking on rule second
CLIPS> (run)↵
FIRE 1 second: f-1
Breaking on rule third
CLIPS> (run)↵
FIRE 1 third: f-2
CLIPS>
```

В данном случае выполнение команд останавливается прежде, чем будет разрешен запуск правил `second` и `third`. Обратите внимание на то, что после выдачи команды `run` должно быть запущено по меньшей мере одно правило, и только после этого прекращается выполнение в точке останова. Например, после останова выполнения правила `second` повторная выдача команды `run` больше не приводит к останову выполнения.

Для вывода списка всех точек останова может применяться **команда show-breaks**, которая имеет следующий синтаксис:

```
(show-breaks)
```

Для удаления точек останова может использоваться **команда remove-break**. Эта команда имеет такой синтаксис:

```
(remove-break [<rule-name>])
```

Если в качестве параметра задано имя правила `<rule-name>`, то удаляется только точка останова, соответствующая этому правилу. В противном случае удаляются все точки останова.

7.17 Загрузка и сохранение конструкций

Загрузка конструкций из файла

В систему CLIPS с помощью **команды load** может быть загружен файл с конструкциями, подготовленный с использованием текстового редактора. Команда `load` имеет следующий синтаксис:

```
(load <file-name>)
```

В этом определении параметр `<file-name>` представляет собой строку или символ, который содержит имя файла, подлежащего загрузке.

При условии, что правила и конструкции `deftemplate`, описывающие чрезвычайную ситуацию, хранятся в файле “`fire.clp`” на гибком диске персонального компьютера IBM, следующая команда загрузит эти конструкции в систему CLIPS:

```
(load "B:fire.clp")
```

Безусловно, формат спецификации имени файла зависит от используемой операционной системы, поэтому данный пример должен рассматриваться только в качестве общего руководства. В процессе загрузки может возникнуть проблема, связанная с тем, что в некоторых операционных системах в качестве разделителя компонентов имени пути к каталогу используется обратная косая черта. А поскольку в системе CLIPS обратная косая черта интерпретируется как знак переключения на другой режим обработки, то для представления в строке единственной обратной косой черты необходимо использовать две обратные косые черты. Например, обычно имя пути записывается примерно таким образом:

```
B:\usr\clips\fire.clp
```

Но, чтобы сохранить знаки обратной косой черты, это имя пути необходимо записать, как показано в следующем примере команды:

```
(load "B:\\usr\\clips\\fire.clp")
```

Не все конструкции должны обязательно храниться в единственном файле. Все конструкции программы можно распределить по нескольким файлам и загружать с использованием ряда команд `load`. Если во время загрузки файла ошибки не возникают, то команда `load` возвращает символ `TRUE` (подробные сведения о возвращаемых значениях приведены в главе 8). В противном случае эта команда возвращает символ `FALSE`.

Отслеживание операций компиляции

Если предусмотрено отслеживание операций компиляции (применяемое по умолчанию), то команда `load` после загрузки каждой конструкции выводит информационное сообщение, в котором содержится имя конструкции. Например, предположим, что непосредственно перед этим произошел запуск системы CLIPS и осуществляется ввод следующих команд:

```
CLIPS> (load "fire.clp")↵
Defining deftemplate: emergency
Defining deftemplate: response
Defining defrule: fire-emergency +j
TRUE
CLIPS>
```

Приведенные здесь сообщения показывают, что загружены две конструкции `deftemplate` (`emergency` и `response`), а вслед за ними загружено правило `fire-emergency`. Строка “`+j`”, находящаяся в конце сообщения “`Defining defrule`” (Определение конструкции `defrule`), представляет собой поступающую из системы CLIPS информацию о внутренней структуре компилируемых правил. Эта информация может оказаться полезной при настройке программы

и будет рассматриваться более подробно в главе 9, в которой речь идет об эффективности.

Если же отслеживание операций компиляции не предусмотрено, то система CLIPS выводит по одному знаку в расчете на каждую загруженную конструкцию: * обозначает конструкции defrules; % — конструкции deftemplate; $ — конструкции deffacts, например, как показано ниже.

```
CLIPS> (clear)↵
CLIPS> (unwatch compilations)↵
CLIPS> (load fire.clp)↵
%%*
TRUE
CLIPS>
```

Сохранение конструкций в файле

В языке CLIPS предусмотрена также команда, противоположная по своему назначению команде load. **Команда save** предоставляет возможность сохранить в файле на диске множество конструкций, хранящихся в системе CLIPS. Синтаксис команды save приведен ниже.

```
(save <file-name>)
```

Например, команда, которая сохраняет конструкции правил борьбы с огнем в файле fire.clp на гибком диске B, имеет такой вид:

```
(save "B:fire.clp")
```

Команда save сохраняет в указанном файле все конструкции, заданные в системе CLIPS. Возможность сохранить в файле только указанные конструкции отсутствует. Обычно если для создания и модификации конструкций используется редактор, то нет необходимости применять команду save, поскольку конструкции сохраняются в ходе работы с редактором. Но иногда бывает удобно вводить конструкции непосредственно в приглашении CLIPS, а затем сохранять их в файле.

7.18 Комментирование конструкций

В программу CLIPS рекомендуется включать комментарии. Иногда при попытке понять назначение конструкций возникают сложности, поэтому для предоставления читателю пояснений, касающихся назначения конструкций, могут применяться комментарии. Кроме того, комментарии используются для качественного документирования программ и являются полезными при изучении длинных программ.

Комментарием в языке CLIPS является любой текст, который начинается с точки с запятой и заканчивается знаком возврата каретки. Ниже приведен пример применения комментариев в программе `fire`.

```
;***********************************
;*                                 *
;*                                 *
;* Programmer: G. D. Riley         *
;*                                 *
;* Title: The Fire Program         *
;*                                 *
;* Date: 05/17/04                  *
;*                                 *
;***********************************
; Конструкции deftemplate
(deftemplate emergency "template #1"
   (slot type)) ; Тип чрезвычайной ситуации
(deftemplate response "template #2"
   (slot type)) ; Способ реагирования на чрезвычайную
                ; ситуацию
; Назначение этого правила состоит в активизации
;спринклерной системы, если возник пожар
(defrule fire-emergency "An example rule" ; IF
; Чрезвычайная ситуация, связанная с возникновением пожара
   (emergency (type fire))
=>                                          ; Стрелка THEN
; Активизировать спринклерную систему
(assert (response
   (action activate-sprinkler-system))))
```

Загрузка этих конструкций в систему CLIPS и последующий структурированный вывод показывает, что в программе CLIPS уничтожаются все комментарии, начинающиеся с точки с запятой. Единственным сохранившимся комментарием является тот, который задан в кавычках после имени конструкции.

7.19 Переменные

В языке CLIPS, как и в других языках программирования, для хранения значений можно воспользоваться **переменными**. Переменные в языке CLIPS всегда записываются с применением синтаксиса, предусматривающего использование вопросительного знака, за которым следует имя символического поля. Имена переменных должны соответствовать требованиям к синтаксису символов, не считая

того исключения, что они должны начинаться с буквы. Требования к хорошему стилю программирования диктуют, чтобы переменным присваивались осмысленные имена. Некоторые примеры переменных приведены ниже.

```
?speed
?sensor
?value
?noun
?color
```

Между вопросительным знаком и именем символического поля не должно быть пробелов. Как будет описано позже, сам вопросительный знак имеет собственное назначение. Переменные используются в левой части правила для хранения значений слотов, которые в дальнейшем могут сравниваться с другими значениями в левой части правила или применяться в правой части правила. Для описания операции присваивания значения переменной и полученного при этом результата используются термины **связывать** и **связанный**.

Один из общепринятых способов использования переменных состоит в том, что с переменной в левой части правила связывается значение, а затем это значение применяется в правой части правила, как показано ниже.

```
CLIPS> (clear)↵
CLIPS>
(deftemplate person
   (slot name)
   (slot eyes)
   (slot hair))↵
CLIPS>
(defrule find-blue-eyes
   (person (name ?name) (eyes blue))
   =>
   (printout t ?name " has blue eyes." crlf))↵
CLIPS>
(deffacts people
   (person (name Jane)
           (eyes blue) (hair red))
   (person (name Joe)
           (eyes green) (hair brown))
   (person (name Jack)
           (eyes blue) (hair black))
   (person (name Jeff)
           (eyes green) (hair brown)))↵
CLIPS> (reset)↵
```

```
CLIPS> (run)↵
Jack has blue eyes.
Jane has blue eyes.
CLIPS>
```

В данном случае и Джейн, и Джек имеют синие глаза, поэтому правило `find-blue-eyes` активизируется дважды, по одному разу для каждого из фактов, описывающих Джейн и Джека. После запуска правила слот `name` проверяется на наличие факта, активизировавшего запускаемое в текущий момент правило, после чего это значение используется в операторе `printout`.

Если в правой части правила применяется ссылка на переменную, а в левой части этого правила не происходит связывание переменной, то система CLIPS выводит следующее сообщение об ошибке (предполагается, что несвязанной была переменная `?x`):

```
[PRCCODE3] Undefined variable x referenced in
           RHS of defrule.
```

7.20 Многократное использование переменных

Переменные с одним и тем же именем, которые используются в нескольких местах левой части правила, обладают одним важным и полезным свойством. После того как переменная будет впервые связана со значением, это значение остается в правиле неизменным. Другие вхождения одной и той же переменной должны согласовываться с тем же значением, которое было впервые присвоено переменной.

Поэтому вместо записи отдельного правила, в котором выполняется только поиск людей с синими глазами, можно вставить в список фактов такой факт, который указывает конкретный искомый цвет глаз, например, как показано ниже.

```
CLIPS> (undefrule *)↵
CLIPS> (deftemplate find (slot eyes))↵
CLIPS>
(defrule find-eyes
   (find (eyes ?eyes))
   (person (name ?name) (eyes ?eyes))
   =>
   (printout t ?name " has " ?eyes " eyes."
      crlf))↵
CLIPS>
```

Конструкция `deftemplate` с именем `find` указывает цвет глаз; это значение задается с помощью слота `eyes`. В таком случае с использованием правила

find-eyes может осуществляться выборка значения из слота eyes конструкции deftemplate с именем find, а затем поиск всех фактов person, для которых значение слота eyes является таким же, какое было связано с переменной ?eyes. В следующем диалоге показано, как применяется этот прием:

```
CLIPS> (reset)↵
CLIPS> (assert (find (eyes blue)))↵
<Fact-5>
CLIPS> (run)↵
Jack has blue eyes.
Jane has blue eyes.
CLIPS> (assert (find (eyes green)))↵
<Fact-6>
CLIPS> (run)↵
Jeff has green eyes.
Joe has green eyes.
CLIPS> (assert (find (eyes purple)))↵
<Fact-7>
CLIPS> (run)
CLIPS>
```

Обратите внимание на то, что после ввода факта (find (eyes purple)) не происходит запуск каких-либо правил, поскольку отсутствуют факты person со значением purple в слоте eyes.

7.21 Адреса фактов

Операции извлечения, модификации и дублирования фактов являются широко применяемыми и обычно осуществляются в правой части правила, а не на верхнем уровне. В данной главе уже было показано, как выполняются эти операции с фактами в приглашении верхнего уровня с использованием индексов фактов. Но прежде чем появится возможность выполнять манипуляции с фактом из правой части правила, необходимо предусмотреть определенный способ задания факта, который согласуется с конкретным шаблоном. Для достижения этой цели можно связать некоторую переменную с **адресом факта**, относящимся к тому факту, который согласуется с шаблоном левой части правила с помощью операции **связывания с шаблоном**, <-. После связывания эту переменную можно использовать в командах retract, modify или duplicate вместо индекса факта. Например, следующий диалог показывает, как обновить значения слота конструкции deftemplate из правой части правила:

```
CLIPS> (clear)↵
CLIPS>
```

```
(deftemplate person
   (slot name)
   (slot address))↵
CLIPS>
(deftemplate moved
   (slot name)
   (slot address))↵
CLIPS>
(defrule process-moved-information
   ?f1 <- (moved (name ?name)
          (address ?address))
   ?f2 <- (person (name ?name))
   =>
   (retract ?f1)
   (modify ?f2 (address ?address)))↵
CLIPS>
(deffacts example
(person (name "John Hill")
        (address "25 Mulberry Lane"))
(moved  (name "John Hill")
        (address "37 Cherry Lane")))↵
CLIPS> (reset)↵
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (run)↵
FIRE 1 process-moved-information: f-2,f-1
<== f-2 (moved (name "John Hill")
               (address "37 Cherry Lane"))
<== f-1 (person (name "John Hill")
               (address "25 Mulberry Lane"))
==> f-3 (person (name "John Hill")
               (address "37 Cherry Lane"))
CLIPS>
```

В данном примере используются две конструкции `deftemplate`. Конструкция `deftemplate` с именем `person` применяется для хранения информации о некотором лице, в этом случае только об имени и адресе этого лица. Существует также возможность хранить другую информацию, такую как возраст или цвет глаз. А конструкция `deftemplate` с именем `moved` используется для указания на то, что адрес лица изменился. Новый адрес задается в слоте `address`.

Первый шаблон в правиле process-moved-information определяет, требуется ли выполнить обработку, связанную со сменой адреса, а второй шаблон применяется для поиска факта person, для которого потребуется изменить информацию об адресе. Адрес факта, относящийся к факту moved, связывается с переменной ?f1 таким образом, что этот факт можно легко извлечь, приступая к осуществлению изменения. Адрес факта, относящийся к факту person, связывается с переменной ?f2, которая в дальнейшем используется в правой части правила для модификации значения слота address.

Обратите внимание на то, что могут одновременно использоваться переменные, связанные со значением в факте, и переменные, связанные с адресом факта, относящимся к некоторому факту. Кроме того, значение переменной ?address может все еще применяться в правой части правила даже для извлечения факта, из которого было получено указанное значение.

Для того чтобы правило действовало должным образом, важно предусмотреть извлечение факта moved из правой части правила process-moved-information. Обратите внимание на то, что происходит в следующем примере после удаления команды retract:

```
CLIPS>
(defrule process-moved-information
   (moved (name ?name) (address ?address))
   ?f2 <- (person (name ?name))
   =>
   (modify ?f2 (address ?address)))↵
CLIPS> (unwatch facts)↵
CLIPS> (unwatch rules)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (watch activations)↵
CLIPS> (run 3)↵
FIRE     1 process-moved-information: f-2,f-1
<== f-1     (person (name "John Hill")
                    (address "25 Mulberry Lane"))
==> f-3     (person (name "John Hill")
                    (address "37 Cherry Lane"))
==> Activation 0 process-moved-information: f-2,f-3
FIRE     2 process-moved-information: f-2,f-3
<== f-3     (person (name "John Hill")
                    (address "37 Cherry Lane"))
==> f-4     (person (name "John Hill")
```

```
                    (address "37 Cherry Lane"))
==> Activation 0 process-moved-information: f-2,f-4
FIRE 3 process-moved-information: f-2,f-4
<== f-4     (person (name "John Hill")
                    (address "37 Cherry Lane"))
==> f-5     (person (name "John Hill")
                    (address "37 Cherry Lane"))
==> Activation 0 process-moved-information: f-2,f-5
CLIPS>
```

Программа входит в бесконечный цикл, поскольку в ней вводится новый факт `person` после модификации факта, связанного с переменной `?f2`, в результате чего повторно активизируется правило `process-moved-information`. Напомним, что команда `modify` рассматривается как команда `retract`, за которой следует команда `assert`. А поскольку в команде `run` было указано, что должен быть произведен запуск только трех правил, программа останавливается именно после этого. Если же в команде `run` не был бы задан такой лимитирующий параметр, то программа проходила бы по циклу бесконечно. Единственный способ остановить бесконечный цикл, подобный этому, состоит в прерывании работы системы CLIPS с помощью комбинации клавиш <Ctrl+C> или соответствующей команды прерывания для операционной системы, под управлением которой работает система CLIPS.

7.22 Однозначные подстановочные символы

Иногда возникает необходимость проверить наличие некоторого поля в слоте без фактического присваивания значения переменной. Такая возможность особенно полезна при работе с многозначными слотами. Например, предположим, что необходимо вывести номер карточки социального обеспечения для каждого человека с указанной фамилией. Ниже показаны конструкция `deftemplate`, используемая для описания каждого человека; конструкция `deffacts`, в которой заранее заданы сведения о некоторых людях; а также правило, с помощью которого осуществляется вывод номера карточки социального обеспечения каждого человека с указанной фамилией.

```
(deftemplate person
   (multislot name)
   (slot social-security-number))
(deffacts some-people
   (person (name John Q. Public)
           (social-security-number 483-98-9083))
   (person (name Jack R. Public)
```

```
          (social-security-number 483-98-9084)))
(defrule print-social-security-numbers
   (print-ss-numbers-for ?last-name)
   (person (name ?first-name ?middle-name
                 ?last-name)
           (social-security-number ?ss-number))
   =>
   (printout t ?ss-number crlf))
```

Правило `print-social-security-numbers` связывает имя и отчество рассматриваемого лица соответственно с переменными `?first-name` и `?middle-name`. Но на эти переменные нет ссылок либо в действиях правой части правила или в других шаблонах, либо в слотах в левой части правила. Вместо использования переменной, когда потребуется подстановка некоторого поля, но его значение не представляет интереса, можно применить **однозначный подстановочный символ**, который представляет собой вопросительный знак. С помощью однозначных подстановочных символов правило `print-social-security-numbers` можно перезаписать следующим образом:

```
(defrule print-social-security-numbers
   (print-ss-numbers-for ?last-name)
   (person (name ? ? ?last-name)
   (social-security-number ?ss-number))
   =>
   (printout t ?ss-number crlf))
```

Обратите внимание на то, что слот `name` факта `person` должен содержать точно три поля, чтобы обеспечивалась возможность активизировать правило `print-social-security-numbers`. Например, следующий факт не будет удовлетворять правилу `print-social-security-numbers`:

```
(person (name Joe Public)
        (social-security-number 345-89-3039))
```

Следует отметить, насколько важно не включить пробел между вопросительным знаком и символическим именем переменной. В частности, такой шаблон предусматривает использование двух полей для слота `name`:

```
(person (name ?first ?last))
```

а следующий шаблон предусматривает применение для слота `name` трех полей, причем последним полем должен быть символ `last`:

```
(person (name ?first ? last))
```

Если в некотором шаблоне остается незаданным некоторый однозначный слот, то система CLIPS автоматически трактует его таким образом, как если бы для это-

го слота была предусмотрена проверка с помощью однозначного подстановочного символа. Например, шаблон

```
(person (name ?first ?last))
```

эквивалентен такому шаблону:

```
(person (name ?first ?last)
(social-security-number ?))
```

7.23 Мир блоков

Для демонстрации возможностей средств связывания переменных разработаем программу, управляющую перемещением блоков в простом мире блоков. Программа такого типа аналогична классической программе для мира блоков, в которой область знаний ограничена теми знаниями, которые касаются блоков [25, с. 224–226]. Решение задачи перемещения блоков представляет собой полноценный пример планирования, которое может применяться для автоматизации производственных процессов, связанных с перемещением деталей с помощью манипулятора.

В мире блоков интерес представляют только блоки. Блоки могут складываться в столбики с учетом того, что на каждый блок может устанавливаться один и только один блок. Перед программой в мире блоков ставится цель — изменить взаимное расположение блоков в столбиках и получить требуемую конфигурацию с помощью минимального количества ходов. В рассматриваемом примере можно принять целый ряд упрощающих ограничений. Первое из этих ограничений состоит в том, что допускается лишь одна начальная цель, которая может состоять только в том, чтобы один блок был перемещен на верхнюю поверхность другого. Благодаря этому ограничению задача определения оптимальных ходов, позволяющих достичь намеченной цели, становится довольно тривиальной. Если цель состоит в том, чтобы переместить блок `x` на верхнюю поверхность блока `y`, то достаточно переместить все блоки (если таковые имеются), стоящие над блоком `x`, на пол, а также переместить на пол все блоки, стоящие на блоке `y` (если таковые имеются), а затем поставить блок `x` на блок `y`.

В качестве второго ограничения можно принять такое условие, чтобы не выдвигалась какая-либо цель, которая уже была достигнута. Это означает, что цель не может состоять в перемещении блока `x` на блок `y`, если блок `x` уже стоит на блоке `y`. Проверка соблюдения такого условия является довольно простой; тем не менее синтаксис, с помощью которого можно было бы проверить соблюдение указанного условия, будет представлен только в главе 8.

Прежде чем приступить к решению сформулированной задачи, целесообразно определить конфигурацию блоков, которая должна использоваться для проверки программы. Применяемая конфигурация показана на рис. 7.2. Эта конфигурация

состоит из двух столбиков. В первом столбике блок A стоит на блоке B, стоящем на блоке C. Во втором столбике блок D стоит на блоке E, стоящем на блоке F.

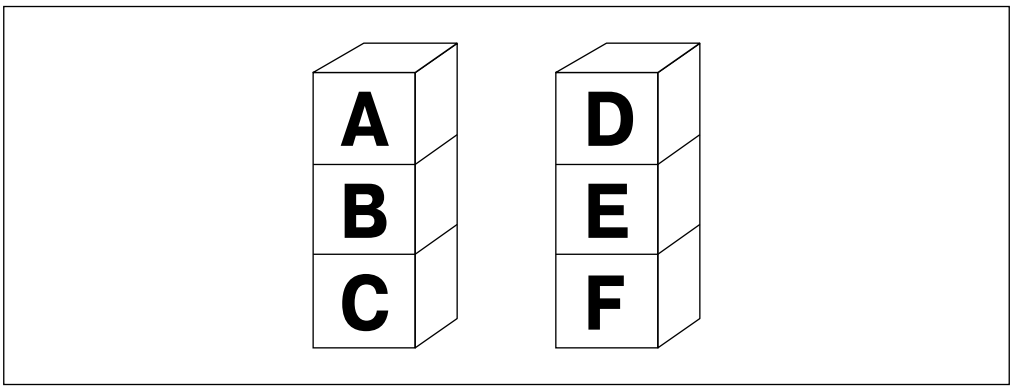


Рис. 7.2. Начальная конфигурация в мире блоков

Чтобы определить, какие типы правил позволят эффективно решить эту задачу, имеет смысл попытаться достичь цели в мире блоков, рассматривая последовательно один ход за другим. Какие этапы должны быть выполнены для перемещения блока C на блок E? Простейшим решением этой задачи было бы непосредственное перемещение C на E. Но правило, подразумевающее такое решение, можно было бы применить, только если ни на блоке C, ни на блоке E не стоят другие блоки. Ниже приведен псевдокод для данного правила.

```
RULE MOVE-DIRECTLY
IF   The goal is to move block ?upper on top
        of block ?lower and
     block ?upper is the top block in its stack and
     block ?lower is the top block in its stack,
THEN Move block ?upper on top of block ?lower.
```

В рассматриваемом случае правило `move-directly` использовать невозможно, поскольку на блоке C стоят блоки A и B, а на блоке E стоит блок D. Для того чтобы обеспечить применение правила `move-directly` для перемещения блока C на блок E, необходимо переместить блоки A, B и D на пол. Поскольку это — простейший способ избавиться от указанных блоков, именно он и применяется. В рассматриваемом простом мире блоков не требуется, чтобы блоки снова устанавливались в другие столбики, кроме того, допускается лишь единственная начальная цель, поэтому не требуется составлять блоки в столбики при удалении их таким образом. Данное правило может быть выражено в виде двух приведенных ниже правил на псевдокоде: одно правило предназначено для снятия блоков с того блока, который подлежит перемещению, а другое правило позволяет сни-

мать блоки с того блока, на который в дальнейшем должен быть поставлен другой блок.

```
RULE CLEAR-UPPER-BLOCK
IF   The goal is to move block ?x and
     block ?x is not the top block in its stack and
     block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor
RULE CLEAR-LOWER-BLOCK
IF   The goal is to move another block on top of
     block ?x and
     block ?x is not the top block in its stack and
     block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor
```

С помощью правила `clear-upper-block` блоки будут сниматься с блока `C`. Это правило вначале позволяет определить, что на пол следует переместить блок `B`. Кроме того, то же самое правило позволит определить, что для перемещения на пол блока `B` необходимо переместить на пол блок `A`. Аналогичным образом, правило `clear-lower-block` позволяет определить, что на пол необходимо переместить блок `D`, для того чтобы можно было что-то поставить на блок `E`.

К этому времени появились следующие подцели: переместить на пол блоки `A`, `B` и `D`. Блоки `A` и `D` могут быть сразу же перемещены на пол. Если правило `move-directly` будет сформулировано должным образом, то с его помощью появится возможность обеспечить перемещение, наряду с другими блоками, а также и блоков, находящихся на полу. Дело в том, что пол фактически не является блоком, поэтому может потребоваться трактовать связанные с ним операции по-другому. Ниже приведено правило на псевдокоде, в котором учитывается частный случай перемещения блока на пол.

```
RULE MOVE-TO-FLOOR
IF   The goal is to move block ?upper on top of the floor
        and block ?upper is the top block in its stack,
THEN Move block ?upper on top of the floor.
```

Теперь можно воспользоваться правилом `move-to-floor` для перемещения блоков `A` и `D` на пол. А после перемещения на пол блока `A` правило `move-to-floor` можно активизировать для перемещения на пол блока `B`. Наконец, после того как блоки `A`, `B` и `D` будут на полу, блоки `C` и `E` окажутся верхними блоками своих столбиков и поэтому появится возможность применить правило `move-directly` для перемещения блока `C` на блок `E`.

После формулировки необходимых правил на псевдокоде необходимо приступить к определению фактов, которые будут использованы в правилах. Следует

отметить, что без подготовки некоторых прототипов не всегда удается установить необходимые типы фактов. А в данном случае на необходимость применения нескольких типов фактов указывают сами правила, сформулированных на псевдокоде. Например, очень важна информация о том, какие блоки стоят на других блоках. Эту информацию можно представить с помощью следующей конструкции deftemplate:

```
(deftemplate on-top-of
   (slot upper)
   (slot lower))
```

Факты, описанные с помощью этого шаблона, принимают такой вид:

```
(on-top-of (upper A) (lower B))
(on-top-of (upper B) (lower C))
(on-top-of (upper D) (lower E))
(on-top-of (upper E) (lower F))
```

Важно также знать, какие блоки находятся в верхней и нижней частях каждого столбика, поэтому следует включить такие факты:

```
(on-top-of (upper nothing) (lower A))
(on-top-of (upper C) (lower floor))
(on-top-of (upper nothing) (lower D))
(on-top-of (upper F) (lower floor))
```

В этих фактах слова nothing и floor имеют особый смысл. Факты (on-top-of (upper nothing) (lower A)) и (on-top-of (upper nothing) (lower D)) указывают, что A и D являются самыми верхними блоками в своих столбиках. Аналогичным образом, факты (on-top-of (upper C) (lower floor)) и (on-top-of (upper F) (lower floor)) показывают, что нижними блоками в своих столбиках являются блоки C и F. Но включение этих фактов не обязательно должно привести к решению задачи определения верхнего и нижнего блоков в столбике. Если правила не будут сформулированы должным образом, то ключевые слова floor и nothing будут неправильно трактоваться как имена блоков. Поэтому могут потребоваться также факты, позволяющие указывать имена блоков. Ниже перечислены факты, в которых используется подразумеваемая конструкция deftemplate с именем block, которые позволяют отличать имена блоков от специальных слов nothing и floor.

```
(block A)
(block B)
(block C)
(block D)
(block E)
(block F)
```

Наконец, требуется факт, позволяющий описывать достигаемые цели перемещения блоков. Эти цели можно описать с помощью следующей конструкции deftemplate:

```
(deftemplate goal (slot move) (slot on-top-of))
```

В качестве начальной цели сформулируем с помощью конструкции deftemplate следующую цель:

```
(goal (move C) (on-top-of E))
```

Итак, на данный момент определены все факты и конструкции deftemplate, поэтому можно описать начальную конфигурацию рассматриваемого мира блоков с помощью такой конструкции deffacts:

```
(deffacts initial-state
   (block A)
   (block B)
   (block C)
   (block D)
   (block E)
   (block F)
   (on-top-of (upper nothing) (lower A))
   (on-top-of (upper A) (lower B))
   (on-top-of (upper B) (lower C))
   (on-top-of (upper C) (lower floor))
   (on-top-of (upper nothing) (lower D))
   (on-top-of (upper D) (lower E))
   (on-top-of (upper E) (lower F))
   (on-top-of (upper F) (lower floor))
   (goal (move C) (on-top-of E)))
```

Правило move-directly записывается следующим образом:

```
(defrule move-directly
   ?goal <- (goal (move ?block1)
                  (on-top-of ?block2))
   (block ?block1)
   (block ?block2)
   (on-top-of (upper nothing) (lower ?block1))
   ?stack-1 <- (on-top-of (upper ?block1)
   (lower ?block3))
   ?stack-2 <- (on-top-of (upper nothing)
                          (lower ?block2))
   =>
   (retract ?goal ?stack-1 ?stack-2)
```

```
   (assert (on-top-of (upper ?block1)
                      (lower ?block2))
   (on-top-of (upper nothing)
                      (lower ?block3)))
   (printout t ?block1 " moved on top of " ?block2
               "." crlf))
```

Первые три шаблона определяют, что задана цель перемещения одного блока на другой блок, при этом второй и третий шаблоны исключают возможность достижения с помощью данного правила цели перемещения какого-либо блока на пол. С помощью четвертого и шестого шаблонов проверяется, являются ли рассматриваемые блоки верхними блоками в своих столбиках. Пятый и шестой шаблоны согласуются с информацией, необходимой для обновления состояния столбиков, с которых снимаются и на которые ставятся перемещаемые блоки. Обновление информации о столбиках, относящейся к обоим столбикам, и вывод сообщения осуществляются с помощью действий правила. При этом учитывается условие, что блок, находившийся под снятым блоком, становится верхним блоком в своем столбике, а блок, поставленный на другой блок, становится верхним блоком в том столбике, в который он перенесен.

Правило `move-to-floor` реализовано следующим образом:

```
(defrule move-to-floor
   ?goal <- (goal (move ?block1)
   (on-top-of floor))
   (block ?block1)
   (on-top-of (upper nothing) (lower ?block1))
   ?stack <- (on-top-of (upper ?block1)
   (lower ?block2))
   =>
   (retract ?goal ?stack)
   (assert (on-top-of (upper ?block1)
                      (lower floor))
   (on-top-of (upper nothing)
                      (lower ?block2)))
   (printout t ?block1 " moved on top of floor."
               crlf))
```

Это правило аналогично правилу `move-directly`, за исключением того, что информацию о состоянии пола обновлять не требуется, поскольку пол — не блок.

Правило `clear-upper-block` реализовано следующим образом:

```
(defrule clear-upper-block
   (goal (move ?block1))
   (block ?block1)
```

```
   (on-top-of (upper ?block2) (lower ?block1))
   (block ?block2)
   =>
   (assert (goal (move ?block2)
                 (on-top-of floor))))
```

Правило clear-lower-block реализовано следующим образом:

```
(defrule clear-lower-block
   (goal (on-top-of ?block1))
   (block ?block1)
   (on-top-of (upper ?block2) (lower ?block1))
   (block ?block2)
   =>
   (assert (goal (move ?block2)
                 (on-top-of floor))))
```

На этом разработка программы, состоящей из правил move-directly, move-to-floor, clear-upper-block и clear-lower-block, конструкций deftemplate с именами goal и on-top-of, а также конструкции deffacts с именем initial-state, завершается. Пример прогона этой программы для мира блоков показан в следующем выводе:

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (run)↵
A moved on top of floor.
B moved on top of floor.
D moved on top of floor.
C moved on top of E.
CLIPS>
```

Вначале на пол перемещаются блоки A и B для освобождения блока C. Затем на пол перемещается блок D для освобождения блока E. Наконец, появляется возможность переместить блок C на блок E и решить задачу по достижению исходной цели.

В этом примере показано, как разрабатывается программа с использованием поэтапного метода. Вначале правила записываются на псевдокоде, формулировки которого напоминают текст на естественном языке. Затем эти правила на псевдокоде используются для того, чтобы определить, факты какого типа потребуются в программе. После этого определяются конструкции deftemplate с описанием проектируемых фактов и с помощью этих конструкций deftemplate формулируются начальные знания, относящиеся к данной программе. Наконец, правила на

псевдокоде преобразуются в правила CLIPS, а в качестве руководящих указаний для такого преобразования используются конструкции `deftemplate`.

Для разработки экспертной системы, как правило, требуется значительно больший объем работы по созданию прототипов и приходится проводить намного больше итераций, чем в данном примере. Кроме того, не всегда возможно определить наилучший метод представления фактов или выбрать правила таких типов, которые обеспечат успешное создание экспертной системы. Тем не менее соблюдение единообразной методологии может помочь при разработке экспертной системы, даже если потребуется создание большого количества прототипов и использование нескольких итераций разработки.

7.24 Многозначные подстановочные символы и переменные

Многозначные подстановочные символы

Многозначные подстановочные символы и переменные могут применяться для согласования с полями шаблона, количество которых может составлять от нуля и больше. **Многозначный подстановочный символ** обозначается знаком доллара, за которым следует вопросительный знак, `$?`, и представляет количество вхождений некоторого поля, составляющее от нуля и больше. Следует отметить, что обычные переменные и подстановочные символы согласуются точно с одним полем. На первый взгляд это различие кажется незначительным, но является очень важным. В качестве примера использования многозначных подстановочных символов еще раз рассмотрим правило `print-social-security-numbers`, описанное в разделе 7.22:

```
(defrule print-social-security-numbers
   (print-ss-numbers-for ?last-name)
   (person (name ? ? ?last-name)
   (social-security-number ?ss-number))
   =>
   (printout t ?ss-number crlf))
```

Это правило согласуется только с таким слотом `name`, в котором имеются точно три поля. Поэтому следующий факт не согласуется с данным правилом:

```
(person (name Joe Public)
        (social-security-number 345-89-3039))
```

Но если два однозначных подстановочных символа будут заменены одним многозначным подстановочным символом (как показано ниже), то шаблон `person`

станет согласовываться с любым слотом name, содержащим по меньшей мере одно поле и включающим в качестве своего последнего поля указанное имя лица:

```
(defrule print-social-security-numbers
   (print-ss-numbers-for ?last-name)
   (person (name $? ?last-name)
   (social-security-number ?ss-number))
   =>
   (printout t ?ss-number crlf))
```

Аналогичным образом, если в шаблоне останется незаданным какой-либо многозначный слот, то система CLIPS будет автоматически трактовать эту ситуацию так, как если бы для данного слота была предусмотрена проверка многозначного подстановочного символа. Например, шаблон

```
(person (social-security-number ?ss-number))
```

эквивалентен шаблону

```
(person (name $?)
(social-security-number ?ss-number))
```

Многозначные переменные

Однозначные переменные обозначаются префиксом ?, а многозначные переменные — префиксом $?. Ниже приведены конструкции, которые показывают, как вывести список имен всех детей указанного лица.

```
(deftemplate person
   (multislot name)
   (multislot children))
(deffacts some-people
   (person (name John Q. Public)
           (children Jane Paul Mary))
   (person (name Jack R. Public)
           (children Rick)))
(defrule print-children
   (print-children $?name)
   (person (name $?name)
           (children $?children))
   =>
   (printout t ?name " has children " ?children
               crlf))
```

Первый шаблон правила print-children связывает имя лица, для которого должен быть создан список имен детей, с переменной $?name. Второй шаблон

согласовывает факт person с указанным именем, содержащимся в переменной $?name, а затем связывает список детей этого лица с переменной $?children. После этого полученное значение выводится с помощью действия, заданного в правой части указанного правила.

Обратите внимание на то, что при формировании ссылки на многозначную переменную в правой части правила не обязательно включать знак $ в состав имени переменной. Знак $ используется только в левой части для указания на то, что количество полей, которые могут быть связаны с этой переменной, составляет от нуля и больше.

Ниже приведен диалог, который демонстрирует действие правила print-children.

```
CLIPS> (reset)↵
CLIPS> (assert (print-children John Q. Public))↵
<Fact-3>
CLIPS> (run)↵
(John Q. Public) has children (Jane Paul Mary)
CLIPS>
```

Обратите внимание на то, что при выводе на внешнее устройство многозначные значения, связанные с переменными ?name и ?children, заключаются в круглые скобки.

В каждом отдельном слоте допускается использование больше чем одной многозначной переменной. Предположим, например, что требуется найти всех людей, имеющих ребенка с указанным именем. Данная задача решается с помощью следующего правила:

```
(defrule find-child
   (find-child ?child)
   (person (name $?name)
           (children $?before ?child $?after))
   =>
   (printout t ?name " has child " ?child crlf)
   (printout t "Other children are "
               ?before " " ?after crlf))
```

Вообще говоря, если бы нас интересовало только значение переменной ?child, то переменные $?before и $?after можно было бы заменить многозначными подстановочными символами (и удалить оператор printout, в котором упоминаются эти переменные). Ниже приведен диалог, который показывает, как действует такое правило find-child.

```
CLIPS> (reset)↵
CLIPS> (assert (find-child Paul))↵
```

```
<Fact-3>
CLIPS> (run)↵
(John Q. Public) has child Paul
Other children are (Jane) (Mary)
CLIPS> (assert (find-child Rick))↵
<Fact-4>
CLIPS> (run)↵
(Jack R. Public) has child Rick
Other children are () ()
CLIPS> (assert (find-child Bill))↵
<Fact-5>
CLIPS> (run)↵
CLIPS>
```

Если с переменной `?child` связывается имя ребенка `Paul`, то с переменной `$?before` связывается значение `(Jane)`, а с переменной `$?after` — значение `(Mary)`. Аналогичным образом, если с переменной `?child` связывается значение `Rick`, то с переменной `$?before` связывается значение `()`, т.е. многозначное значение, содержащее нуль полей, а с переменной `$?after` также связывается значение `()`.

Применение нескольких различных способов согласования с шаблонами

До сих пор в данной главе рассматривались лишь такие ситуации, в которых единственный факт мог быть согласован с шаблоном только одним способом. Если же используются многозначные переменные или подстановочные символы, то появляется возможность согласования с шаблоном больше чем одним способом. Предположим, что введен факт с данными о некотором лице, назвавшем всех своих детей так же, как зовут его:

```
CLIPS> (reset)↵
CLIPS> (assert (person (name Joe Fiveman)
                       (children Joe Joe Joe)))↵
<Fact-3>
CLIPS> (assert (find-child Joe))↵
<Fact-4>
CLIPS> (agenda)↵
0     find-child: f-4,f-3
0     find-child: f-4,f-3
0     find-child: f-4,f-3
For a total of 3 activations.
```

```
CLIPS> (run)↵
(Joe Fiveman) has child Joe
Other children are () (Joe Joe)
(Joe Fiveman) has child Joe
Other children are (Joe) (Joe)
(Joe Fiveman) has child Joe
Other children are (Joe Joe) ()
CLIPS>
```

Как показывают результаты запуска правил, существуют три различных способа связывания переменных `?child`, `$?before` и `$?after` с фактом `f-3`. В первом случае переменная `$?before` связывается со значением `()`, `?child` — со значением `Joe`, а переменная `$?after` — со значением `(Joe Joe)`. Во втором случае переменная `$?before` связывается со значением `(Joe)`, переменная `?child` — со значением `Joe`, а переменная `$?after` — со значением `(Joe)`. В третьем случае переменная `$?before` связывается со значением `(Joe Joe)`, переменная `?child` — со значением `Joe`, а переменная `$?after` — со значением `()`.

Реализация стека

Стек — это упорядоченная структура данных, обеспечивающая добавление и удаление элементов. Элементы добавляются и удаляются только на одном из "концов" стека. В одной операции может осуществляться добавление нового элемента к стеку (элемент задвигается в стек), а в другой операции последний добавленный элемент удаляется (выталкивается) из стека. В стеке первый добавленный элемент становится последним удаляемым элементом, а последний добавленный элемент — первым удаляемым элементом.

В качестве наглядной аналогии для понятия стека может служить стопка подносов в кафетерии. Новые подносы добавляются (задвигаются) поверх подносов, уже находящихся в стеке. Последние подносы, наложенные поверх имеющихся в стопке подносов, становятся первыми снимаемыми (выталкиваемыми) подносами.

С помощью многозначных переменных можно относительно легко реализовать стек, который способен выполнять операции `push` (задвинуть) и `pop` (вытолкнуть). Прежде всего необходимо предусмотреть упорядоченный факт `stack`, который содержит список элементов. Ниже приведено правило, с помощью которого можно задвинуть значение в факт `stack`.

```
(defrule push-value
   ?push-value <- (push-value ?value)
   ?stack <- (stack $?rest)
   =>
```

```
   (retract ?push-value ?stack)
   (assert (stack ?value $?rest))
   (printout t "Pushing value " ?value crlf))
```

А для реализации действия по выталкиванию необходимо предусмотреть два правила: относящееся к пустому стеку и относящееся к стеку, в котором имеется значение, предназначенное для выталкивания:

```
(defrule pop-value-valid
   ?pop-value <- (pop-value)
   ?stack <- (stack ?value $?rest)
   =>
   (retract ?pop-value ?stack)
   (assert (stack $?rest))
   (printout t "Popping value " ?value crlf))
(defrule pop-value-invalid
   ?pop-value <- (pop-value)
   (stack)
   =>
   (retract ?pop-value)
   (printout t "Popping from empty stack" crlf))
```

Эти правила можно легко модифицировать таким образом, чтобы с их помощью задвигать и выталкивать значения в именованных стеках. Например, шаблоны

```
?push-value <- (push-value ?value)
?stack <- (stack $?rest)
```

можно заменить на такие шаблоны, в которых переменная `?name` представляет имя стека:

```
?push-value <- (push-value ?name ?value)
?stack <- (stack ?name $?rest)
```

Еще один вариант программы для мира блоков

Задачу составления плана для мира блоков можно решить иначе, намного проще, с использованием многозначных подстановочных символов и переменных. Для этого, как показано ниже, применяется возможность представить каждый столбик блоков с помощью единственного факта. Операции по перемещению блоков аналогичны операциям, применяемым в примере реализации операций `push` и `pop`:

```
(deffacts initial-state
   (stack A B C)
```

```
   (stack D E F)
   (goal (move C) (on-top-of E))
   (stack))
```

Пустой факт `stack` включен для того, чтобы исключить возможность добавления этого факта в дальнейшем; например, это может произойти, если в столбце имеется только один блок, а этот блок перемещается на другой.

Ниже приведены правила программы для мира блоков, в которой используются многозначные переменные.

```
(defrule move-directly
   ?goal <- (goal (move ?block1)
   (on-top-of ?block2))
   ?stack-1 <- (stack ?block1 $?rest1)
   ?stack-2 <- (stack ?block2 $?rest2)
   =>
   (retract ?goal ?stack-1 ?stack-2)
   (assert (stack $?rest1))
   (assert (stack ?block1 ?block2 $?rest2))
   (printout t ?block1 " moved on top of "
   ?block2 "." crlf))
(defrule move-to-floor
   ?goal <- (goal (move ?block1) (on-top-of floor))
   ?stack-1 <- (stack ?block1 $?rest)
   =>
   (retract ?goal ?stack-1)
   (assert (stack ?block1))
   (assert (stack $?rest))
   (printout t ?block1 " moved on top of floor."
               crlf))
(defrule clear-upper-block
   (goal (move ?block1))
   (stack ?top $? ?block1 $?)
   =>
   (assert (goal (move ?top) (on-top-of floor))))
(defrule clear-lower-block
   (goal (on-top-of ?block1))
   (stack ?top $? ?block1 $?)
   =>
   (assert (goal (move ?top) (on-top-of floor))))
```

7.25 Резюме

В настоящей главе приведено вводное описание фундаментальных компонентов CLIPS. Первым компонентом системы CLIPS являются факты. Факты формируются из полей, которые могут представлять собой символ, строку, целое число или число с плавающей точкой. Первое поле факта обычно используется для указания типа информации, хранимой в факте, и называется *именем отношения*. Для присваивания имен слотов конкретным полям факта, начиная с указанного имени отношения, используется конструкция `deftemplate`. Для определения фактов как начальных знаний служит конструкция `deffacts`.

Вторым компонентом системы CLIPS являются правила. Каждое правило состоит из левой и правой части. Левая часть правила может рассматриваться как часть `IF`, а правая часть — как часть `THEN`. Правила могут иметь несколько шаблонов и действий.

Третьим компонентом системы CLIPS является машина логического вывода. Правила, шаблоны которых удовлетворяются фактами (т.е. проверка шаблонов на соответствие фактам оканчивается успешно), активизируются, в результате чего активизированные правила помещаются в рабочий список правил. А наличие в системе CLIPS свойства релаксации исключает возможность постоянной активизации одних и тех же правил под действием уже воспринятых фактов.

Кроме того, в настоящей главе дано вводное описание такого понятия, как переменные. Переменные используются для выборки информации от фактов и ограничения значений слотов при сопоставлении с шаблонами в левой части правила. Переменные могут сохранять адреса фактов, сопоставленных с шаблонами в левой части правила, что позволяет извлекать факты, связанные с шаблонами, в правой части правила. Если поле, с которым должно быть выполнено согласование, может иметь любое значение и его значение в дальнейшем не требуется в левой или в правой части правила, то вместо переменных могут использоваться однозначные подстановочные символы. Многозначные переменные и подстановочные символы позволяют согласовывать факты больше чем с одним полем в шаблоне.

Задачи

7.1. Преобразуйте следующие предложения в факты, заданные в операторе `deffacts`. Для каждой группы взаимосвязанных фактов определите конструкцию `deftemplate`, которая описывает более общее отношение.

```
The father of John is Tom.
The mother of John is Susan.
The parents of John are Tom and Susan.
Tom is a father.
Susan is a mother.
```

```
John is a son.
Tom is a male.
Susan is a female.
John is a male.
```

7.2. Определите конструкцию `deftemplate` для факта, содержащего информацию о некотором множестве. Конструкция `deftemplate` должна включать информацию об имени или описании множества, содержать список элементов в множестве и указывать, является ли это множество подмножеством другого множества. Представьте следующие множества как факты с использованием формата, определяемого предложенной вами конструкцией `deftemplate`:

```
A = { 1, 2, 3 }
B = { 1, 2, 3, red, green }
C = { red, green, yellow, blue }
```

7.3. Массив с разреженным заполнением содержит относительно немного элементов, отличных от нуля. Его можно было бы представить более эффективно в виде связного списка или дерева. Как можно представить массив с разреженным заполнением с использованием фактов? Опишите конструкцию `deftemplate`, на основе которой факты могут использоваться для представления массива. Каковы возможные недостатки массива с использованием фактов, в отличие от способа представления массива с использованием структуры данных типа массива в процедурном языке?

7.4. Преобразуйте сеть общего вида, на которой представлены авиалинии, показанную на рис. 2.4, *а* (см. с. 146), в ряд фактов, заданных в операторе `deffacts`. Для описания фактов используйте единственную конструкцию `deftemplate`.

7.5. Преобразуйте семантическую сеть, представляющую семью, которая показана на рис. 2.4, *б* (см. с. 146), в ряд фактов, заданных в операторе `deffacts`. Для описания сформированных фактов используйте несколько конструкций `deftemplate`.

7.6. Преобразуйте семантическую сеть, которая показана на рис. 2.5 (см. с. 147), в ряд фактов, заданных в операторе `deffacts`. Для описания фактов используйте несколько конструкций `deftemplate`. Например, связи `IS-A` и `AKO` должны стать именами отношений, и для каждой из них должна быть предусмотрена собственная конструкция `deftemplate`.

7.7. Преобразуйте бинарное дерево решений, представляющее информацию о классификации животных на рис. 3.3 (см. с. 197), в ряд фактов, заданных в операторе `deffacts`. Покажите, как могут быть представлены связи

между узлами. Требуется ли для листовых узлов дерева такое представление, которое отличается от представления для других узлов дерева?

7.8. Реализуйте семантическую сеть, рассматриваемую в задаче 2.1, как оператор `deffacts` с использованием конструкций `deftemplate` с именами `AKO` и `IS-A`.

7.9. Для надлежащего роста растений требуется много разных типов питательных веществ. Тремя наиболее важными питательными веществами для растений, которые поступают с удобрениями, являются азот, фосфор и калий. В случае нехватки одного из этих питательных веществ возникают различные симптомы. Преобразуйте приведенные ниже эвристики в правила, позволяющие определить дефицит питательного вещества. Примите предположение, что в условиях сбалансированного питания растение имеет зеленый цвет.

```
A plant with stunted growth may have a nitrogen
   deficiency.
A plant that is pale yellow in color may have a
   nitrogen deficiency.
A plant that has reddish-brown leaf edges may have a
   nitrogen deficiency.
A plant with stunted root growth may have a phosphorus
   deficiency.
A plant with a spindly stalk may have a phosphorus
   deficiency.
A plant that is purplish in color may have a
   phosphorus deficiency.
A plant that has delayed in maturing may have a
   phosphorus deficiency.
A plant with leaf edges that appear scorched may have
   a potassium deficiency.
A plant with weakened stems may have a potassium
   deficiency.
A plant with shriveled seeds or fruits may have a
   potassium deficiency.
```

Определите конструкции `deftemplate`, необходимые для описания фактов, которые используются в правилах. Ввод данных в программу должен осуществляться путем внесения в список фактов в виде фактов информации о симптомах. В выводе программы, отображаемом на терминале, должно быть указано, каких питательных веществ недостает растению. Реализуйте такой метод, который позволил бы избежать вывода нескольких сообщений, касающихся дефицита одного и того же вещества, вызванных наличием

больше чем одного симптома. Проверьте работу своей программы со следующими входными данными:

```
The plant has stunted root growth.
The plant is purplish in color.
```

7.10. Пожары классифицируются в соответствии с тем, каковым является основной участвующий в них горючий материал. Преобразуйте следующую информацию в правила, применяемые для определения категории пожара:

```
Type A fires involve ordinary combustibles such as
   paper, wood, and cloth.
Type B fires involve flammable and combustible
   liquids (such as oil and gas), greases, and
   similar materials.
Type C fires involve energized electrical equipment.
Type D fires involve combustible metals such as
   magnesium, sodium, and potassium.
```

Типы огнетушителей, которые должны использоваться для тушения пожара, зависят от категории пожара. Преобразуйте следующую информацию в правила:

```
Class A fires should be extinguished with
   heat-absorbing or combustion-retarding
   extinguishers such as water or water-based liquids
   and dry chemicals.
Class B fires should be extinguished by excluding air,
   inhibiting the release of combustible vapors, or
   interrupting the combustion chain reaction.
   Extinguishers include dry chemicals, carbon
   dioxide, foam, and bromotrifluoromethane.
Class C fires should be extinguished with a
   nonconducting agent to prevent short circuits. If
   possible the power should be cut. Extinguishers
   include dry chemicals, carbon dioxide, and
   bromotrifluoromethane.
Class D fires should be extinguished with smothering
   and heat-absorbing chemicals that do not react
   with the burning metals. Such chemicals include
   trimethoxyboroxine and screened graphitized coke.
```

Опишите факты, используемые в правилах. Ввод данных в программу должен осуществляться путем внесения в список фактов информации о типе горючего материала в виде факта. В выводе программы необходимо указать,

какие огнетушители могут использоваться и какие прочие действия должны быть предприняты, например отключение подачи электроэнергии. Продемонстрируйте работу своей программы на одном горючем материале из тех, что относятся к каждой категории пожара.

7.11. К облакам нижнего яруса, находящимся на высоте 1800 м или меньше, относятся слоистые и слоисто-кучевые облака. К облакам среднего яруса, находящимся на высоте от 1800 до 6000 м, относятся высокослоистые, высококучевые и слоисто-дождевые облака. К облакам верхнего яруса, находящимся на высоте больше 6000 м, относятся перистые, перисто-слоистые и перисто-кучевые облака. Кучевые и кучево-дождевые облака могут простираться от нижнего до верхнего яруса. Кучевые, слоисто-кучевые, высококучевые, кучево-дождевые и перисто-кучевые облака выглядят как крупные скругленные груды. Слоистые, высокослоистые, слоисто-дождевые и перисто-слоистые облака напоминают гладкие, ровные листы. Перистые облака с виду кажутся тонкими, как пучки волос. Слоисто-дождевые и кучево-дождевые облака несут за собой осадки и имеют темно-серый цвет. Напишите программу, позволяющую идентифицировать типы облаков. В качестве входных данных для программы должны использоваться факты, описывающие атрибуты облака. После этого программа должна выводить информацию о типе идентифицированного облака.

7.12. Один из способов классификации звезды предусматривает их разбиение на цветовые группы, называемые спектральными классами. Классы охватывают звезды со светимостью во всем спектре видимого света, начиная со звезд синего спектрального класса “O” и заканчивая звездами красного спектрального класса “M”; промежуточное положение между ними занимают звезды желтого спектрального класса “G”. Спектральный класс звезды связан с ее температурой: звезды класса “O” имеют температуру больше 37 000°F; звезды “B” класса имеют температуру от 17 001°F до 37 000°F; звезды класса “A” имеют температуру от 12 501°F до 17 000°F; звезды класса “F” имеют температуру от 10 301°F до 12 500°F; звезды класса “G” имеют температуру от 8 001°F до 10 300°F; звезды класса “K” имеют температуру от 5 501°F до 8 000°F и звезды класса “M” имеют температуру от 5 500°F или меньше. Звезды могут также классифицироваться по величине, которая является мерой их яркости. Чем ниже величина, тем ярче звезда. Может быть принято предположение, что величины находятся в пределах от -7 до 15. В табл. 7.1 перечислены некоторые самые яркие, наиболее широко известные звезды с указанием спектрального класса и величины, а также их расстояния от Земли в световых годах. Напишите программу, которая принимает на входе два факта, представляющие спектральный класс и величину звезды. После этого программа должна вывести следующую информацию

в указанном порядке: все звезды, имеющие заданный спектральный класс; все звезды, имеющие заданную величину; наконец, все звезды, соответствующие и заданному спектральному классу, и заданной величине, наряду с их расстоянием от Земли в световых годах.

Таблица 7.1. Некоторые самые яркие и наиболее широко известные звезды

Звезда	Спектральный класс	Величина	Расстояние от Земли
Сириус	A	1	8,8
Канопус	F	−3	98
Арктур	K	0	36
Вега	A	1	26
Капелла	G	−1	46
Ригель	B	−7	880
Процион	F	3	11
Бетельгейзе	M	−5	490
Альтаир	A	2	16
Альдебаран	K	−1	68
Спика	B	−3	300
Антарес	M	−4	250
Поллукс	K	1	35
Денеб	A	−7	1630

7.13. В табл. 7.2 приведены характеристики наиболее широко известных драгоценных камней, включая их твердость (сопротивление внешним усилиям, измеряемое по шкале твердости Моса), плотность (отношение массы к единице объема, измеряемое в граммах на кубический сантиметр) и цвета. Запишите правила, необходимые для того, чтобы определить, является ли драгоценный камень хризобериллом, если даны три факта, представляющие твердость, плотность и цвет драгоценного камня.

7.14. Напишите программу CLIPS, которая помогает в процессе выбора кустарника, подходящего для посадки. В табл. 7.3 перечислено несколько видов кустарников и указано, обладает ли каждое из этих растений определенными характеристиками, которые включают устойчивость к холоду, к затенению, к засухе, к влажной и кислой почве, к городским условиям (к загазованности воздуха), пригодность для кадочного выращивания, простота культивирования и быстрота роста. Отметка в таблице указывает, что кустарник обладает соответствующей характеристикой. В качестве входных данных

Таблица 7.2. Характеристики наиболее широко известных драгоценных камней

Драгоценный камень	Твердость	Плотность	Цвет
Алмаз	10	3,52	Желтый, коричневый, зеленый, синий, белый, бесцветный
Корунд	9	4	Красный, розовый, желтый, коричневый, зеленый, синий, фиолетовый, черный, белый, бесцветный
Хризоберилл	8,5	3,72	Желтый, коричневый, зеленый
Шпинель	8	3,6	Красный, розовый, желтый, коричневый, зеленый, синий, фиолетовый, белый, бесцветный
Топаз	8	3,52–3,56	Красный, розовый, желтый, коричневый, синий, фиолетовый, белый, бесцветный
Берилл	7,5–8,0	2,7	Красный, розовый, желтый, коричневый, зеленый, синий, белый, бесцветный
Циркон	6–7,5	4,7	Желтый, коричневый, зеленый, фиолетовый, белый, бесцветный
Кварц	7	2,65	Красный, розовый, зеленый, синий, фиолетовый, белый, черный, бесцветный
Турмалин	7	3,1	Красный, розовый, желтый, коричневый, зеленый, синий, белый, черный, бесцветный
Перидот	6,5–7	3,3	Желтый, коричневый, зеленый
Жадеит	6,5–7	3,3	Красный, розовый, желтый, коричневый, зеленый, синий, фиолетовый, белый, черный, бесцветный
Опал	5,5–6,5	2–2,2	Красный, розовый, желтый, коричневый, белый, черный, бесцветный
Нефрит	5–6	2,9–3,4	Зеленый, белый, черный, бесцветный
Бирюза	5–6	2,7	Синий

для программы должны использоваться факты, указывающие желательную характеристику, которую должен иметь кустарник, а результатом работы программы должен быть список растений, обладающих всеми необходимыми характеристиками.

Таблица 7.3. Характеристики кустарников различных видов

Кустарник	Устойчивость к холоду	Устойчивость к затенению	Устойчивость к засухе	Устойчивость к влажной почве	Устойчивость к кислой почве	Устойчивость к городским условиям	Пригодность для кадочного выращивания	Простота культивирования	Быстрота роста
Древовидная гортензия		•				•	•		•
Олеандр						•	•	•	•
Войлочная восковница	•	•	•	•		•		•	•
Жимолость						•	•	•	•
Гардения		•			•		•		
Обыкновенный можжевельник	•		•		•	•		•	
Ольховолистная клетра	•	•		•	•			•	
Татарский кизил	•	•		•		•		•	
Японская аукуба		•	•				•	•	
Канадский рододендрон		•		•	•		•		

7.15. В стеке первое введенное значение становится последним удаляемым значением, а последнее введенное значение — первым удаляемым значением. Очередь действует по противоположному принципу — первое введенное значение становится первым удаляемым значением, а последнее введенное значение — последним удаляемым значением. Запишите правила, которые вводят и удаляют значения в очереди. Примите предположение, что существует только одна очередь.

7.16. Напишите одно или несколько правил, которые формируют все перестановки исходного факта `base-fact` и выводят их на внешнее устройство. Например, обработка следующего факта:

```
(base-fact red green blue)
```

должна привести к получению такого вывода:

```
Permutation is (red green blue)
Permutation is (red blue green)
Permutation is (green red blue)
Permutation is (green blue red)
Permutation is (blue red green)
Permutation is (blue green red)
```

7.17. Напишите правила, которые переводят конечный автомат из текущего состояния в следующее состояние после ввода факта в следующей форме:

```
(input <value>)
```

Узлы и дуги графа конечного автомата должны быть представлены в виде фактов. После перехода машины в следующее состояние входной факт должен быть извлечен. Проверьте разработанные правила и представления фактов на конечных автоматах, которые показаны на рис. 3.5 (см. с. 200) и 3.6 (см. с. 202).

Глава 8

Развитые средства сопоставления с шаблонами

8.1 Введение

Правила такого типа, которые рассматривались в главе 7, иллюстрируют простые средства сопоставления шаблонов с фактами. А в этой главе приведено вводное описание некоторых понятий, лежащих в основе мощных средств сопоставления фактов с шаблонами и манипулирования фактами. Прежде всего рассматриваются ограничения полей. Затем описаны возможности использования функций CLIPS, в том числе рассматривается ряд функций для выполнения основных арифметических операций и операций ввода-вывода. После этого обсуждаются различные способы управления ходом выполнения правил. Для создания более мощных и более сложных программ по сравнению с теми, которые рассматривались в предыдущей главе, используются группы правил. Кроме того, демонстрируются методы ввода, сравнения значений и формирования циклов, а также методы задания управляющих знаний с применением правил. К тому же в данной главе представлено несколько других типов условных элементов, кроме условных элементов шаблона. Эти условные элементы позволяют использовать единственное правило для выполнения функций нескольких правил и предоставляют возможность выполнять согласование не только с существующими, но и с несуществующими фактами.

8.2 Ограничения полей

Ограничение поля `not`

Безусловно, простейшие возможности сопоставления с шаблонами предоставляют литеральные константы и результаты связывания переменных, но в языке CLIPS предусмотрены также более мощные операции сопоставления с шаблонами. Эти дополнительные возможности сопоставления с шаблонами будут представлены на примере повторного анализа задачи определения групп людей с конкретным цветом волос и глаз. В частности, предположим, что необходимо найти всех людей, не имеющих каштановый цвет волос. Один из способов решения этой задачи состоит в написании отдельных правил для каждого цвета волос. Например, следующее правило позволяет найти людей с черными волосами (в примерах данного раздела используется конструкция `deftemplate` с именем `person`, которая рассматривалась в разделе 7.19):

```
(defrule black-hair-is-not-brown-hair
   (person (name ?name) (hair black))
   =>
   (printout t ?name " does not have brown hair"
               crlf))
```

А такое правило позволяет найти людей со светлыми волосами:

```
(defrule blonde-hair-is-not-brown-hair
   (person (name ?name) (hair blonde))
   =>
   (printout t ?name " does not have brown hair"
               crlf))
```

Еще одно правило может быть предусмотрено для поиска людей с рыжими волосами. Но проблема, возникающая, если запись правил осуществляется в такой форме, состоит в том, что каждый раз выполняется проверка на то, что волосы рассматриваемого лица не являются каштановыми. В приведенных выше правилах предпринимается попытка проверить это условие окольным путем. Иначе говоря, в этих правилах определяются все цвета волос, отличные от каштанового, и для каждого из них записывается правило. Разумеется, если есть возможность задать все цвета, то данный метод является осуществимым. Но в этом примере было бы проще принять предположение, что цвет волос может быть любым (пусть даже фиолетовым или зеленым).

Один из способов устранения указанной проблемы состоит в использовании **ограничения поля** для регламентации значений, которые может иметь поле в левой части правила. Один из типов ограничений поля называется **соединительным ограничением** (такое название принято потому, что оно используется для соеди-

нения переменных и других ограничений). Соединительные ограничения подразделяются на три типа. Ограничение первого типа называется **ограничение not** и символически обозначается с помощью тильды, ~. Ограничение `not` действует на одно ограничение или переменную, которые непосредственно следуют за ним. Если ограничение, следующее за ограничением `not`, сопоставляется с полем, то под действием ограничения `not` успешное сопоставление становится неудачным. Если же ограничение, следующее за ограничением `not`, сопоставляется с полем, то под действием ограничения `not` неудачное сопоставление становится успешным. По существу, ограничение `not` отрицает результат применения следующего за ним ограничения.

Таким образом, как показано ниже, правило поиска людей, не имеющих каштанового цвета волос, может быть записано гораздо более просто с помощью ограничения `not`.

```
(defrule person-without-brown-hair
   (person (name ?name) (hair ~brown))
   =>
   (printout t ?name " does not have brown hair"
               crlf))
```

Благодаря использованию ограничения `not` появляется возможность записать единственное правило, выполняющее работу многих других правил, в которых требовалось указать каждый возможный цвет волос.

Ограничение поля `or`

Вторым соединительным ограничением является **ограничение or**, которое символически обозначается с помощью вертикальной черты, `|`. Ограничение `or` используется для обеспечения возможности согласовывать с полем шаблона одно или несколько допустимых значений.

Например, следующее правило позволяет найти с помощью ограничения `or` всех людей, имеющих волосы черного или каштанового цвета:

```
(defrule black-or-brown-hair
   (person (name ?name) (hair brown | black))
   =>
   (printout t ?name " has dark hair" crlf))
```

В результате ввода фактов `(person (name Joe) (eyes blue) (hair brown))` и `(person (name Mark) (eyes brown) (hair black))` в списке фактов активизируется это правило, относящееся к каждому из указанных фактов, и вводится в рабочий список правил.

Ограничение поля and

Соединительным ограничением третьего типа является **ограничение and**. Ограничение `and` следует отличать от условного элемента `and`, который рассматривался в разделе 7.15. Ограничения `and` символически обозначается знаком амперсанда, `&`. Как правило, ограничение `and` используется только в сочетании с другими ограничениями; другие возможные способы применения этого ограничения не имеют практического значения.

Один из случаев, в которых ограничение `and` является очень полезным, состоит в том, что в экземпляр связывания переменной помещаются дополнительные ограничения. Например, предположим, что некоторое правило активизируется под действием факта `person`, в котором указано, что цвет волос является каштановым или черным. Как показано в предыдущем примере, шаблон для такого факта можно легко представить с использованием ограничения `or`. Но как после этого определить значение, представляющее цвет волос? Решение этой задачи состоит в том, чтобы связать с помощью ограничения `and` переменную со значением цвета, которое было согласовано с шаблоном, а затем вывести значение этой переменной, как показано ниже.

```
(defrule black-or-brown-hair
   (person (name ?name) (hair ?color&brown | black))
   =>
   (printout t ?name " has " ?color " hair" crlf))
```

В этом случае переменная `?color` будет связана с тем значением цвета, который был сопоставлен с помощью ограничения `brown | black`.

Ограничение `and` может также применяться в сочетании с ограничением `not`. Например, активизация следующего правила происходит в том случае, если цвет волос какого-то человека не является ни черным, ни каштановым:

```
(defrule black-or-brown-hair
   (person (name ?name)
           (hair ?color&~brown&~black))
   =>
   (printout t ?name " has " ?color " hair" crlf))
```

Совместное применение ограничений полей с другими конструкциями

Ограничения полей могут использоваться совместно с переменными и другими литеральными значениями для создания мощных средств сопоставления с шаблонами. Предположим, например, что требуется правило, позволяющее определить, есть ли два таких человека, описания которых соответствуют следующим условиям. У первого человека глаза синие или зеленые, а волосы — не черные.

Второй человек имеет цвет глаз, отличный от цвета глаз первого человека, а его волосы либо рыжие, либо имеют такой же цвет, как и у первого человека. Ниже приведено правило, позволяющее выполнить сопоставление с учетом этих ограничений.

```
(defrule complex-eye-hair-match
   (person (name ?name1)
           (eyes ?eyes1&blue | green)
           (hair ?hair1&~black))
   (person (name ?name2&~?name1)
           (eyes ?eyes2&~?eyes1)
           (hair ?hair2&red|?hair1))
   =>
   (printout t ?name1 " has " ?eyes1 " eyes and "
               ?hair1 " hair" crlf)
   (printout t ?name2 " has " ?eyes2 " eyes and "
               ?hair2 " hair" crlf))
```

Приведенный пример целесообразно рассмотреть более подробно. Ограничение `?eyes1&blue | green` в слоте `eyes` первого шаблона связывает цвет глаз первого человека с переменной `?eyes1`, если значение цвета глаз в рассматриваемом факте сопоставляется либо с синим, либо с зеленым цветом. Ограничение `?hair1&~black` в слоте `hair` первого шаблона связывает переменную `?hair1`, если значение цвета волос в сопоставляемом факте отлично от черного.

Ограничение `?name2&~?name1` в слоте `name` второго шаблона выполняет важную операцию. Оно связывает значение слота `name` факта `person` с переменной `?name2`, если это значение не совпадает со значением переменной `?name1`. Если имена представляют собой уникальные идентификаторы (т.е. в базе данных, под которой подразумевается список фактов, нет данных о двух людях с одинаковыми именами), это ограничение гарантирует, что оба шаблона не будут сопоставляться с одним и тем же фактом. Такой случай не может произойти с этими двумя шаблонами, поскольку цвет глаз второго человека должен отличаться от цвета глаз первого человека. Но этот полезный метод, имеющий много важных приложений, нужно полностью понять. В разделе 12.2 показано, как можно реализовать такой же метод с использованием адресов фактов для обеспечения того, чтобы факты стали различными. Этот метод позволяет применять рассматриваемое правило к данным о людях, имеющих одинаковые имена, но разные цвета волос и глаз.

Ограничение `?eyes2&~?eyes1` в слоте `eyes` второго шаблона выполняет во многом такую же проверку, как и описанное перед этим ограничение. Последнее ограничение, заданное в слоте `hair` второго шаблона, `?hair2&red | ?hair1`, связывает значение цвета волос второго человека с переменной `?hair2`, либо

если цвет волос является рыжим, либо если он имеет такое же значение, как и значение переменной ?hair1. Следует отметить, что переменная уже должна быть связана, если она используется в составе ограничения поля or.

Переменные связываются, только если они представляют собой первое условие в поле и только если они встречаются отдельно или соединяются другими условиями с помощью соединительного ограничения and. Например, попытка применить следующее правило приводит к возникновению ошибки, поскольку переменная ?hair не связана:

```
(defrule bad-variable-use
   (person (name ?name) (hair red | ?hair))
   =>
   (printout t ?name " has " ?hair " hair " crlf))
```

В качестве заключительного замечания, касающегося объединения ограничений, отметим следующее: необходимо учитывать, что некоторые комбинации ограничений не позволяют осуществлять какие-либо полезные действия. Например, использование ограничения and для соединения литеральных констант (например, black&blue) всегда вызывает то, что ограничения не удовлетворяются (единственным исключением может быть лишь применение литералов, которые являются идентичными. Аналогичным образом, связывание отрицаемых литералов с помощью ограничения or (например, ~black|~blue) всегда приводит к тому, что ограничение удовлетворяется.

8.3 Функции и выражения

Элементарные математические функции

Язык CLIPS позволяет не только обрабатывать символические факты, но и проводить вычисления. Но всегда следует учитывать, что такой язык экспертных систем, как CLIPS, не предназначен для сложных математических расчетов. Безусловно, математические функции CLIPS являются достаточно мощными, но они в основном должны обеспечивать модификацию таких числовых значений, которые применяются для осуществления операций логического вывода с помощью прикладной программы. С другой стороны, для выполнения сложных математических расчетов в большей степени приспособлены другие языки, такие как FORTRAN, в которых почти не предусматривается или вообще не предусматривается проведение рассуждений, касающихся чисел. В языке CLIPS предусмотрена поддержка элементарных арифметических операций, показанных в табл. 8.1. (В приложении Д содержится список прочих математических функций, предусмотренных в языке CLIPS. Дополнительные сведения обо всех функциях, применяемых

в языке CLIPS, приведены в руководстве *Basic Programming Guide* в электронном формате на компакт-диске, прилагаемом к данной книге.)

Таблица 8.1. Элементарные арифметические операции языка CLIPS

Арифметические операции	Назначение
+	Сложение
–	Вычитание
*	Умножение
/	Деление

Для представления числовых выражений в языке CLIPS применяется такой же стиль, как в языке LISP. В языках LISP и CLIPS числовое выражение, которое обычно принято записывать как `2 + 3`, должно быть записано в **префиксной форме**, `(+ 2 3)`. Общепринятый способ записи числовых выражений называется **инфиксной формой**, поскольку в этой форме знаки математических операций записываются между **операндами**, или **параметрами**. А в префиксной форме, применяемой в языке CLIPS, знак операции должен предшествовать операндам, а числовое выражение должно быть заключено в круглые скобки.

Преобразование из инфиксной формы в префиксную является довольно несложным. Например, предположим, что требуется проверить две точки на плоскости, чтобы определить, имеет ли проходящая через них прямая положительный наклон. В обычном инфиксном представлении такое условие можно записать следующим образом:

```
(y2 – y1) / (x2 – x1) > 0
```

Обратите внимание на то, что символ больше, >, представляет собой функцию CLIPS, которая определяет, является ли первый параметр бóльшим, чем второй параметр. (Функция > описана в приложении Д и в документе *Basic Programming Guide*.) Чтобы записать это выражение в префиксной форме, целесообразно начать с такого действия: представить числитель дроби как `(Y)`, а знаменатель — как `(X)`. Поэтому приведенное выше выражение можно записать таким образом:

```
(Y) / (X) > 0
```

В данном случае операция деления в префиксной форме принимает следующий вид, поскольку в префиксной форме знак операции предшествует параметрам:

```
(/ (Y) (X))
```

Теперь к результату деления необходимо применить операцию проверки, чтобы определить, превышает ли полученное значение `0`. Таким образом, префиксная форма принимает следующий вид:

```
(> (/ (Y) (X)) 0)
```

В инфиксной форме выражение `Y = y2 - y1`. Но мы обязаны и в этом случае использовать префиксную форму, поскольку создается префиксное выражение. Таким образом, вместо выражения `(Y)` необходимо подставить `(- y2 y1)`, а вместо `(X)` ввести `(- x2 x1)`. Заменив `(Y)` и `(X)` их префиксными формами, получим окончательное выражение, предназначенное для проверки того, имеет ли прямая, проходящая через две точки, положительный наклон, как показано ниже.

```
(> (/ (- y2 y1) (- x2 x1)) 0)
```

Простейшим способом вычисления числового выражения (а также любого другого выражения) в языке CLIPS является ввод этого выражения в приглашении верхнего уровня. Например, после ввода выражения `(+ 2 2)` в приглашении CLIPS будет получен следующий вывод:

```
CLIPS> (+ 2 2)↵
4
CLIPS>
```

В этом выводе показан правильный ответ — `4`. Вообще говоря, в приглашении верхнего уровня может быть введено любое выражение CLIPS, подлежащее вычислению. Большинство функций (таких как функция сложения) имеют **возвращаемое значение**. Возвращаемым значением может быть целое число, число с плавающей точкой, символ, строка или даже многозначное значение. Другие функции (такие как команды `facts` и `agenda`) не имеют возвращаемого значения. Но функции, не имеющие возвращаемого значения, обычно создают так называемый **побочный эффект**. Например, побочным эффектом применения команды `facts` является вывод на внешнее устройство фактов из списка фактов.

В приглашении верхнего уровня можно также вводить другие арифметические функции. Ниже приведены результаты, полученные при вычислении некоторых других выражений.

```
CLIPS> (+ 2 3)↵
5
CLIPS> (- 2 3)↵
-1
CLIPS> (* 2 3)↵
6
CLIPS> (/ 2 3)↵
0.66666667
CLIPS>
```

Обратите внимание на то, что в ответе, полученном после выполнения операции деления, вполне может обнаруживаться ошибка округления в последней цифре. Но приведенный выше результат может измениться в зависимости от применяемой операционной системы.

Если всеми параметрами функций +, – и * являются целые числа, то возвращаемое значение также является целочисленным. Если же одним из параметров функции является число с плавающей точкой, то и возвращаемое значение представляет собой число с плавающей точкой. Первый параметр функции / всегда преобразуется в число с плавающей точкой, поэтому возвращаемое значение этой функции всегда является числом с плавающей точкой, например, как показано ниже.

```
CLIPS> (+ 2 3.0)↵
5.0
CLIPS> (+ 2.0 3)↵
5.0
CLIPS> (+ 2 3)↵
5
CLIPS>
```

Переменное количество параметров

Префиксная система обозначений позволяет очень просто представить переменное количество параметров, причем переменное количество параметров принимают многие функции языка CLIPS. Параметры числового выражения в количестве больше двух могут быть указаны после функций +, –, / и *. К параметрам, количество которых превышает два, применяется одна и та же последовательность арифметических вычислений. Следующие примеры, введенные в приглашении верхнего уровня, показывают, как используются три параметра (вычисление осуществляется слева направо):

```
CLIPS> (+ 2 3 4)↵
9
CLIPS> (- 2 3 4)↵
-5
CLIPS> (* 2 3 4)↵
24
CLIPS> (/ 2 3 4)↵
0.16666667
CLIPS>
```

Еще раз отметим, что результаты выполнения операции деления немного зависят от используемой операционной системы.

Определение приоритета и уровня вложенности выражений

Одна из важных особенностей, касающихся вычислений в программах на языке CLIPS или LISP, состоит в том, что арифметические операции не имеют встроенных свойств предшествования. В других языках программирования считается, что операции умножения и деления имеют более высокий приоритет, чем сложение и вычитание, а на компьютере в первую очередь должны выполняться операции с более высоким приоритетом. С другой стороны, в языках LISP и CLIPS все выражения просто вычисляются слева направо, причем порядок вычисления определяется порядком вложенности круглых скобок.

В префиксной системе обозначений могут также выполняться вычисления, состоящие из разных операций. Например, предположим, что должно быть вычислено следующее инфиксное выражение:

```
2 + 3 * 4
```

Общепринятый подход к вычислению этого выражения состоит в том, что 3 следует умножить на 4, а затем сложить результат с 2. Но в языке CLIPS порядок вычислений должен быть задан явно. Приведенное выше выражение вычисляется путем ввода в приглашении верхнего уровня следующей строки:

```
CLIPS> (+ 2 (* 3 4))↵
14
CLIPS>
```

В этом правиле в первую очередь вычисляется выражение, заданное в самых внутренних круглых скобках; таким образом, 3 умножается на 4. Затем полученный результат складывается с 2. Если же требуется вычислить выражение `(2 + 3) * 4`, то вначале выполняется сложение, поэтому в приглашении верхнего уровня следует ввести такое выражение:

```
CLIPS> (* (+ 2 3) 4)↵
20
CLIPS>
```

Вообще говоря, выражения можно свободно вкладывать в другие выражения. Это означает, что выражение может быть помещено и в команду `assert`, как показывает следующий пример:

```
CLIPS> (clear)↵
CLIPS> (assert (answer (+ 2 2)))↵
<Fact-0>
CLIPS> (facts)↵
f-0   (answer 4)
For a total of 1 fact.
```

```
CLIPS>
```

Кроме того, поскольку имена функций также представляют собой символы, то их можно использовать наравне с любыми другими символами. Например, их можно применять в качестве полей в факте, как показано ниже.

```
CLIPS> (clear)↵
CLIPS> (assert (expression 2 + 3 * 4))↵
<Fact-0>
CLIPS> (facts)↵
f-0   (expression 2 + 3 * 4)
For a total of 1 fact.
CLIPS>
```

С другой стороны, круглые скобки используется в языке CLIPS как разграничители, поэтому возможность применять их наравне с другими символами исключена. Для того чтобы использовать круглые скобки в фактах или передать в качестве параметров в функцию, их следует заключить в кавычки, чтобы преобразовать в строки.

8.4 Суммирование значений с использованием правил

В качестве простого примера использования функций для выполнения вычислений рассмотрим задачу нахождения суммы площадей группы прямоугольников. Значение высоты и ширины прямоугольника могут быть заданы с помощью следующей конструкции `deftemplate`:

```
(deftemplate rectangle (slot height) (slot width))
```

Сумму площадей прямоугольников можно определить с использованием примерно такого упорядоченного факта:

```
(sum 20)
```

Ниже приведена конструкция `deffacts`, содержащая информацию, которая рассматривается в качестве примера.

```
(deffacts initial-information
   (rectangle (height 10) (width 6))
   (rectangle (height 7) (width 5))
   (rectangle (height 6) (width 8))
   (rectangle (height 2) (width 5))
   (sum 0))
```

Первоначальная попытка подготовить правило суммирования площадей прямоугольников может состоять в следующем:

```
(defrule sum-rectangles
   (rectangle (height ?height) (width ?width))
   ?sum <- (sum ?total)
   =>
   (retract ?sum)
   (assert (sum (+ ?total (* ?height ?width)))))
```

Но применение указанного правила приведет к созданию бесконечного цикла. Извлечение факта `sum`, а затем повторное его введение в список фактов приводит к возникновению цикла, в котором обрабатывается один и тот же факт `rectangle`. Одним из решений, позволяющих устранить эту проблему, могло бы быть извлечение факта `rectangle` после того, как площадь соответствующего прямоугольника будет добавлена к сумме, представленной фактом `sum`. Таким образом, должна быть исключена возможность запуска правила применительно к тому же факту `rectangle`, но к другому факту `sum`. Тем не менее, если факт `rectangle` должен быть сохранен в списке фактов, то потребуется другой подход. В частности, для каждого факта `rectangle` может создаваться временный факт, содержащий значение площади, которое должно складываться с общей суммой. После этого данный временный факт может быть удален и тем самым предотвращен бесконечный цикл. Программа, модифицированная таким образом, приведена ниже.

```
(defrule sum-rectangles
   (rectangle (height ?height) (width ?width))
   =>
   (assert (add-to-sum (* ?height ?width))))
(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (assert (sum (+ ?total ?area))))
```

Ниже приведен вывод, который показывает, как взаимодействуют эти два правила в процессе суммирования площадей прямоугольников.

```
CLIPS> (unwatch all)↵
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (watch activations)↵
CLIPS> (reset)↵
==> f-0     (initial-fact)
==> f-1     (rectangle (height 10) (width 6))
==> Activation 0     sum-rectangles: f-1
```

```
==> f-2     (rectangle (height 7) (width 5))
==> Activation 0     sum-rectangles: f-2
==> f-3     (rectangle (height 6) (width 8))
==> Activation 0     sum-rectangles: f-3
==> f-4     (rectangle (height 2) (width 5))
==> Activation 0     sum-rectangles: f-4
==> f-5     (sum 0)
CLIPS>
(run)↵
FIRE     1 sum-rectangles: f-4
==> f-6     (add-to-sum 10)
==> Activation 0        sum-areas: f-5,f-6
FIRE     2 sum-areas: f-5,f-6
<== f-5     (sum 0)
<== f-6     (add-to-sum 10)
==> f-7     (sum 10)
FIRE     3 sum-rectangles: f-3
==> f-8     (add-to-sum 48)
==> Activation 0        sum-areas: f-7,f-8
FIRE     4 sum-areas: f-7,f-8
<== f-7     (sum 10)
<== f-8     (add-to-sum 48)
==> f-9     (sum 58)
FIRE     5 sum-rectangles: f-2
==> f-10    (add-to-sum 35)
==> Activation 0        sum-areas: f-9,f-10
FIRE     6 sum-areas: f-9,f-10
<== f-9     (sum 58)
<== f-10    (add-to-sum 35)
==> f-11    (sum 93)
FIRE     7 sum-rectangles: f-1
==> f-12    (add-to-sum 60)
==> Activation 0        sum-areas: f-11,f-12
FIRE     8 sum-areas: f-11,f-12
<== f-11    (sum 93)
<== f-12    (add-to-sum 60)
==> f-13    (sum 153)
CLIPS> (unwatch all)↵
CLIPS>
```

Правило sum-rectangles активизируется четыре раза, в то время как происходит вставка фактов rectangle в список фактов в результате выполнения команды reset. Правило sum-rectangles при каждом запуске обеспечивает ввод в список фактов такого факта, который активизирует правило sum-areas. Правило sum-areas обеспечивает добавление текущего значения площади к промежуточной сумме и удаление факта add-to-sum. Поскольку шаблон правила sum-rectangles не сопоставляется с фактом sum, после ввода в список фактов нового факта sum повторная активизация этого правила не происходит.

8.5 Функция bind

Часто возникает необходимость сохранить некоторое значение во временной переменной, чтобы избежать повторного вычисления. Это особенно важно, если используются функции, которые производят побочные эффекты. Для связывания значения переменной со значением некоторого выражения может применяться **функция bind**. Функция bind имеет следующий синтаксис:

```
(bind <variable> <value>)
```

В этом определении для задания связанной переменной <variable> используется синтаксис однозначной переменной, в качестве нового значения <value> должно служить выражение, вычисление которого приводит к получению либо однозначного, либо многозначного значения. Например, ниже приведено правило sum-areas, позволяющее выводить на внешнее устройство общую сумму и площади каждого прямоугольника, из которых складывалась эта сумма.

```
(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (printout t "Adding " ?area " to " ?total crlf)
   (printout t "New sum is " (+ ?total ?area) crlf)
   (assert (sum (+ ?total ?area))))
```

Обратите внимание на то, что в правой части этого правила выражение (+ ?total ?area) использовалось дважды. А замена двух отдельных вычислений с помощью одной функции bind позволяет устранить ненужные вычисления. Ниже приведено то же правило, но сформулированное иначе благодаря использованию функции bind.

```
(defrule sum-areas
   ?sum <- (sum ?total)
```

```
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (printout t "Adding " ?area " to " ?total crlf)
   (bind ?new-total (+ ?total ?area))
   (printout t "New sum is " ?new-total crlf)
   (assert (sum ?new-total)))
```

Функция bind может применяться не только для создания новых переменных, предназначенных для включения в правую часть какого-либо правила, но и для повторного связывания со значением такой переменной, которая встречается в левой части некоторого правила.

8.6 Функции ввода-вывода

Функция read

В процессе работы экспертных систем часто возникает необходимость в том, чтобы пользователь ввел в программу некоторые данные. Система CLIPS позволяет считывать информацию с клавиатуры с использованием **функции read**. Основной синтаксис функции read не требует параметров, а следующий пример показывает, как используется эта функция для ввода данных:

```
CLIPS> (clear)↵
CLIPS> (defrule get-first-name
   =>
   (printout t "What is your first name? ")
   (bind ?response (read))
   (assert (user's-name ?response)))↵
CLIPS> (reset)↵
CLIPS> (run)↵
What is your first name? Gary↵
CLIPS> (facts)↵
f-0      (initial-fact)
f-1      (user's-name Gary)
For a total of 2 facts.
CLIPS>
```

Следует отметить, что функция read может считать введенную лексему только после нажатия клавиши ввода. Функция read может использоваться для ввода одновременно только одного поля. Все дополнительные знаки, введенные после ввода первого поля, вплоть до обозначения конца строки, сформированного путем нажатия клавиши ввода, отбрасываются. Например, если с помощью правила

get-first-name будет предпринята попытка прочитать с помощью следующего ввода не только имя, но и фамилию, то будет введено лишь первое поле, Gary:

```
Gary Riley↵
```

Чтобы обеспечить чтение всего ввода, необходимо заключить оба поля в двойные кавычки. Безусловно, после обозначения вводимых данных с помощью двойных кавычек эти данные преобразуются в единственное литеральное поле, поэтому исключается возможность легко получить доступ к отдельным полям, Gary и Riley.

Функция read позволяет также вводить поля, не являющиеся символами, строками, целочисленными значениями или числовыми значениями с плавающей точкой. В частности, с ее помощью можно вводить знаки круглых скобок. Такие поля обозначаются двойными кавычками и рассматриваются как строки. Указанная возможность демонстрируется с помощью следующего диалога в командной строке:

```
CLIPS> (read)↵
(↵
"("
CLIPS>
```

Функция open

Язык CLIPS обеспечивает не только ввод с клавиатуры и вывод на терминал, но и позволяет читать данные из файлов и записывать в файлы. Но прежде чем появится возможность получить доступ к файлу для чтения или записи, файл необходимо открыть с помощью **функции open**. Количество файлов, которые могут быть открыты одновременно, зависит от конкретной операционной системы и применяемых в ней аппаратных средств.

Функция open имеет следующий синтаксис:

```
(open <file-name> <file-ID> [<file-access>])
```

В качестве примера рассмотрим такую команду:

```
(open "input.dat" data "r")
```

Первый параметр команды open, <file-name>, — это строка, представляющая имя файла на локальном компьютере. В данном примере используется файл с именем “input.dat”. Имя файла может также включать информацию о пути (о каталоге, в котором находится файл). Формат спецификации пути, как правило, зависит от операционной системы, поэтому, чтобы правильно задать путь, необходимо иметь определенное представление об особенностях операционной системы применяемого компьютера.

Второй параметр, <file-ID>, представляет собой **логическое имя**, которое должно быть связано в системе CLIPS с данным файлом. Логическое имя — это глобальное имя, с помощью которого обеспечивается возможность получить доступ к файлу в системе CLIPS из любого правила или приглашения верхнего уровня. Безусловно, логическое имя может совпадать с именем файла, но рекомендуется использовать другое имя для предотвращения путаницы. В рассматриваемом примере для ссылки на файл "input.dat" служит логическое имя data. Могут также применяться другие осмысленные имена, такие как input или file-data.

Одним из преимуществ использования логического имени является то, что оно позволяет легко заменить имя файла другим именем файла, не внося в программу существенные изменения. Дело в том, что имя файла используется только в функции open, а в дальнейшем ссылка на файл осуществляется только по его логическому имени, поэтому для обеспечения чтения из другого файла достаточно внести изменения лишь в вызов функции open.

Третий параметр, <file-access>, — это строка, представляющая один из четырех возможных режимов доступа к файлу. Режимы доступа к файлу описаны в табл. 8.2.

Таблица 8.2. Режимы доступа к файлу

Режим	Действие
"r"	Доступ только для чтения
"w"	Доступ только для записи
"r+"	Доступ для чтения и записи
"a"	Доступ только для дополнения

Если в качестве параметра не задано значение режима доступа к файлу <file-access>, используется заданное по умолчанию значение "r". В некоторых операционных системах те или иные режимы доступа могут оказаться неприменимыми. Но большинство операционных систем поддерживает доступ для чтения (обеспечивающий ввод данных) и доступ для записи (обеспечивающий вывод данных). Доступ для чтения или записи, а также доступ для дополнения (позволяющий добавить выводимые данные к концу файла) может не всегда оказаться применимым.

Важно помнить, что в разных операционных системах открытие файла может приводить к разным последствиям. Например, в операционных системах, не поддерживающие возможность создания многочисленных версий файла, таких как MS-DOS (на персональном компьютере IBM) или Unix, при использовании режима доступа для записи при открытии файла заменяют существующий файл вновь созданным файлом. В отличие от этого, при открытии файла с применением

режима доступа для записи в операционной системе VMS на компьютере VAX создается новая версия файла, если файл уже существует, а старая версия файла не уничтожается.

Функции `open` действует как предикативная функция (определение понятия предикативной функции приведено в разделе 8.8). Эта функция возвращает символ `TRUE`, если файл был открыт успешно; в противном случае возвращается символ `FALSE`. Возвращаемое значение может использоваться для проверки на наличие ошибок. Например, если пользователь задал имя файла, который не существует, и была предпринята попытка открыть файл с применением режима доступа для чтения, то функция `open` возвращает символ `FALSE`. Такой результат может быть проверен в правиле, где происходит открытие файла, после чего предприняты соответствующие действия, например, передано приглашение пользователю для ввода другого имени файла.

Функция `close`

Если доступ к файлу больше не требуется, файл должен быть закрыт. Если такой файл не будет закрыт, то нет никакой гарантии, что записанная в него информация сохранится. Более того, чем дольше файл остается открытым, тем больше вероятность того, что из-за отключения питания или другого нарушения в работе не удастся сохранить записанную информацию.

Функция close имеет следующую общую форму, в которой необязательный параметр `<file-ID>` задет логическое имя закрываемого файла:

```
(close [<file-ID>])
```

Если параметр `<file-ID>` не задан, закрываются все открытые файлы. В качестве примера можно указать, что следующая команда закрывает файл, известный системе CLIPS под логическим именем `data`:

```
(close data)
```

А приведенные ниже операторы закрывают файлы, связанные с логическими именами `input` и `output`.

```
(close input)
(close output)
```

Следует отметить, что для закрытия только конкретных файлов необходимо применять отдельные операторы.

При использовании файлов важно помнить, что каждый открытый файл должен быть в конечном итоге закрыт с помощью функции `close`. Особенно важно то, что при отсутствии команды закрытия файла могут быть потеряны записанные в него данные. Система CLIPS не выводит приглашение для пользователя, чтобы он закрыл открытый файл. Единственная мера предосторожности, встроенная в систему CLIPS и касающаяся закрытия файлов, которые непреднамеренно были

оставлены открытыми, состоит в том, что все открытые файлы закрываются после вызова на выполнение команды `exit`.

Чтение из файла и запись в файл

В примерах, которые рассматривались до сих пор в данной главе, все входные данные считывались с клавиатуры, а все выходные данные записывались на терминал. С другой стороны, с применением логических имен обеспечивается ввод и вывод данных с использованием других источников данных. В главе 7 было показано, что функция `printout`, в вызове которой задано логическое имя `t`, позволяет передавать выходные данные на экран. В функции `printout` могут также применяться другие логические имена для передачи выходных данных в место назначения, отличное от экрана.

При использовании логического имени `t` в качестве параметра с обозначением логического имени в любой функции вывода происходит запись выходных данных в стандартное устройство вывода, обычно на терминал. Аналогичным образом, при использовании логического имени `t` в качестве параметра с обозначением логического имени в любой функции ввода осуществляется ввод данных из **стандартного устройства ввода данных**, обычно с клавиатуры.

Ниже приведен пример, который показывает, как применяются логические имена для записи в файл.

```
CLIPS> (open "example.dat" example "w")↵
TRUE
CLIPS> (printout example "green" crlf)↵
CLIPS> (printout example 7 crlf)↵
CLIPS>
(close example)↵
TRUE
CLIPS>
```

Вначале осуществляется открытие файла “`example.dat`” в режиме доступа для записи, чтобы обеспечить запись в него значений данных. После выполнения функции `open` логическое имя `example` становится связанным с файлом “`example.dat`”. Затем в файл “`example.dat`” записываются значения `green` и 7 с помощью логического имени `example` в качестве первого параметра функции `printout`. А после записи значений в файл последний закрывается с помощью команды `close`.

Теперь, после записи указанных значений в файл, этот файл можно открыть в режиме доступа для чтения и осуществить выборку тех же значений с помощью функции `read`. Функция `read` имеет следующий общий формат:

```
(read [<logical-name>])
```

По умолчанию функция read, если в ней не заданы параметры вызова, осуществляет чтение из стандартного устройства ввода данных, t. Это же логическое имя использовалось по умолчанию и в предыдущем примере вызова функции read. А в следующем примере показано, как можно применить логическое имя в функции read для выборки значений из файла "example.dat":

```
CLIPS> (open "example.dat" example "r")↵
TRUE
CLIPS> (read example)↵
green
CLIPS> (read example)↵
7
CLIPS>
(read example)↵
EOF
CLIPS>
(close example)↵
TRUE
CLIPS>
```

Вначале происходит открытие файла "example.dat", но на этот раз в режиме доступа для чтения. Обратите внимание на то, что при открытии файла "example.dat" во втором вызове функции open не требовалось явно задавать опцию "r", поскольку она применяется по умолчанию. После открытия файла для выборки значений green и 7 из файла, связанного с логическим именем example, использовалась функция read. Обратите внимание на то, что после третьего вызова функции read возвращен **символ EOF**. Система CLIPS возвращает это значение после вызова функций ввода данных, если предпринимаются попытки чтения вслед за концом файла. Проверяя это возвращаемое значение функции read (или другой функции ввода данных), можно определить тот момент, когда в файле уже не остается данных.

Функция format

Иногда возникает необходимость отформатировать вывод программы CLIPS, например, при оформлении данные в виде таблиц. Безусловно, для этой цели можно воспользоваться функцией printout, но имеется отдельная функция, специально предназначенная для форматирования и называемая **format**, которая предоставляет возможность воспользоваться широким разнообразием стилей форматирования. Функция format имеет следующий синтаксис:

```
(format <logical-name> <control-string>
        <parameters>*)
```

Вызов функции format состоит из нескольких частей. Параметр <logical-name> представляет собой логическое имя файла, в который будут передаваться выходные данные. С помощью логического имени t может быть указано применяемое по умолчанию стандартное устройство вывода. За этим параметром следует параметр с **управляющей строкой**; эта строка должна быть заключена в двойные кавычки. Управляющая строка состоит из **флажков формата**, которые показывают, как должен осуществляться вывод данных, обозначенных параметрами функции format. Вслед за управляющей строкой находится список параметров. Количество флажков формата в управляющей строке определяет количество задаваемых параметров. В качестве параметров могут применяться либо значения констант, либо выражения. Возвращаемым значением функции format является отформатированная строка. Если в вызове команды format используется логическое имя nil, то вывод каких-либо выходных данных не производится (ни на терминал, ни в файл), но возврат отформатированной строки все равно выполняется.

Пример применения функции format показан в приведенном ниже диалоге, в котором создается отформатированная строка, содержащая имя некоторого лица (причем для этого имени резервируются 15 пробелов), а за этим именем следуют данные о возрасте этого лица. В данном примере заслуживает внимания то, как значения имени и возраста выравниваются в своих столбцах. Таким образом функция format является удобным средством размещения данных по столбцам

```
CLIPS> (format nil "Name: %-15s Age: %3d"
                   "Bob Green" 35)↵
"Name: Bob Green      Age: 35"
CLIPS> (format nil "Name: %-15s Age: %3d"
                   "Ralph Heiden" 32)↵
"Name: Ralph Heiden   Age: 32"
CLIPS>
```

Флажки формата всегда начинаются со знака процента, %. В управляющую строку можно также помещать обычные строковые данные, такие как “Name:”, для последующего вывода. Но некоторые флажки формата не форматируют параметры. Например, флажок “%n” используется для вывода знаков возврата каретки и (или) перевода строки, аналогично тому, как в команде printout применяется символ crlf.

В данном примере для вывода имени в столбец, имеющий ширину 15 знаков, используется флажок формата “%-15s”. Знак - показывает, что вывод должен быть выровнен влево, а буква s говорит о том, что должна быть выведена строка или символ. Флажок формата “%3d” показывает, что число должно быть выведено как целочисленное значение, выровненное вправо в столбце шириной три знака. А если бы в качестве параметра для этого флажка формата было передано значение

5.25, то программа вывела бы число 5, поскольку в целочисленном формате не допускается вывод дробной части числа.

Следует отметить, что при использовании функции format в правой части правила ее возвращаемое значение, как правило, игнорируется. В этих случаях либо логическое имя должно быть связано с файлом, либо должно применяться логическое имя t для передачи выходных данных на экран. Флажок формата имеет следующую общую спецификацию, в которой знак "-" является необязательным и означает, что должно быть выполнено **выравнивание влево**:

```
%-M.Nx
```

По умолчанию применяется **выравнивание вправо**. Выравнивание происходит, если количество знаковых позиций, предусмотренных для вывода значения, превышает количество знаков, необходимых для вывода этого значения. Если возникает такая ситуация, то выравнивание значения влево вызывает вывод указанного значения в левой части отведенного для него места, при условии, что неиспользуемое пространство в правой части заполняется пробелами. Применение выравнивания вправо приводит к тому, что значение выводится в правой части отведенного для него места, а пробелами заполняется неиспользуемое пространство слева.

Буквой M в этой спецификации обозначается ширина поля в знаках. Если значение M задано, то выводится по меньшей мере такое количество знаков, которое задано числом M. Обычно для заполнения неиспользуемой части пространства вывода, состоящего из M знаков, применяются пробелы, если число M не начинается с цифры 0; в последнем случае для заполнения служат ноли. Если выводимое значение превышает по ширине значение M, то функция format расширяет поле вывода настолько, насколько потребуется.

Буквой N в спецификации формата задается необязательное число, определяющее количество цифр после десятичной точки, которое должно быть выведено. По умолчанию при выводе чисел с плавающей точкой после десятичной точки выводится шесть цифр.

Буква x в спецификации представляет собой буквенное значение, определяющее спецификацию формата отображения. Возможные значения спецификации формата отображения показаны в табл. 8.3.

Функция readline

Функция readline может использоваться для чтения полной строки ввода и имеет следующий синтаксис:

```
(readline [<logical-name>])
```

Как и в функции read, параметр <logical-name> с обозначением логического имени является необязательным. Если логическое имя не задано или

Таблица 8.3. Спецификации формата отображения

Знак	Значение
d	Целое число
f	Число с плавающей точкой
e	Число в экспоненциальном формате (в котором задаются значащая часть числа и его порядок — степень числа десять)
g	Общий (числовой) формат; допускается отображение в любом формате, требующем меньшего количества знаков
o	Восьмеричный; число без знака (применение спецификатора N не допускается)
x	Шестнадцатеричный формат; число без знака (применение спецификатора N не допускается)
s	Строка; перед преобразованием строк, заключенных в кавычки, во внутреннее представление открывающая и закрывающая кавычки удаляются
n	Знаки возврата каретки и (или) перевода строки
%	Знак % в своем непосредственном виде

используется логическое имя t, то входные данные считываются со стандартного устройства ввода данных. Функция readline возвращает в качестве строкового значения следующую строку ввода, полученную из источника входных данных, связанного с указанным логическим именем; строка содержит все введенные данные, вплоть до обозначения конца строки, и включая это обозначение. А если достигнут конец файла, функция readline возвращает символ EOF. Такая ситуация может возникнуть только в том случае, если логическое имя, применяемое в функции readline, связано с файлом. Приведенный ниже диалог иллюстрирует использование функции readline.

```
CLIPS> (clear)↵
CLIPS>
(defrule get-name
   =>
   (printout t "What is your name? ")
   (bind ?response (readline))
   (assert (user's-name ?response)))
CLIPS> (reset)↵
CLIPS> (run)↵
What is your name? Gary Riley↵
CLIPS>
(facts)↵
f-0     (initial-fact)
```

```
f-1     (user's-name "Gary Riley")
For a total of 2 facts.
CLIPS>
```

В этом примере имя "Gary Riley" сохраняется в факте `user's-name` как единственное поле. А поскольку эти данные хранятся в виде единственного поля, то с использованием переменных шаблона невозможно непосредственно осуществить выборку из этого поля данных об имени и фамилии. Но с помощью **функции explode$**, которая принимает единственный строковый параметр и преобразует его в многозначное значение, мы можем трансформировать строку, возвращаемую функцией `readline`, в многозначное значение, которое должно быть представлено в виде ряда полей в факте `user's-name`. Применение функции `readline` в сочетании с функцией `explode$` демонстрируется в следующем диалоге:

```
CLIPS>
(defrule get-name
   =>
   (printout t "What is your name? ")
   (bind ?response (explode$ (readline)))
   (assert (user's-name ?response)))↵
CLIPS> (reset)↵
CLIPS> (run)↵
What is your name? Gary Riley↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (user's-name Gary Riley)
For a total of 2 facts.
CLIPS>$
```

Перечень других функций, которые могут применяться для создания и манипулирования строками и многозначными значениями, приведен в приложении Д. Дополнительные сведения обо всех функциях, доступных в языке CLIPS, можно найти в документе *Basic Programming Guide* на компакт-диске, прилагаемом к данной книге.

8.7 Игра в палочки

В следующих нескольких разделах для демонстрации различных методов управления процессом выполнения программы на языке, основанном на правилах, будет использоваться простая игра с двумя игроками, называемая Sticks (Игра в палочки). Цель игры Sticks состоит в том, чтобы избежать необходимости взять последнюю палочку из кучи палочек. Каждый игрок должен брать из кучи по

очереди 1, 2 или 3 палочки. Весь секрет выигрыша в этой игре (или эвристика) состоит в том, что можно вынудить противника проиграть, если при вашей очереди хода останутся 2, 3 или 4 палочки, т.е. необходимо к этому стремиться. Таким образом, игрок, при ходе которого осталось 5 палочек, проиграл. Чтобы вынудить другого игрока столкнуться с ситуацией, в которой для него остается 5 палочек, следует всегда оставлять после своего хода 5 палочек плюс некоторое количество палочек, кратное 4. Иными словами, следует всегда стремиться к тому, чтобы после вашего хода количество оставшихся в куче палочек было равно 5, 9, 13 и т.д. Если же вы ходите первым и количество палочек в куче равно одному из этих "безвыигрышных" чисел, то вы не сможете выиграть, при условии безошибочной игры противника. С другой стороны, если ваш ход — первый, а количество палочек в куче не соответствует одному из "безвыигрышных" чисел, то вы всегда можете рассчитывать на победу.

Прежде чем программа сможет приступить к ведению игры Sticks, она должна определить некоторую информацию. Прежде всего программа должна играть против противника-человека, поэтому необходимо в первую очередь выяснить, кто ходит первым. Кроме того, следует определить начальный размер кучи. Эта информация может быть помещена в конструкцию `deffacts`. Но весьма несложно также запросить такую информацию у того, кто собирается состязаться с программой, чтобы он ввел необходимые сведения с клавиатуры. Следующий пример показывает, как применяется функция `read` для ввода данных:

```
(deffacts initial-phase
   (phase choose-player))
(defrule player-select
   (phase choose-player)
   =>
   (printout t "Who moves first (Computer: c "
               "Human: h)? ")
   (assert (player-select (read))))
(defrule good-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&c | h)
   =>
   (retract ?phase ?choice)
   (assert (player-move ?player)))
```

В обоих правилах используется шаблон `(phase choose-player)` для указания на то, что эти правила применимы только при наличии конкретного факта в списке фактов. Такой шаблон называется **управляющим шаблоном**, поскольку он специально предназначен для управления тем, является ли правило применимым или нет. А для запуска управляющего шаблона используется **управляющий**

факт. В управляющем шаблоне для этих правил используются только литеральные поля, поэтому управляющий факт должен точно согласовываться с шаблоном. В рассматриваемом случае управляющим фактом, применяемым для запуска этих правил, должен быть факт `(phase choose-player)`. Кроме всего прочего, такой управляющий факт позволяет исправить ошибку, если входные данные, полученные с помощью функции `read`, не совпадают с ожидаемыми значениями, "c" или "h" (сокращения от "computer" — компьютер и "human" — человек). Повторную активизацию правила `player-select` можно обеспечить, извлекая и вновь вводя в список фактов этот управляющий факт.

Ниже приведен вывод, который показывает, как работают правила `player-select` и `good-player-choice` при определении того, кто должен ходить первым.

```
CLIPS> (unwatch all)↵
CLIPS> (watch facts)↵
CLIPS> (reset)↵
==> f-0     (initial-fact)
==> f-1     (phase choose-player)
CLIPS> (run)↵
Who moves first (Computer: c Human: h)? c↵
==> f-2     (player-select c)
<== f-1     (phase choose-player)
<== f-2     (player-select c)
==> f-3     (player-move c)
CLIPS>
```

Эти правила функционируют должным образом, если вводится предусмотренный в них ответ `c` или `h`, но если введенный ответ отличается от требуемого, то проверка на наличие ошибок не осуществляется. Это — один из примеров многочисленных ситуаций, в которых необходимо повторить запрос на ввод, чтобы исправить ошибку при вводе. В приведенном ниже коде показан удобный способ реализации в программе циклического повтора запроса на ввод. Правило, приведенное в этом коде, при использовании в сочетании с правилами `player-select` и `good-player-choice` обеспечивает проверку на наличие ошибок.

```
(defrule bad-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&~c&~h)
   =>
   (retract ?phase ?choice)
   (assert (phase choose-player))
   (printout t "Choose c or h." crlf))
```

И в этом случае следует отметить, что используется управляющий шаблон (phase choose-player). Он обеспечивает общее управление над циклом ввода, а также предотвращает запуск этой группы правил при осуществлении других этапов выполнения программы.

Здесь заслуживает внимания то, как действуют совместно два правила, player-select и bad-player-choice, а именно, как каждое правило предоставляет факты, необходимые для активизации другого правила. Если ответ на вопрос о выборе игрока, начинающего игру, является недопустимым, то с помощью правила bad-player-choice извлекается управляющий факт (phase choose-player), после чего этот факт снова вводится, что вызывает повторную активизацию правила player-select.

8.8 Предикативные функции

По определению как **предикативная функция** может рассматриваться любая функция, возвращающая либо символ TRUE, либо символ FALSE. По существу при выполнении любых операций в рамках предикативной логики в языке CLIPS любое значение, отличное от символа FALSE, рассматривается как символ TRUE. Можно также считать, что предикативной функцией является функция, имеющая булево возвращаемое значение. Предикативные функции подразделяются на две категории: **заранее определенные функции** и **определяемые пользователем функции**. Заранее определенными функциями называются функции, которые уже предусмотрены в языке CLIPS. С другой стороны, определяемыми пользователем или **внешними функциями**, являются функции, отличные от заранее определенных, которые написаны на языке C или на другом языке, а код реализации этих функций связан с интерпретатором CLIPS. В приложении Д содержится список заранее определенных предикативных функций CLIPS, предназначенных для выполнения булевых логических операций, операций сравнения значений и операций проверки на наличие конкретного типа. Примеры применения некоторых из этих функций показаны в следующем диалоге:

```
CLIPS> (and (> 4 3) (> 4 5))↵
FALSE
CLIPS> (or (> 4 3) (> 4 5))↵
TRUE
CLIPS> (> 4 3)↵
TRUE
CLIPS> (< 6 2)↵
FALSE
CLIPS> (integerp 3)↵
TRUE
```

```
CLIPS> (integerp 3.5)↵
FALSE
CLIPS>
```

8.9 Условный элемент `test`

Часто встречаются такие ситуации, в которых было бы целесообразно повторить вычисления или еще раз выполнить какую-либо операцию обработки информации. Общепринятым способом решения такой задачи является создание цикла. В предыдущем примере цикл создавался для повторного вывода вопроса, который появлялся на экране до тех пор, пока от пользователя не поступал правильный ответ. Но во многих других ситуациях цикл должен завершаться автоматически, без участия человека, в результате вычисления какого-то произвольного выражения.

В основе одного из мощных способов вычисления выражений в левой части правила может лежать **условный элемент test**. В частности, условный элемент `test` не требует сопоставления шаблона с одним из фактов в списке фактов, а просто вычисляет выражение. Но самой внешней функцией этого выражения должна быть предикативная функция. Если вычисление выражения приводит к получению любого значения, отличного от символа `FALSE`, то проверка с помощью условного элемента `test` завершается успешно. Если же вычисление выражения приводит к получению символа `FALSE`, то проверка с помощью условного элемента `test` оканчивается неудачей. Запуск правила происходит только в том случае, если проверка всех его условных элементов `test`, наряду со всеми другими шаблонами, завершается успешно. Условный элемент `test` имеет следующий синтаксис:

```
(test <predicate-function>)
```

В качестве примера предположим, что очередность хода принадлежит игроку-человеку. Если в куче осталась только одна палочка, то человек проиграл. А если палочек больше одной, то компьютер должен спросить человека, сколько палочек следует убрать из кучи. В том правиле, в котором формулируется вопрос к человеку о том, сколько палочек следует убрать из кучи, необходимо предусмотреть проверку, что количество палочек в куче больше одной. Как показано в следующем условном элементе `test`, для представления этого ограничения может использоваться предикативная функция `>`; в рассматриваемом выражении переменная `?size` обозначает количество палочек, оставшихся в куче:

```
(test (> ?size 1))
```

После того как человек укажет количество палочек, подлежащих удалению, его ответ необходимо проверить, чтобы убедиться в том, что он является допустимым. Количество удаляемых палочек должно быть выражено целым числом,

а также должно быть больше или равно одному и меньше или равно трем. Кроме того, человек не может взять больше палочек, чем имеется в куче, и его необходимо вынудить взять последнюю палочку. Как показано в следующем условном элементе test, в котором переменная ?choice будет содержать количество взятых из кучи палочек, а переменная ?size представляет собой количество палочек, оставшихся в куче, предикативная функция and может использоваться для выражения всех этих ограничений:

```
(test (and (integerp ?choice)
   (>= ?choice 1)
   (<= ?choice 3)
   (< ?choice ?size)))
```

Аналогичным образом, как неправильно заданное количество палочек, которые должны быть взяты из кучи, рассматривается значение, не являющееся целочисленным, меньшее единицы или большее трех, а также большее или равное количеству оставшихся палочек (если количество палочек больше одной). Эти условия можно представить с помощью предикативной функции or, как показано в следующем условном элементе test, в котором переменная ?choice будет содержать данные о количестве палочек, которые должны быть взяты из кучи, а переменная ?size представляет количество палочек, оставшихся в куче:

```
(test (or (not (integerp ?choice))
   (< ?choice 1)
   (> ?choice 3)
   (>= ?choice ?size)))
```

Ниже приведены правила, в которых описанные здесь условные элементы test применяются для проверки того, являются ли допустимыми данные о количестве палочек, взятых из кучи игроком-человеком. В этих правилах для хранения информации о количестве палочек, оставшихся в куче, используется факт pile-size.

```
(defrule get-human-move
   (player-move h)
   (pile-size ?size)
   ; У человека-игрока есть выбор, только если в куче
   ; осталось больше одной палочки
   (test (> ?size 1))
   =>
   (printout t
         "How many sticks do you wish to take?")
   (assert (human-takes (read))))
(defrule good-human-move
```

```
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (and (integerp ?choice)
              (>= ?choice 1)
              (<= ?choice 3)
              (< ?choice ?size)))
   =>
   (retract ?whose-turn ?number-taken)
   (printout t "Human made a valid move" crlf))
(defrule bad-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (or (not (integerp ?choice))
             (< ?choice 1)
             (> ?choice 3)
             (>= ?choice ?size)))
=>
   (printout t "Human made an invalid move" crlf)
   (retract ?whose-turn ?number-taken)
   (assert (player-move h)))
```

Выполнение данной программы завершается только после того, как будет введено допустимое значение количества взятых из кучи палочек. И в этом случае для повторной активизации правила, позволяющего еще раз ввести ответ после ввода в предыдущей попытке недопустимого ответа, применяется управляющий факт. Повторный ввод управляющего факта `(player-move h)` для еще одной активизации правила `bad-human-move` осуществляется с помощью правила `get-human-move`. После этого игрок-человек получает возможность еще раз указать количество палочек, которые должны быть взяты из кучи.

8.10 Предикативное ограничение поля

Для выполнения предикативных проверок непосредственно в шаблонах могут использоваться так называемые **предикативные ограничения поля**, выраженные с помощью символа `:`. Применение такого ограничения во многих отношениях аналогично выполнению проверки с помощью условного элемента `test` непосредственно после сопоставления с шаблоном, но в некоторых случаях, как будет показано в главе 9, предикативные ограничения поля являются более эффективными по сравнению с условными элементами `test`. Предикативное ограничение

поля может использоваться полностью аналогично литеральному ограничению поля. Кроме того, предикативное ограничение поля может быть задано в поле отдельно или введено как одна из частей более сложного поля с помощью соединительных ограничений поля ~, & и |. За предикативным ограничением поля всегда следует функция, подлежащая вызову на выполнение. Как и в случае условного элемента test, эта функция должна быть предикативной.

В качестве примера рассмотрим следующие два шаблона, применявшиеся в правиле get-human-move (которое описано в предыдущем разделе), для проверки того, что в куче содержится больше одной палочки:

```
(pile-size ?size)
(test (> ?size 1))
```

Эти два шаблона можно заменить таким одним шаблоном:

```
(pile-size ?size&:(> ?size 1))
```

Предикативное ограничение поля может использоваться отдельно, но ситуации, в которых такая возможность является действительно удобной, встречаются редко. Как правило, осуществляется связывание переменной, а затем проверка с помощью предикативного ограничения поля. При чтении шаблона предикативное ограничение поля можно рассматривать как означающее "такой, что". Например, ограничение поля, приведенное выше:

```
?size&:(> ?size 1)
```

можно прочитать как "выполнить привязку значения ?size, такого, что ?size больше 1".

Одной из областей применения предикативного ограничения поля является проверка наличия ошибок в данных. Например, в следующем правиле выполняется проверка для определения того, что элемент данных является числовым, перед его сложением с промежуточной суммой:

```
(defrule add-sum
   (data-item ?value&:(numberp ?value))
   ?old-total <- (total ?total)
   =>
   (retract ?old-total)
   (assert (total (+ ?total ?value))))
```

Второе поле первого шаблона можно прочитать как "выполнить связывание значения ?value, такого, что ?value является числом". Ниже приведены два правила, в которых выполняется такого же рода проверка наличия ошибок и демонстрируется использование соединительных ограничений поля ~ и | в сочетании с предикативным ограничением поля.

```
(defrule find-data-type-1
   (data ?item&:(stringp ?item)|:(symbolp ?item))
```

```
   =>
   (printout t ?item " is a string or symbol "
   crlf))
(defrule find-data-type-2
   (data ?item&~:(integerp ?item))
   =>
   (printout t ?item " is not an integer " crlf))
```

В правиле `find-data-type-1` проверяется, является ли элемент данных строкой или символом; для этого используются такие предикативные функции, как `stringp` и `symbolp`. Правило `find-data-type-2` обеспечивает проверку того, что элемент данных не является числовым; для этого вызывается на выполнение функция `integerp`, а затем применяется соединительное ограничение `~` для получения логического значения, являющегося отрицанием значения, возвращаемого функцией `integerp`.

8.11 Ограничение поля для возвращаемого значения

Ограничение поля для возвращаемого значения, обозначаемое символом `=`, позволяет использовать для сравнения внутри шаблона возвращаемое значение функции. Ограничение поля для возвращаемого значения может применяться в сочетании с соединительными ограничениями поля `~`, `&` и `|`, а также с предикативным ограничением поля. Как и за предикативным ограничением поля, за ограничением поля для возвращаемого значения должен следовать вызов функции. Но вызываемая функция не обязательно должна быть предикативной функцией. Единственным ограничением является то, что вызываемая функция должна иметь однозначное возвращаемое значение. Ниже приведено правило, которое показывает, как можно использовать ограничение возвращаемого значения в программе Sticks для определения количества палочек, которые игрок-компьютер должен убрать из кучи.

```
(deftemplate take-sticks
   (slot how-many)
   (slot for-remainder))
(deffacts take-sticks-information
   (take-sticks (how-many 1) (for-remainder 1))
   (take-sticks (how-many 1) (for-remainder 2))
   (take-sticks (how-many 2) (for-remainder 3))
   (take-sticks (how-many 3) (for-remainder 0)))
(defrule computer-move
```

```
?whose-turn <- (player-move c)
?pile <- (pile-size ?size)
(test (> ?size 1))
(take-sticks (how-many ?number)
(for-remainder =(mod ?size 4)))
=>
(retract ?whose-turn ?pile)
(assert (pile-size (- ?size ?number)))
(assert (player-move h)))
```

Подходящее количество палочек, которое компьютер должен взять на своем ходе, определяется с помощью правила `computer-move`. Первый условный элемент шаблона гарантирует, что правило применяется, только если ходить должен компьютер. Второй и третий условные элементы выполняют проверку для определения того, измеряется ли количество оставшихся палочек числом больше единицы. Если оказалось, что количество оставшихся палочек составляет только одну палочку, то компьютер будет вынужден взять ее и проиграть. Последний шаблон действует в сочетании с конструкцией `deffacts` с именем `take-sticks-information` для определения подходящего количества палочек, которые должны быть взяты из кучи. **Функция mod** (сокращение от modulus) возвращает целочисленный остаток от деления своего первого параметра на второй. При чтении шаблона целесообразно рассматривать ограничение поля для возвращаемого значения как имеющее смысл “равно”. Например, ограничение поля

```
=(mod ?size 4)
```

можно прочитать как “это поле равно остатку от деления значения переменной `?size` на 4”.

Компьютер должен попытаться взять такое количество палочек, чтобы этот остаток от деления общего количества палочек на четыре стал равным единице. А если указанный остаток равен единице сразу после того как очередность хода переходит к компьютеру, то компьютер неизбежно проиграет, при условии, что его противник не допустит ошибку. В данном случае компьютер берет только одну палочку, чтобы затянуть игру (в надежде, что игрок-человек допустит ошибку). Во всех других случаях компьютер может взять такое подходящее количество палочек, которое вынудит проиграть его противника.

Для того чтобы ознакомиться более подробно с тем, как действует ограничение поля для возвращаемого значения, рассмотрим конкретный пример. Предположим, что в куче находятся 7 палочек и очередность хода принадлежит компьютеру. Проверка первых трех условных элементов правила `computer-move` завершится успешно, и останется только проверить последний условный элемент. В функциональном выражении `(mod ?size 4)` вместо переменной `?size` бу-

дет подставлено значение 7 и получено выражение (mod 7 4), результат вычисления которого равен 3. В списке фактов имеется только один факт take-sticks со значением слота for-remainder, равным 3. Для этого факта значение слота how-many равно 2, поэтому компьютер в конечном итоге возьмет 2 палочки, оставив кучу с пятью палочками. А после любого следующего хода игрока-человека компьютер вынудит его проиграть.

Обратите внимание на, что ограничение поля =, которое может использоваться только в условном элементе шаблона, не следует путать с предикативной функцией =, которая может применяться в ограничении :, в условном элементе test, в правой части правила или в приглашении верхнего уровня. И в этом случае, как и при использовании предикативного ограничения поля, можно формировать более сложные ограничения поля, применяя ограничения возвращаемого значения в сочетании с соединительными ограничениями поля ~, & и |.

8.12 Программа Sticks

В предыдущих разделах данной главы описаны все основные методы, необходимые для завершения программы Sticks. Полный листинг программы Sticks приведен в файле sticks.clp, который находится на компакт-диске, прилагаемом к данной книге. Ниже показан пример прогона этой программы после ее загрузки.

```
CLIPS> (reset)↵
CLIPS> (run)↵
Who moves first (Computer: c Human: h)? c↵
How many sticks in the pile? 15↵
Computer takes 2 stick(s).
13 stick(s) left in the pile.
How many sticks do you wish to take? 3↵
10 stick(s) left in the pile.
Computer takes 1 stick(s).
9 stick(s) left in the pile.
How many sticks do you wish to take? 2↵
7 stick(s) left in the pile.
Computer takes 2 stick(s).
5 stick(s) left in the pile.
How many sticks do you wish to take? 1↵
4 stick(s) left in the pile.
Computer takes 3 stick(s).
1 stick(s) left in the pile.
You must take the last stick!
```

```
You lose!
CLIPS>
```

8.13 Условный элемент or

До сих пор все рассматриваемые правила содержали между шаблонами **неявно заданный условный элемент and**. Это означает, что запуск правила осуществляется, только если все шаблоны принимают истинное значение. Кроме того, в языке CLIPS предусмотрена возможность задавать в левой части правил и **явный условный элемент and**, и **явный условный элемент or**.

В качестве примера применения условного элемента or рассмотрим приведенные ниже правила, предназначенные для использования в системе текущего контроля над промышленной установкой (см. главу 7), которые вначале формулируются без условного элемента or. Затем в данном разделе будет показано, как перезаписать эти правила с условным элементом or.

```
(defrule shut-off-electricity-1
   (emergency (type flood))
   =>
   (printout t "Shut off the electricity" crlf))
(defrule shut-off-electricity-2
   (extinguisher-system (type water-sprinkler)
                        (status on))
   =>
   (printout t "Shut off the electricity" crlf))
```

В данном случае записаны два отдельных правила, но с использованием условного элемента or их можно объединить в следующее единственное правило:

```
(defrule shut-off-electricity
   (or (emergency (type flood))
   (extinguisher-system (type water-sprinkler)
                        (status on)))
   =>
   (printout t "Shut off the electricity" crlf))
```

Одно это правило, в котором применяется условный элемент or, эквивалентно двум предыдущим правилам. Ввод в список фактов двух фактов, согласующихся с шаблонами данного правила, приведет к двукратному запуску этого правила, по одному разу в расчете на каждый из фактов.

Другие условные элементы могут быть включены вне условного элемента or и войти в состав неявного условного элемента and для всей левой части правила. Например, следующее правило:

```
(defrule shut-off-electricity
   (electrical-power (status on))
   (or (emergency (type flood))
   (extinguisher-system (type water-sprinkler)
                        (status on)))
   =>
   (printout t "Shut off the electricity" crlf))
```

эквивалентно таким двум правилам:

```
(defrule shut-off-electricity-1
   (electrical-power (status on))
   (emergency (type flood))
   =>
   (printout t "Shut off the electricity" crlf))
(defrule shut-off-electricity-2
   (electrical-power (status on))
   (extinguisher-system (type water-sprinkler)
                        (status on))
   =>
   (printout t "Shut off the electricity" crlf))
```

Таким образом, условные элементы `or` позволяют создавать правила, эквивалентные нескольким правилам, поэтому возможно, что созданное с их помощью правило будет активизироваться несколько раз, с учетом различных шаблонов, содержащихся в условном элементе `or`. Итак, возникает резонный вопрос: как устранить проблему неоднократного запуска? Например, не требуется многократный вывод сообщения "`Shut off the electricity`" (Отключите подачу электроэнергии), связанный с наличием нескольких активизирующих фактов. Ведь подачу электроэнергии требуется отключить только один раз, независимо от того, сколько причин вынуждают это сделать.

По-видимому, наиболее приемлемый способ предотвращения запуска прочих правил состоит в модификации данного правила таким образом, чтобы оно обновляло список фактов, указывая, что сообщение о необходимости отключения электроэнергии уже передано. Модифицированное правило приведено ниже. (Важно отметить, что эти правила сформулированы на основании того предположения, что в нашем распоряжении есть внимательный оператор технологической установки, который следит за сообщениями системы текущего контроля и предпринимает необходимые действия. В реальном мире экспертной системе может быть предоставлена возможность непосредственно управлять включением и отключением различных систем; еще один вариант состоит в том, что оператору технологической установки может быть предоставлен механизм, позволяющий сообщить

системе текущего контроля о том, что им были выполнены те или другие действия.)

```
(defrule shut-off-electricity
   ?power <- (electrical-power (status on))
   (or (emergency (type flood))
   (extinguisher-system (type water-sprinkler)
                        (status on)))
   =>
   (modify ?power (status off)))
   (printout t "Shut off the electricity" crlf))
```

После этой корректировки запуск правила осуществляется только один раз, поскольку вслед за запуском правила модифицируется факт, содержащий информацию о подаче электроэнергии. В результате этого возможность других активизаций правила под действием других шаблонов исключается. А если необходимо также удалить из списка фактов сам факт, ставший причиной отключения подачи электроэнергии, то данное правило должно быть записано следующим образом:

```
(defrule shut-off-electricity
   ?power <- (electrical-power (status on))
   (or ?reason <- (emergency (type flood))
   ?reason <- (extinguisher-system
              (type water-sprinkler)
              (status on)))
   =>
   (retract ?reason)
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))
```

Обратите внимание на то, что все условные элементы шаблона, относящиеся к условному элементу `or`, связываются с одной и той же переменной `?reason`. На первый взгляд может показаться, что это — ошибка, поскольку чаще всего одна и та же переменная не присваивается разным шаблонам обычного правила. Но правила, в которых используются условные элементы `or`, имеет свою специфику. Дело в том, что применение условного элемента `or` фактически приводит к созданию нескольких правил, поэтому приведенное выше правило эквивалентно следующим двум правилам:

```
(defrule shut-off-electricity-1
   ?power <- (electrical-power (status on))
   ?reason <- (emergency (type flood))
   =>
   (retract ?reason)
```

```
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))
(defrule shut-off-electricity-2
   ?power <- (electrical-power (status on))
   ?reason <- (extinguisher-system
               (type water-sprinkler)
               (status on))
   =>
   (retract ?reason)
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))
```

Рассматривая эти два правила, можно убедиться в том, что для согласования действия `(retract ?power ?reason)` в правой части правила требовалось применить одно и то же имя переменной.

8.14 Условный элемент and

По своему назначению условный элемент `and` является противоположным условному элементу `or`. В последнем случае для активизации правила достаточно использовать любой из нескольких условных элементов, а в первом случае (когда применяется условный элемент `and`) требуется, чтобы были выполнены успешно проверки с помощью всех условных элементов. Система CLIPS автоматически включает левую часть правила в неявный условный элемент `and`. Например, правило

```
(defrule shut-off-electricity
   ?power <- (electrical-power (status on))
   (emergency (type flood))
   =>
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))
```

может быть также записано с явным условным элементом `and`, таким образом:

```
(defrule shut-off-electricity
   (and ?power <- (electrical-power (status on))
   (emergency (type flood)))
   =>
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))
```

Безусловно, способ записи правила с указанием явного условного элемента `and`, включающего всю левую часть правила, не дает никаких преимуществ.

Условный элемент and предусмотрен для того, чтобы его можно было использовать в сочетании с другими условными элементами для создания более сложных шаблонов. Например, условный элемент and может применяться в составе условного элемента or для обеспечения успешного выполнения проверки с помощью сразу нескольких условий, как показано в следующем примере:

```
(defrule use-carbon-dioxide-extinguisher
   ?system <- (extinguisher-system
               (type carbon-dioxide)
               (status off))
   (or (emergency (type class-B-fire))
       (and (emergency (type class-C-fire))
            (electrical-power (status off))))
   =>
   (modify ?system (status on))
   (printout t "Use carbon dioxide extinguisher"
               crlf))
```

Это правило активизируется, только если возникает чрезвычайная ситуация, при которой происходит пожар категории B (когда горит мазут или консистентная смазка) или категории C (охватывающий электрическое оборудование), а подача электроэнергии уже была отключена. По существу, углекислотные огнетушители всегда должны использоваться для тушения пожара категории B, но их применение для борьбы с пожаром категории C допустимо, только если выключение подачи электроэнергии не привело к угасанию огня. Правило use-carbon-dioxide-extinguisher эквивалентно следующим двум правилам:

```
(defrule use-carbon-dioxide-extinguisher-1
   ?system <- (extinguisher-system
               (type carbon-dioxide)
               (status off))
   (emergency (type class-B-fire))
   =>
   (modify ?system (status on))
   (printout t "Use carbon dioxide extinguisher"
               crlf))
(defrule use-carbon-dioxide-extinguisher-2
   ?system <- (extinguisher-system
               (type carbon-dioxide)
               (status off))
   (emergency (type class-C-fire))
   (electrical-power (status off))
   =>
```

```
   (modify ?system (status on))
   (printout t "Use carbon dioxide extinguisher"
               crlf))
```

8.15 Условный элемент `not`

Иногда возникает такая необходимость, чтобы активизацию правила можно было осуществить исходя из отсутствия какого-то конкретного факта в списке фактов. Система CLIPS позволяет задавать условие отсутствия некоторого факта в левой части правила с помощью **условного элемента not**. В качестве простого примера укажем, что экспертная система текущего контроля может иметь следующие два правила, позволяющие получить отчет о состоянии контролируемой установки:

```
IF the monitoring status is to be reported and
   there is an emergency being handled
THEN report the type of the emergency
IF the monitoring status is to be reported and
   there is no emergency being handled
THEN report that no emergency is being handled
```

Для реализации приведенных выше простых правил удобно применить условный элемент `not` следующим образом:

```
(defrule report-emergency
   (report-status)
   (emergency (type ?type))
   =>
   (printout t "Handling " ?type " emergency"
               crlf))
(defrule no-emergency
   (report-status)
   (not (emergency))
   =>
   (printout t "No emergency being handled" crlf))
```

Обратите внимание на то, что рассматриваемые два правила являются взаимоисключающими. Это означает, что эти два правила не могут находиться в одно и то же время в рабочем списке правил, поскольку невозможно одновременно получить успешный результат проверки второго шаблона в каждом правиле.

Для создания некоторых интересных эффектов в отрицаемом шаблоне можно также использовать переменные. Рассмотрим следующее правило, применяемое для поиска наибольшего числа в группе фактов, представляющих числа:

```
(defrule largest-number
   (number ?x)
   (not (number ?y&:(> ?y ?x)))
   =>
   (printout t "Largest number is " ?x crlf))
```

В первом шаблоне выполняется привязка к переменной значений всех фактов number, а второй шаблон предотвращает возможность активизации правила применительно к любому факту, кроме того факта, в котором хранится наибольшее значение ?x.

Заслуживает внимания то, что переменные, впервые связанные со значением в условном элементе not, сохраняют свое значение только в области определения условного элемента not. Например, применение следующего правила:

```
(defrule no-emergency
   (report-status)
   (not (emergency (type ?type)))
   =>
   (printout t "No emergency of type " ?type crlf))
```

приводит к появлению ошибки, поскольку переменная ?type используется в правой части правила, тогда как она остается связанной только в условном элементе not в левой части правила.

Область определения переменных необходимо также учитывать, анализируя левую часть правила. Например, следующее правило позволяет определить, есть ли такие лица, данные о которых имеются в списке фактов, день рождения которых выпадает на какой-то конкретный день:

```
(defrule no-birthdays-on-specific-date
   (check-for-no-birthdays (date ?date))
   (not (person (birthday ?date)))
   =>
   (printout t "No birthdays on " ?date crlf))
```

Если бы первые два условных элемента были переставлены местами, как показано ниже, то данное правило прекратило бы действовать должным образом.

```
(defrule no-birthdays-on-specific-date
   (not (person (birthday ?date)))
   (check-for-no-birthdays (date ?date))
   =>
   (printout t "No birthdays on " ?date crlf))
```

Проверка первого условного элемента оканчивалась бы неудачей, даже при наличии подходящих фактов person с данными о каких-либо людях. Дело в том, что значение, связанное с переменной ?date в первом условном элементе, не

влияет на допустимые значения для переменной ?date во втором условном элементе. Такая ситуация противоположна ситуации, складывающейся при использовании первоначальной версии правила no-birthdays-on-specific-date, в которой значение, связанное с переменной ?date в первом условном элементе, ограничивает поиск фактов person во втором условном элементе теми фактами, которые содержат данные о дне рождении, выпадающем на конкретную дату. В отличие от условных элементов шаблона, порядок расположения условного элемента not в левой части правила может повлиять на активизацию правила.

Условный элемент not может использоваться в сочетании с другими условными элементами. Например, следующее правило позволяет определить, нет ли в списке фактов информации о двух таких лицах, дни рождения которых выпадают на одну и ту же дату:

```
(defrule no-identical-birthdays
   (not (and (person (name ?name)
                     (birthday ?date))
   (person (name ~?name)
                     (birthday ?date))))
   =>
   (printout t
      "No two people have the same birthday"
      crlf)
```

В условном элементе not применяется условный элемент and, поскольку условный элемент not может содержать самое большее один условный элемент. Обратите внимание на то, что в условном элементе not для правильного ограничения поиска двух человек с одним и тем же днем рождения повторно используются переменные ?name и ?date.

Кроме того, специфика основополагающих алгоритмов, применяемых в системе CLIPS, такова, что шаблон (initial-fact) добавляется в начало любого условного элемента and (неявного или явного), первым условным элементом которого является условный элемент not или условный элемент test. Таким образом, следующее правило:

```
(defrule no-emergencies
   (not (emergency))
   =>
   (printout t "No emergencies" crlf))
```

преобразуется в такое правило:

```
(defrule no-emergencies
   (initial-fact)
   (not (emergency))
```

```
   =>
   (printout t "No emergencies" crlf))
```

Об этом преобразовании следует помнить, изучая вывод команды matches. Кроме того, необходимо учитывать, что индексы фактов для условных элементов not не отображаются в частичных согласованиях или в активизациях правил. Таким образом, активизация "f-5,,f-3" показывает, что первый условный элемент согласован с фактом, имеющим индекс факта 5, вторым условным элементом был условный элемент not, который не был согласован с каким-либо фактом, и поэтому проверка с его помощью завершилась успешно, а третий условный элемент согласован с фактом, имеющим индекс факта 3.

8.16 Условный элемент exists

Условный элемент exists позволяет выполнять сопоставление с шаблонами с учетом наличия по меньшей мере одного факта, согласующегося с шаблоном, не принимая во внимание то, какое общее количество фактов в действительности согласуется с шаблоном. Это дает возможность создавать единственное частичное согласование или единственную активизацию для правила исходя из наличия хотя бы одного факта из класса фактов. Например, предположим, что при возникновении любой критической ситуации необходимо выводить информационное сообщение, чтобы указать операторам технологической установки на то, что они должны быть готовы к неблагоприятному развитию событий. Такое правило может быть записано следующим образом:

```
(deftemplate emergency (slot type))
(defrule operator-alert-for-emergency
   (emergency)
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

Обратите внимание на то, что произойдет, если в список фактов будет введен больше чем один факт emergency, свидетельствующий о наличии критической ситуации, — сообщение операторам будет выведено несколько раз, как показано ниже.

```
CLIPS> (reset)↵
CLIPS> (assert (emergency (type fire)))↵
<Fact-1>
CLIPS>
(assert (emergency (type flood)))↵
<Fact-2>
```

```
CLIPS> (run)↵
Emergency: Operator Alert
Emergency: Operator Alert
CLIPS>
```

Правило operator-alert-for-emergency можно модифицировать следующим образом для предотвращения повторной активизации правила в случае возникновения еще одной критической ситуации:

```
(defrule operator-alert-for-emergency
   (emergency)
   (not (operator-alert))
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

Повторную активизацию правила предотвращает условный элемент (not (operator-alert)). Но, модифицируя правило указанным образом, приходится исходить из предположения, что если к операторам уже поступил тревожный сигнал, то он был активизирован с помощью правила operator-alert-for-emergency. Однако, как показывает следующее правило, тревожный сигнал операторам может быть передан в ходе учебы по подготовке к устранению чрезвычайных ситуаций:

```
(defrule operator-alert-for-drill
   (operator-drill)
   (not (operator-alert))
   =>
   (printout t "Drill: Operator Alert" crlf)
   (assert (operator-alert)))
```

Обратите внимание на то, что если вначале произойдет запуск правила, касающегося формирования тревожного сигнала в ходе учебной подготовки, то в дальнейшем невозможно будет выполнить запуск правила формирования тревожного сигнала при возникновении критической ситуации, поскольку факт operator-alert (оператору передано сообщение) уже был вставлен в список фактов. Для устранения этой проблемы можно модифицировать структуру факта operator-alert, чтобы он хранил данные о причине формирования тревожного сигнала. Тем не менее при чрезмерном использовании управляющих фактов для предотвращения запуска правил сложность правил увеличивается, а удобство сопровождения правил уменьшается, поскольку вводятся дополнительные управляющие зависимости между правилами.

К счастью, правило operator-alert-for-emergency можно модифицировать для использования в нем условного элемента exists, который устраняет

необходимость в применении управляющих фактов. Условный элемент exists вырабатывает только одно частичное согласование, независимо от того, сколько фактов сопоставляется с условными элементами в условном элементе exists. Таким образом, если условный элемент exists является N-м условным элементом правила и предыдущие N–1 условных элементов выработали M частичных согласований, то наибольшее количество частичных согласований, которое может быть выработано первыми N условными элементами, равно M. Ниже приведено правило operator-alert-for-emergency, откорректированное в целях использования условного элемента exists.

```
(defrule operator-alert-for-emergency
   (exists (emergency))
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

Это правило вырабатывает только одну активизацию, и поэтому тревожное сообщение операторам будет выведено только один раз, как показывает следующий диалог:

```
CLIPS> (reset)↵
CLIPS> (assert (emergency (type fire)))↵
<Fact-1>
CLIPS>
(assert (emergency (type flood)))↵
<Fact-2>
CLIPS> (agenda)↵
0      operator-alert-for-emergency: f-0,
For a total of 1 activation.
CLIPS> (run)↵
Emergency: Operator Alert
CLIPS>
```

Условный элемент exists реализуется с использованием сочетания условных элементов and и условных элементов not. Условные элементы, относящиеся к условному элементу exists, заключаются в один условный элемент and, а затем в два условных элемента not. Таким образом, правило operator-alert-for-emergency можно преобразовать в следующее правило, заключив всю левую часть правила в условный элемент and, а затем заменив условный элемент exists, как указано выше:

```
(defrule operator-alert-for-emergency
   (and (not (not (and (emergency)))))
   =>
```

```
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

Условный элемент `and`, в который входит левая часть правила, содержит в качестве первого условного элемента условный элемент `not`, поэтому в начале добавляется условный элемент шаблона `(initial-fact)`. В результате добавления шаблона `(initial-fact)` и удаления ненужного условного элемента `and`, включающего в себя шаблон `emergency`, формируется следующее правило:

```
(defrule operator-alert-for-emergency
   (and (initial-fact)
   (not (not (emergency))))
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

Объяснением тому, что в приведенном выше диалоге для данного правила после выдачи команды `agenda` был указан индекс факта `f-0`, является наличие шаблона `(initial-fact)` в начале правила. Если фактов `emergency` в списке фактов нет, то проверка самого внутреннего условного элемента `not` завершается успешно. А если проверка этого условного элемента завершается успешно, то проверка самого внешнего условного элемента `not` оканчивается неудачей, поэтому правило не активизируется. И наоборот, если в списке фактов есть факты `emergency`, то проверка самого внутреннего условного элемента `not` оканчивается неудачей. А поскольку проверка этого условного элемента оканчивается неудачей, проверка самого внешнего условного элемента `not` завершается успешно и правило активизируется.

8.17 Условный элемент `forall`

Условный элемент forall позволяет выполнять сопоставление с шаблонами с использованием множества условных элементов, проверка которых завершается успешно применительно к каждому вхождению другого условного элемента. Например, предположим, что промышленная установка расположена на нескольких производственных площадках (таких как отдельные здания), на которых может возникнуть пожар, и мы хотим определить, проведена ли эвакуация в каждом здании, охваченном пожаром, и отправлена ли в каждое здание команда пожарников, пытающихся погасить огонь. Для проверки этого можно воспользоваться условным элементом `forall`. С этой целью модифицирован факт `emergency`, как показано ниже, чтобы он содержал информацию не только о том, к какому типу относится чрезвычайная ситуация, но и в каком месте она возникла. Две другие конструкции `deftemplate` применяются для обозначения местонахождения пожар-

ных команд и для указания на то, эвакуированы ли люди из здания. Эти конструкции deftemplate используются в правиле all-fires-being-handled для определения того, выполнены ли указанные условия.

```
(deftemplate emergency
   (slot type)
   (slot location))
(deftemplate fire-squad
   (slot name)
   (slot location))
(deftemplate evacuated
   (slot building))
(defrule all-fires-being-handled
   (forall (emergency (type fire)
                      (location ?where))
           (fire-squad (location ?where))
           (evacuated (building ?where)))
   =>
   (printout t
   "All buildings that are on fire " crlf
   "have been evacuated and" crlf
   "have firefighters on location" crlf))
```

Для каждого факта, который согласуется с шаблоном (emergency (type fire) (location ?where)), должны присутствовать также факты, согласующиеся с шаблонами (fire-squad (location ?where)) и (evacuated (building ?where)). После первоначальной загрузки в систему указанных конструкций deftemplate и конструкции defrule с последующей выдачей команды reset проверка условий этого правила должна быть завершена успешно, поскольку чрезвычайные ситуации, связанные с пожарами, отсутствуют, как показано ниже.

```
CLIPS> (watch activations)↵
CLIPS>
(reset)↵
==> Activation 0           all-fires-being-handled: f-0,
CLIPS>
```

А после ввода в список фактов любого факта emergency данное правило переводится в неактивное состояние и остается в нем до тех пор, пока в список фактов не будут введены соответствующие факты fire-squad и evacuated:

```
CLIPS>
(assert (emergency (type fire)
```

```
                    (location building-11)))↵
<== Activation 0   all-fires-being-handled: f-0,
<Fact-1>
CLIPS>
(assert (evacuated (building building-11)))↵
<Fact-2>
CLIPS>
(assert (fire-squad (name A)
                    (location building-11)))↵
==> Activation 0    all-fires-being-handled: f-0,
<Fact-3>
CLIPS>
(assert (fire-squad (name B)
                    (location building-1)))↵
<Fact-4>
CLIPS>
(assert (emergency (type fire)
                   (location building-1)))↵
<== Activation 0   all-fires-being-handled: f-0,
<Fact-5>
CLIPS> (assert (evacuated (building building-1)))↵
==> Activation 0   all-fires-being-handled:
 f-0,
<Fact-6>
CLIPS>
```

Если будет удален факт `fire-squad`, касающийся здания `building-1`, то правило переводится в неактивное состояние. А если будет удален факт `emergency`, относящийся к тому же зданию, это правило снова активизируется следующим образом:

```
CLIPS> (retract 4)↵
<== Activation 0        all-fires-being-handled: f-0,
CLIPS> (retract 5)↵
==> Activation 0        all-fires-being-handled: f-0,
CLIPS> (run)↵
All buildings that are on fire
have been evacuated and
have firefighters on location
CLIPS>
```

Условный элемент forall имеет такой общий формат:

```
(forall <first-CE>
        <remaining-CEs>+)
```

Для того чтобы проверка с помощью условного элемента forall была завершена успешно, каждому факту, согласующемуся с шаблоном <first-CE>, должны также соответствовать факты, согласующиеся со всеми шаблонами <remaining-CEs>. Этот общий формат условного элемента forall может быть заменен сочетаниями условных элементов and и not, как показывает следующее определение:

```
(not (and <first-CE>
     (not (and <remaining-CEs>+))))
```

8.18 Условный элемент logical

Условный элемент logical позволяет указать, что существование некоторого факта зависит от существования другого факта или группы фактов. Условный элемент logical представляет собой средство поддержания истинности, предусмотренное в языке CLIPS. В качестве примера рассмотрим следующее правило, которое показывает, что пожарники должны использовать кислородные маски, если при пожаре выделяется ядовитый дым:

```
(defrule noxious-fumes-present
   (emergency (type fire))
   (noxious-fumes-present)
   =>
   (assert (use-oxygen-masks)))
```

Как показывает следующий диалог, в приведенном выше правиле предусмотрена вставка факта use-oxygen-masks в список фактов каждый раз, когда возникает указанная критическая ситуация, связанная с пожаром:

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS>
(assert (emergency (type fire))
                   (noxious-fumes-present)))↵
==> f-1     (emergency (type fire))
==> f-2     (noxious-fumes-present)
<Fact-2>
CLIPS> (run)↵
```

```
==> f-3     (use-oxygen-masks)
CLIPS>
```

А что должно произойти после того, как пожар будет потушен и в воздухе рассеется ядовитый дым? Как показывает следующий диалог, извлечение факта `emergency` или `noxious-fumes-present` не влияет на факт `use-oxygen-masks`.

```
CLIPS> (retract 1 2)↵
<== f-1     (emergency (type fire))
<== f-2     (noxious-fumes-present)
CLIPS> (facts)↵
f-0   (initial-fact)
f-3   (use-oxygen-masks)
For a total of 2 facts.
CLIPS>
```

В языке CLIPS предусмотрен механизм поддержания истинности для создания зависимостей между фактами; таким механизмом является условный элемент `logical`. Ниже приведено модифицированное правило `noxious-fumes-present`, позволяющее создать зависимость между фактами, которые сопоставляются с шаблонами в левой части правила, и фактами, введенными в список фактов в результате выполнения правой части правила.

```
(defrule noxious-fumes-present
   (logical (emergency (type fire))
   (noxious-fumes-present))
   =>
   (assert (use-oxygen-masks)))
```

После выполнения правила `noxious-fumes-present` создается связь между фактами, сопоставляемыми с шаблонами, которые содержатся в условном элементе в левой части правила `logical`, и фактами, введенными в список фактов в результате выполнения правой части правила. Благодаря действию этого правила, если извлекается либо факт `emergency`, либо `noxious-fumes-present`, то извлекается также факт `use-oxygen-masks`, как показывает следующий диалог:

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS>
(assert (emergency (type fire))
                   (noxious-fumes-present)))↵
==> f-1     (emergency (type fire))
```

```
==> f-2     (noxious-fumes-present)
<Fact-2>
CLIPS> (run)↵
==> f-3     (use-oxygen-masks)
CLIPS> (retract 1)↵
<== f-1     (emergency (type fire))
<== f-3     (use-oxygen-masks)
CLIPS>
```

Это означает, что факт use-oxygen-masks получает **логическое обоснование** от фактов emergency и noxious-fumes-present. Иначе эта мысль формулируется таким образом, что факты emergency и noxious-fumes-present предоставляют логическое обоснование факту use-oxygen-masks. Факт use-oxygen-masks называется **зависимым** от фактов emergency и noxious-fumes-present. Факты noxious-fumes-present и emergency называются **зависимостями** для факта use-oxygen-masks.

Условный элемент logical не обязательно должен включать все шаблоны в левой части правила. Но если этот условный элемент используется, то должен включать первый условный элемент в левой части правила, а количество условных элементов logical в правиле не должно быть больше одного. Например, нельзя включить в условные элементы logical второй и четвертый условные элементы правила или даже включить в них первый и третий условные элементы правила, поскольку при этом область действия условного элемента logical разрывается. Такое ограничение по использованию условного элемента logical связано со спецификой реализации, лежащей в его основе. Кроме того, условный элемент logical позволяет сделать факты зависимыми от отсутствия других фактов с использованием условных элементов not. В условном элементе logical можно также проверять более сложные условия с помощью условных элементов exists и forall или других сочетаний условных элементов. Но условный элемент logical действует во всех отношениях подобно условному элементу and, не считая того, что он создает зависимости между группами фактов.

Для модификации правила noxious-fumes-present с тем, чтобы сделать факт use-oxygen-masks зависимым только от факта noxious-fumes-present, потребовалось бы переупорядочить шаблоны, как показано ниже.

```
(defrule noxious-fumes-present
   (logical (noxious-fumes-present))
   (emergency (type fire))
   =>
   (assert (use-oxygen-masks)))
```

Как показывает это модифицированное правило, извлечение факта use-oxygen-masks не будет происходить автоматически после извлечения факта

emergency (в этом состоит более безопасная организация работы по пожаротушению, поскольку ядовитый дым может все еще содержаться в воздухе, даже если огонь полностью погашен).

При обычных обстоятельствах попытка ввести факт, который уже находится в списке фактов, не приводит к возникновению каких-либо изменений в списке фактов. С другой стороны, логически зависимый факт, происходящий из нескольких источников, не извлекается автоматически до тех пор, пока не будет удалено логическое обоснование со стороны всех источников этого факта. Например, предположим, что добавлено одно следующее правило для указания на то, что в случае применения газовых огнетушителей пожарники должны надевать кислородные маски:

```
(defrule gas-extinguishers-in-use
   (logical (gas-extinguishers-in-use))
   (emergency (type fire))
   =>
   (assert (use-oxygen-masks)))
```

Теперь эксплуатация системы вызовет ввод в список фактов одного и того же факта под действием двух отдельных правил и по разным причинам, как показано ниже.

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (assert (emergency (type fire))
               (noxious-fumes-present)
               (gas-extinguishers-in-use))↵
==> f-1     (emergency (type fire))
==> f-2     (noxious-fumes-present)
==> f-3     (gas-extinguishers-in-use)
<Fact-3>
CLIPS> (run)↵
FIRE     1 gas-extinguishers-in-use: f-3,f-1
==> f-4        (use-oxygen-masks)
FIRE     2 noxious-fumes-present: f-1,f-2
CLIPS>
```

Извлечение факта noxious-fumes-present не будет достаточным, чтобы вызвать автоматическое извлечение факта use-oxygen-masks, поскольку остается еще одно логическое обоснование для использования кислородных масок. Прежде чем появится возможность извлечь факт use-oxygen-masks, необхо-

димо также извлечь факт gas-extinguishers-in-use, как показывает следующий диалог:

```
CLIPS> (retract 2)↵
<== f-2         (noxious-fumes-present)
CLIPS> (retract 3)↵
<== f-3         (gas-extinguishers-in-use)
<== f-4         (use-oxygen-masks)
CLIPS>
```

Факт, введенный в список фактов из приглашения верхнего уровня или из правой части правила, не имеющего каких-либо условных элементов logical в своей левой части, называется **безусловно обоснованным**. Факт, который является безусловно обоснованным, ни в коем случае не будет извлечен автоматически в результате извлечения другого факта. А после того как факт получает безусловное обоснование, все предусмотренные ранее логические обоснования того же факта отбрасываются.

В языке CLIPS предусмотрены две команды, позволяющие просматривать **зависимые факты** (команда dependents) и **зависимости** (команда dependencies), связанные с фактами. Эти команды имеют следующий синтаксис:

```
(dependents <fact-index-or-address>)
(dependencies <fact-index-or-address>)
```

Применительно к последнему примеру эти команды, выполненные до извлечения фактов noxious-fumes-present и gas-extinguishers-in-use, приводят к получению следующего вывода:

```
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (emergency (type fire))
f-2     (noxious-fumes-present)
f-3     (gas-extinguishers-in-use)
f-4     (use-oxygen-masks)
For a total of 5 facts.
CLIPS> (dependents 1)↵
None
CLIPS> (dependents 2)↵
f-4
CLIPS> (dependents 3)↵
f-4
CLIPS> (dependents 4)↵
None
CLIPS> (dependencies 1)↵
```

```
None
CLIPS> (dependencies 2)↵
None
CLIPS> (dependencies 3)↵
None
CLIPS> (dependencies 4)↵
f-2
f-3
CLIPS>
```

8.19 Резюме

В настоящей главе приведено вводное описание понятия ограничений поля. Ограничения поля позволяют применять отрицания ограничений и формировать комбинации больше чем из одного ограничения для данного поля. Ограничение поля `not` используется для предотвращения согласования с определенными значениями. Ограничение поля `and` позволяет обеспечить, чтобы целый ряд условий согласования принимал истинное значение. Ограничение поля `or` служит для обеспечения того, чтобы было истинным по крайней мере одно из целого ряда условий согласования.

Функции вводятся в цикле обработки команд верхнего уровня системы CLIPS либо используются в левых или правых частях правил. Многие функции могут иметь переменное количество параметров (в частности, к ним относятся некоторые арифметические функции). В ходе выполнения вызовов функций могут осуществлять вызовы других функции; такие вызовы называются *вложенными*. Команда `bind` позволяет связывать переменные в правой части правила.

В языке CLIPS предусмотрено несколько функций ввода-вывода. Функции `open` и `close` предназначены для открытия и закрытия файлов. Открытые файлы ассоциированы с логическим именем. Логические имена могут использоваться в большинстве функций, которые осуществляют ввод и вывод с использованием больше чем одного типа физических устройств. В частности, логические имена применяются в функциях `printout` и `read`. Функция `printout` обеспечивает вывод на терминал и в файлы. Функция `read` позволяет выполнять ввод с клавиатуры и из файлов. Функции `format` и `readline` также принимают в качестве параметров логические имена. Функция `format` обеспечивает больший контроль над тем, какое внешнее представление приобретают выводимые данные. Функция `readline` может использоваться для чтения целой строки данных. Функция `explode$` преобразовывает строку в многозначное значение.

В программе на языке CLIPS могут использоваться различные методы управления потоком выполнения. В настоящей главе демонстрируется, как может быть

создан простой цикл управления (в котором из списка фактов извлекаются, а затем снова вставляются управляющие факты) для ввода данных с использованием функции `read`. Для обеспечения более мощных возможностей сопоставления с шаблонами в левой части правила наряду с предикативными функциями могут использоваться условные элементы `test`. Кроме того, условные элементы `test` могут служить для поддержания функционирования цикла управления. Предикативные ограничения поля позволяют предусматривать предикативные проверки непосредственно в шаблоне. Ограничения поля со знаком равенства используются для сравнения значения поля со значением, возвращаемым функцией. Некоторые из этих методов управления демонстрируются в программе Sticks.

Кроме условного элемента `test`, в программе на языке CLIPS могут применяться еще несколько типов условных элементов. Условный элемент `or` используется для представления нескольких правил как единственного правила. Условный элемент `not` позволяет выполнять сопоставление с шаблонами исходя из отсутствия некоторого факта в списке фактов. Условный элемент `and` используется для группирования условных элементов и вместе с условными элементами `or` и `not` позволяет создавать выражения с произвольной вложенностью, представляющие сложные условия, которые необходимы для активизации правила. Для определения того, существует ли по крайней мере одна группа фактов, которая согласуется с условным элементом или с комбинацией условных элементов, используется условный элемент `exists`. Условный элемент `forall` позволяет определить, соответствует ли множество условных элементов каждому вхождению другого условного элемента. Условный элемент `logical` позволяет создать механизм поддержания истинности. В программе можно добиться того, чтобы присутствие в списке фактов некоторых фактов зависело от присутствия или отсутствия других фактов.

Задачи

8.1. Предположим, что даны следующие конструкции `deftemplate` для фактов, представляющих генеалогическое дерево:

```
(deftemplate father-of (slot father) (slot child))
(deftemplate mother-of (slot mother) (slot child))
(deftemplate male (slot person))
(deftemplate female (slot person))
(deftemplate wife-of (slot wife) (slot husband))
(deftemplate husband-of (slot husband) (slot wife))
```

Напишите правила, позволяющие вывести перечисленные ниже отношения. Составьте конструкции `deftemplate`, применяемые для выполнения этого задания.

а) `uncle` (дядя), `aunt` (тетя);

б) `cousin` (двоюродный брат или двоюродная сестра);

в) `grandparent` (дедушка или бабушка);

г) `grandfather` (дедушка), `grandmother` (бабушка);

д) `sister` (сестра), `brother` (брат);

е) `ancestor` (предок).

8.2. На промышленной установке имеются десять датчиков с идентификационными номерами 1–10. Каждый датчик показывает состояние контролируемого узла — "исправный" или "неисправный". Составьте конструкцию `deftemplate` для представления состояния датчиков и напишите одно или несколько правил, которые выводят предупреждающее сообщение, если три или больше датчиков показывают неисправное состояние. Проверьте подготовленные вами правила с датчиками 3 и 5, показывающими неисправное состояние; с датчиками 2, 8 и 9, показывающими неисправное состояние; и с датчиками 1, 3, 5 и 10, показывающими неисправное состояние. Что необходимо сделать, чтобы предотвратить многократное отображение предупреждающего сообщения?

8.3. Разработайте программу CLIPS на основе правил `IF-THEN`, составленных в ходе решения задачи 3.5, представленной на стр. 289. Программа должна задавать вопросы о том, какой метод оплаты предпочитает путешественник и какое путешествие его интересует, после чего предлагать возможные поездки с учетом ответов на эти два вопроса.

8.4. Предположим, что дано несколько совокупностей фактов, описывающих геометрические фигуры, в которых используются следующие конструкции `deftemplate`:

```
(deftemplate square
   (slot id) (slot side-length))
(deftemplate rectangle
   (slot id) (slot width) (slot height))
(deftemplate circle
   (slot id) (slot radius))
```

Напишите одно или несколько правил, которые вычисляют указанные ниже суммы.

а) Сумма площадей фигур.

б) Сумма периметров фигур.

Проверьте вывод этих правил с помощью следующих конструкций `deffacts`:

```
(deffacts test-8-8
   (square (id A) (side-length 3))
   (square (id B) (side-length 5))
   (rectangle (id C) (width 5) (height 7))
   (circle (id D) (radius 2))
   (circle (id E) (radius 6)))
```

8.5. Предположим, что дана информация об имени, цвете глаз и волос, а также о гражданстве лица, принадлежащего к некоторой группе, с использованием следующей конструкции `deftemplate`:

```
(deftemplate person (slot name)
   (slot eye-color)
   (slot hair-color)
   (slot nationality))
```

Напишите одно правило, которое позволяет выявить описанных ниже лиц.

а) Любого человека с синими или зелеными глазами, который имеет каштановые волосы и прибыл из Франции.

б) Любого человека, кто не имеет синих глаз или темных волос, а также не имеет волос и глаз одинаково светлого или одинаково темного цвета.

в) Двух человек, первый из которых имеет карие или синие глаза, не имеет светлые волосы и является гражданином Германии; второй имеет зеленые глаза, тот же цвет волос, что и первый человек, а также может быть гражданином любого государства. Глаза второго человека могут быть карими, если волосы первого человека — каштановые.

8.6. Преобразуйте приведенные ниже инфиксные выражения в префиксные выражения.

а) `(3 + 4) * (5 + 6) + 7.`

б) `(5 * (5 + 6 + 7)) - ((3 * (4 / 9) + 2) / 8).`

в) `6 - 9 * 8 / 3 + 4 - (8 - 2 - 3) * 6 / 7.`

8.7. Рассмотрите следующую информацию о бейсбольной команде. Энди не любит кетчера. Сестра Эда замужем за вторым бейсменом. Центральный фильдер имеет более высокий рост, чем правый фильдер. Гарри и третий бейсмен живут в одном доме. И Пол, и Аллен выиграли по 20 долларов у питчера в пинокль. Эд и игроки нападающей команды в свободное время играют в покер. Жена питчера — сестра третьего бейсмена. Вся батарея (питчер и кетчер) и защищающаяся команда, кроме Аллена, Гарри и Энди, имеют рост меньше, чем у Сэма. И Пол, и Энди, и шорт-стоп проиграли по 50 долларов на скачках. Пол, Гарри, Билл и кетчер потерпели поражение в бейсбольном матче от второго бейсмена. Сэм проходит процедуру развода.

И кетчер, и третий бейсмен имеют по два ребенка. Эд, Пол, Джерри, правый фильдер и центральный фильдер — холостяки; остальные члены команды женаты. И шорт-стоп, и третий бейсмен, и Билл заработали по 100 долларов, держа пари на один боксерский поединок. Одного из игроков нападающей команды зовут или Майк, или Энди. Джерри имеет более высокий рост, чем Билл. Майк ниже ростом, чем Билл. Каждый из них весит больше, чем третий бейсмен. Сэм, кетчер и третий бейсмен — левши. Эд, Сэм и шорт-стоп вместе ходили в среднюю школу. Напишите программу CLIPS, чтобы определить имена всех членов команды.

8.8. Преобразуйте дерево решений, показанное на рис. 3.3 (см. стр. 197), в ряд правил CLIPS. Создайте шаблоны для согласования с фактами с использованием следующей конструкции `deftemplate`:

```
(deftemplate question
   (slot query-string)
   (slot answer))
```

Например, если ответом на вопрос, представленный корневым узлом в дереве, является слово “no”, то факт, представляющий эту информацию, должен иметь такой вид:

```
(question (query-string "Is it very big?")
          (answer no))
```

Используйте в правой части правил функции `printout` и `read`, чтобы задавать вопросы, показанные на рис. 3.3, и вводите в базу знаний факты `question` с ответами пользователя.

8.9. Напишите программу CLIPS, которая складывает два двоичных числа без использования каких-либо арифметических функций. Используйте для представления двоичных чисел следующую конструкцию `deftemplate`:

```
(deftemplate binary-#
   (multislot name)
   (multislot digits))
```

После ввода фактов, указывающих, какие два двоичных числа необходимо сложить, программа должна создать новое именованное двоичное число, содержащее сумму. Например, после ввода следующих фактов:

```
(binary-# (name A) (digits 1 0 1 1 1))
(binary-# (name B) (digits 1 1 1 0))
(add-binary-#s (name-1 A) (name-2 B))
```

к списку фактов будет добавлен такой факт:

```
(binary-# (name { A + B }) (digits 1 0 0 1 0 1))
```

8.10. Напишите программу CLIPS, которая запрашивает группу крови пациента, нуждающегося в переливании крови, и группу крови донора. Затем программа на основании данных о группах крови должна определить, должно ли быть сделано переливание крови от данного донора. Кровь группы O может быть перелита только пациенту с группой крови O. Кровь группы A может быть перелита пациенту с группой крови A или пациенту с группой крови O. Кровь группы B может быть перелита пациенту с группой крови B или с группой крови O. Кровь группы AB может быть перелита пациенту с группой крови AB, группой крови A, группой крови B или группой крови O.

8.11. Напишите программу CLIPS, которая предоставляет информацию либо о способах приготовления указанных мясных полуфабрикатов из говядины, либо о мясных полуфабрикатах из говядины, которые могут быть приготовлены с применением указанного способа. Программа должна вначале задать вопрос о том, какая информация должна быть получена, — о мясных полуфабрикатах или способах приготовления, а затем должна предложить сделать соответствующий выбор. Для определения соответствующих мясных полуфабрикатов или способов приготовления используйте следующие рекомендации: кострец на ростбиф следует тушить или запекать; “английский” филей следует жарить в жире, жарить на открытой сковороде почти без жира или жарить на сковороде с малым количеством жира; бифштекс на ребрышке следует жарить в жире, жарить на открытой сковороде почти без жира или жарить на сковороде с малым количеством жира; кусок края следует запекать; говяжий фарш следует запекать, жарить в жире, жарить на открытой сковороде почти без жира, жарить на сковороде с малым количеством жира или тушить; кусок пашины для стейка следует тушить; вырезку следует тушить.

8.12. Модифицируйте программу, разработанную в результате решения задачи 7.14 (см. с. 620), таким образом, чтобы для определения входных данных пользователю предъявлялся ряд вопросов о необходимых характеристиках кустарника. Выходные данные программы должны оставаться прежними.

8.13. Для классификации микроорганизмов применяется несколько их характеристик, включая их основную форму (сферическая, палочковидная, спиралевидная или нитевидная), результаты лабораторного исследования окрашивания по Граму (положительные, отрицательные или отсутствующие), а также потребность в кислороде для выживания (аэробный или анаэробный микроорганизм). Напишите программу, которая идентифицирует тип микроорганизма на основе информации, представленной в табл. 8.4. Пользователь должен указывать по запросу программы форму, окрашивание по Граму и потребность микроорганизма в кислороде. Пользователь должен иметь

возможность указывать, что любые из входных данных являются неизвестными. Результаты работы программы должны представлять собой список всех возможных разновидностей микроорганизмов, составленный на основе информации, полученной от пользователя.

Таблица 8.4. Признаки микроорганизмов

Тип	Форма	Окрашивание по Граму	Потребность в кислороде
Актиномицеты	Палочковидная или нитевидная	Положительное	Аэробный
Коккоиды	Сферическая	Положительное	Аэробный и анаэробный
Коринемикроорганизмы	Палочковидная	Положительное	Аэробный
Формирующие эндоспоры	Палочковидная	Положительное или отрицательное	Аэробный и анаэробный
Кишечные	Палочковидная	Отрицательное	Аэробный
Жгутиковые	Палочковидная	Отрицательное	Аэробный
Микомикроорганизмы	Сферическая	Отсутствует	Аэробный
Микоплазма	Сферическая	Отсутствует	Аэробный
Псевдомонады	Палочковидная	Отрицательное	Аэробный
Риккетсии	Сферическая или палочковидная	Отрицательное	Аэробный
Стрептококки	Нитевидная	Отрицательное	Аэробный
Спириллы	Спиралевидная	Отрицательное	Аэробный
Спирохеты	Спиралевидная	Отрицательное	Анаэробный
Вибрионы	Палочковидная	Отрицательное	Аэробный

8.14. Компания Acme Electronics изготавливает устройства, известные под названием Thingamabob 2000. Эти устройства выпускаются в пяти различных модификациях, которые отличаются по конструкции шасси. В каждом шасси предусмотрен ряд отсеков для установки дополнительных приспособлений и имеется блок питания, способный вырабатывать определенное количество единиц мощности. Общие сведения о характеристиках шасси приведены в табл. 8.5.

Таблица 8.5. Общие сведения о характеристиках шасси

Шасси	Количество отсеков, предусмотренных для приспособлений	Вырабатываемая мощность	Цена (в долларах)
C100	1	4	2000,00
C200	2	5	2500,00
C300	3	7	3000,00
C400	2	8	3000,00
C500	4	9	3500,00

Для работы каждого приспособления, которое может быть установлено в блоке, требуется определенное количество единиц мощности. Общие сведения о характеристиках приспособлений приведены в табл. 8.6.

Таблица 8.6. Общие сведения о характеристиках приспособлений

Приспособление	Потребляемая мощность	Цена (в долларах)
Zaptron	2	100,00
Yatmizer	6	800,00
Phenerator	1	300,00
Malcifier	3	200,00
Zeta-shield	4	150,00
Warnosynchronizer	2	50,00
Dynoseparator	3	400,00

Напишите программу, которая принимает в качестве входных данных ряд фактов, представляющих результаты выбора пользователем шасси и всех необходимых приспособлений, а затем вырабатывает факты, представляющие количество используемых приспособлений, общее количество единиц мощности, необходимых для работы приспособлений, а также общую стоимость шасси и всех выбранных приспособлений.

8.15. Используя табл. 7.2 с данными о драгоценных камнях, представленную в задаче 7.13 (см. с. 620), напишите правила идентификации следующих драгоценных камней: алмаз, корунд, хризоберилл, шпинель, кварц и турмалин. Включите правила, позволяющие запрашивать у пользователя данные о твердости, плотности и цвете драгоценного камня. Если пользователь неправильно укажет цвет, то вопрос должен повторяться до тех пор, пока не будет указан один из цветов, перечисленных в таблице.

8.16. Введите в программу Sticks дополнительные правила, с помощью которых пользователю после окончания игры будет передаваться вопрос, желает ли он сыграть еще раз.

8.17. Модифицируйте программу Sticks так, чтобы она позволяла не только человеку играть против компьютера, но и вести игру друг против друга двум игрокам-людям.

8.18. Представьте следующие правила в виде единственного правила с использованием условных элементов `and` и `or`:

```
(defrule rule-1
   (fact-a)
   (fact-d)
   =>)
(defrule rule-2
   (fact-b)
   (fact-c)
   (fact-e)
   (fact-f)
   =>)
(defrule rule-3
   (fact-a)
   (fact-e)
   (fact-f)
   =>)
(defrule rule-4
   (fact-b)
   (fact-c)
   (fact-d)
   =>)
```

8.19. Напишите с использованием условных элементов `and` и `or` программу для дерева AND–OR, показывающего, как добраться до места работы с помощью различных способов, которое приведено на рис. 3.10 (см. с. 209). Проверьте работу этой программы на всех ребрах графа.

8.20. Определите, правильно ли оформлена ссылка на переменную `x` в каждом из следующих правил. Объясните ваши ответы.

а)
```
(defrule example-1
      (not (fact ?x))
      (test (> ?x 4))
      =>)
```

б)
```
(defrule example-2
   (not (fact ?x&:(> ?x 4)))
   =>)
```

в)
```
(defrule example-3
   (not (fact ?x))
   (fact ?y&:(> ?y ?x)))
   =>)
```

г)
```
(defrule example-4
   (not (fact ?x))
   (fact ?x&:(> ?x 4)))
   =>)
```

8.21. Перепишите программу для “мира блоков”, приведенную в разделе 7.23, таким образом, чтобы она могла переупорядочивать блоки, переходя из любого начального состояния блоков, сложенных в столбики, в любое целевое состояние сложенных блоков. Например, если начальное состояние блоков таково:

```
(stack A B C)
(stack D E F)
```

то одним из возможных целевых состояний может стать следующее:

```
(stack D C B)
(stack A)
(stack F E)
```

8.22. Напишите программу CLIPS, которая запрашивает у пользователя значения цветов, а затем выводит список всех государств, флаги которых содержат все указанные цвета. Цвета флагов различных стран перечислены в табл. 8.7.

8.23. Предположим, что дана следующая конструкция `deftemplate` с описанием множества:

```
(deftemplate set
   (multislot name)
   (multislot members))
```

Напишите одно или несколько правил, которые обеспечивают выполнение перечисленных ниже заданий.

а) Вычисление объединения двух указанных множеств после получения факта, в котором используется следующая конструкция `deftemplate`:

```
(deftemplate union
   (multislot set-1-name)
   (multislot set-2-name))
```

б) Вычисление пересечения двух указанных множеств после получения факта, в котором используется следующая конструкция `deftemplate`:

```
(deftemplate intersection
   (multislot set-1-name)
   (multislot set-2-name))
```

Таблица 8.7. Цвета флагов различных стран

Государство	Цвета флага
Соединенные Штаты Америки	Красный, белый и синий
Бельгия	Черный, желтый и красный
Польша	Белый и красный
Монако	Белый и красный
Швеция	Желтый и синий
Панама	Красный, белый и синий
Ямайка	Черный, желтый и зеленый
Колумбия	Желтый, синий и красный
Италия	Зеленый, белый и красный
Ирландия	Зеленый, белый и оранжевый
Греция	Синий и белый
Ботсвана	Синий, белый и черный

Следует отметить, что после выполнения операций объединения и пересечения не допускается наличие дублирующих элементов в объединении или пересечении множеств. Конечным результатом в заданиях а) и б) должен быть новый факт `set`, содержащий объединение или пересечение двух указанных множеств; после выполнения операции необходимо удалять и факт `union`, и факт `intersection`.

8.24. Напишите ряд правил для классификации силлогистических форм по модусам и фигурам. Например, следующая силлогистическая форма имеет типа EOI-3:

$$\begin{array}{l} \text{No } M \text{ is } P \\ \underline{\text{Some } M \text{ is not } S} \\ \therefore \text{Some } S \text{ is } P \end{array}$$

В качестве входных данных для правил должен служить единственный факт, представляющий большую посылку, меньшую посылку и заключение. Выходом программы должна быть выведенная на внешнее устройство информация о модусе и фигуре.

8.25. Напишите программу, которая будет считывать файл данных, содержащий список имен людей с указанием возрастов, и создавать новый файл, в котором содержится тот же список, отсортированный в порядке увеличения возрастов. Программа должна запрашивать имена обоих файлов, и входного, и выходного. Например, после обработки такого входного файла:

```
Linda A. Martin 43
Phyllis Sebesta 40
Robert Delwood 38
Jack Kennedy 39
Glen Steele 37
```

должен быть создан следующий выходной файл:

```
Glen Steele 37
Robert Delwood 38
Jack Kennedy 39
Phyllis Sebesta 40
Linda A. Martin 43
```

8.26. Напишите программу для вычисления стоимости 13 карт, находящихся на руках игрока в бридж, с помощью метода подсчета очков. Тузы стоят четыре очка; короли — три очка; дамы — два очка; а валеты — одно очко. Пустая масть (отсутствие карт одной масти) стоит три очка; масть с одной картой (одна карта определенной масти) — два очка; а масть с двумя картами (две карты определенной масти) — одно очко.

8.27. Напишите программу, которая указывает, какие действия должны быть предприняты по спасению человека, проглотившего яд. В программе должны присутствовать знания о следующих ядах: кислоты (такие как жидкость для удаления ржавчины и йодистый усилитель), щелочи (такие как аммиачная вода и отбеливатель) и горючие вещества (такие как бензин и скипидар). Все остальные яды должны быть зачислены в категорию “прочие”.

В случае отравления необходимо вызвать врача или бригаду скорой помощи, специализирующихся по ядам. При отравлении кислотами, щелочами и ядами прочих типов (но отличными от горючих веществ) необходимо снижать концентрацию яда, вынуждая пострадавшего пить жидкость, такую как воду или молоко. При попадании в желудок ядов прочих типов следует вызвать рвоту. Но в случае отравления кислотами, щелочами или горючими веществами вызывать рвоту не следует. Нельзя давать жидкость или вызывать рвоту, если пострадавший находится без сознания или у него конвульсии.

8.28. Напишите программу, которая после получения значений координат двух точек на плоскости определяет наклон прямой, проходящей через эти две точки. Программа должна выполнять проверку для определения того, что

координаты точек заданы числами и что одна и та же точка не указана дважды. Линии, направленные перпендикулярно горизонтальной оси, следует рассматривать как имеющие бесконечный наклон.

8.29. Косоугольным называется треугольник, который имеет три неравных стороны. Равнобедренный треугольник имеет две стороны одинаковой длины. Равносторонний треугольник имеет три стороны одинаковой длины. Напишите программу, которая после получения координат трех точек на плоскости, образующих вершины треугольника, определяет тип треугольника. В программе необходимо учитывать возможную ошибку округления (примите предположение, что две стороны равны, если разница между значениями их длины не превышает 0,00001). Проверьте разработанную вами программу по приведенным ниже данным о треугольниках.

а) Точки $(0,0)$, $(2,4)$ и $(6,0)$.

б) Точки $(1,2)$, $(4,5)$ и $(7,2)$.

в) Точки $(0,0)$, $(3,5.196152)$ и $(6,0)$.

8.30. Напишите программу для поиска решения задачи с ханойскими башнями, в которой необходимо переместить ряд колец, имеющих разный наружный диаметр и одинаковый внутренний диаметр, с одного колышка на другой колышек, ни разу не насаживая на колышек кольцо с большим наружным диаметром поверх кольца с меньшим наружным диаметром. Для насаживания колец используются три колышка. Входными данными для программы должно быть количество колец. В начальной конфигурации все кольца находятся на первом колышке, будучи расположенными в порядке уменьшения наружного диаметра от нижнего кольца к верхнему. Конечная цель состоит в перемещении всех колец с первого колышка на третий колышек.

8.31. Напишите программу, позволяющую определить цифровые значения букв, после подстановки которых следующая криптоарифметическая задача решается правильно. Каждой из букв H, O, C, U, S, P, R, E и T соответствует уникальная цифра от 0 до 9.

```
  HOCUS
+ POCUS
= PRESTO
```

8.32. Напишите программу, которая выдает рекомендации по инвестированию во взаимные фонды. В выводе программы должно быть указано, какие процентные доли денежных средств должны инвестироваться в фонды с фиксированным доходом (фонды, предназначенные главным образом для вложения капитала в облигации и привилегированные акции) или в акционерные фонды (характеризующиеся более высоким риском, но предоставляющие больший потенциальный доход). Процентные доли следует определять,

"оценивая", какой риск готов взять на себя инвестор, на основе полученных ответов на различные вопросы. Если инвестору 29 лет или меньше, к оценке следует добавить 4; 30–39 — добавить 3; 40–49 — добавить 2; 50–59 — добавить 1; 60 или больше — добавить 0. Если инвестору осталось до пенсии от 0 до 9 лет — добавить 0; 10–14 — добавить 1; 15–19 — добавить 2; 20–24 — добавить 3; 25 или больше — добавить 4. Если инвестор готов потерпеть убытки только в размере 5% или меньше — добавить к оценке 0; 6–10% — добавить 1; 11–15% — добавить 2; 16% или больше — добавить 3. Если инвестор очень хорошо разбирается в инвестициях и в работе фондовой биржи, добавить к оценке 4; если довольно хорошо осведомлен в этих вопросах, добавить к оценке 2; если не очень хорошо осведомлен, добавить к оценке 0. Если инвестор согласен взять на себя существенный риск ради более высокой возможной прибыли, добавить к оценке 4; если готов пойти на определенный риск, добавить 2; а если для него приемлем лишь небольшой риск, добавить 0. Если инвестор полагает, что цели по достижению определенного благосостояния, которые он поставил перед собой в связи с уходом на пенсию, будут достигнуты, учитывая его текущие доходы и активы, добавить к оценке 4; если эти цели скорее всего будут достигнуты, добавить 2; если эти цели вряд ли будут достигнуты, добавить 0. Если окончательная оценка больше 20 пунктов, то 100% инвестиций должно быть сделано в акционерные фонды; если 16–20 пунктов, то 80% должно быть сделано в акционерные фонды и 20% в фонды с фиксированным доходом; если 11–15 пунктов, то 60% должно быть сделано в акционерные фонды и 40% в фонды с фиксированным доходом; если 6–10 пунктов, то 40% должно быть сделано в акционерные фонды и 60% в фонды с фиксированным доходом; если от 0 до 5 пунктов, то 20% должно быть сделано в акционерные фонды и 80% в фонды с фиксированным доходом.

8.33. Модифицируйте программу, разработанную в результате решения задачи 7.12 (см. с. 619), таким образом, чтобы информация о звездах была представлена с использованием фактов. Вывод программы должен осуществлять в том же порядке: все звезды, имеющие указанный спектральный класс, все звезды, имеющие указанную величину, наконец, все звезды, соответствующие и спектральному классу, и величине, наряду с их расстоянием от Земли в световых годах.

Глава 9

Модульное проектирование, управление выполнением и эффективность правил

9.1 Введение

В настоящей главе приведено вводное описание целого ряда средств языка CLIPS, позволяющих упростить разработку и сопровождение экспертных систем. Атрибуты `deftemplate` обеспечивают принудительное соблюдение ограничений на значения применительно к значениям слотов конструкции `deftemplate`. Во время загрузки правила атрибуты ограничения конструкции `deftemplate` позволяют обнаруживать семантические ошибки, которые исключают возможность сопоставления с левой частью правила. Кроме того, как показано в этой главе, на основе понятия значимости, которое лежит в основе метода определения приоритета правил, а также фактов, представляющих знания о путях передачи управления, могут быть созданы эффективные методы управления исполнением программ CLIPS. Наряду с этим в данной главе рассматривается конструкция `defmodule`, позволяющая секционировать базу знаний и предоставляющая более явный метод управления исполнением программы. Дополнительно к этому в настоящей главе представлено много способов повышения эффективности экспертной системы, основанной на правилах, в которой используется rete-алгоритм сопоставления с шаблонами. Причем вначале дано описание причин, по которым требуется эффективный алгоритм сопоставления с шаблонами, и только после этого дается описание rete-алгоритма. Наконец, обсуждается несколько способов более эффективной формулировки правил.

9.2 Атрибуты deftemplate

В языке CLIPS предусмотрен целый ряд атрибутов слота, которые могут быть заданы при определении слотов конструкции deftemplate. Применение этих атрибутов позволяет упростить разработку и сопровождение экспертной системы и обеспечивает строгий контроль типов и проверку ограничений. К тому же обеспечивается возможность определить допустимые типы и значения, которые могут храниться в слоте, а для числовых значений может быть указан допустимый диапазон. Конструкции multislot позволяют указывать минимальное и максимальное количество полей, которые они могут содержать. Наконец, атрибут default предоставляет возможность определять заданное по умолчанию значение слота, которое будет использоваться, если значение соответствующего слота не задано в команде assert.

Атрибут type

Атрибут type определяет типы данных, которые могут храниться в слоте. Атрибут type имеет общий формат (type <type-specification>), в котором в качестве параметра <type-specification> может быть либо задана переменная ?VARIABLE, либо один или несколько символов SYMBOL, STRING, LEXEME, INTEGER, FLOAT, NUMBER, INSTANCE-NAME, INSTANCE-ADDRESS, INSTANCE, FACT-ADDRESS или EXTERNAL-ADDRESS. Если используется переменная ?VARIABLE, то слот может содержать данные любого типа (по умолчанию для всех слотов предусмотрен именно такой способ их применения). Если же используется одна или несколько символических спецификаций типа, применение слота ограничивается одним из указанных типов. Использование спецификации типа LEXEME эквивалентно заданию спецификаций SYMBOL и STRING. Использование спецификации типа NUMBER эквивалентно заданию спецификаций INTEGER и FLOAT, а применение спецификации типа INSTANCE эквивалентно заданию спецификаций INSTANCE-NAME и INSTANCE-ADDRESS.

Ниже приведена конструкция deftemplate с именем person, которая ограничивает значения, хранимые в слоте name, символами, и значения, хранимые в слоте age, целыми числами.

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER)))
```

После определения этой конструкции deftemplate система CLIPS автоматически предписывает применение заданных ограничений к любым атрибутам слотов. Например, как показано ниже, присваивание слоту age символа four, а не целого числа 4 приводит к возникновению ошибки.

```
CLIPS> (assert (person (name Fred Smith)
                       (age four)))↵
[CSTRNCHK1] A literal slot value found in the assert
command does not match the allowed types for slot age.
CLIPS>
```

В системе CLIPS осуществляется также проверка совместимости связывания переменных в левой и правой частях правила. Например, предположим, что в некотором правиле обновляется слот age с данными о возрасте некоторого лица после каждого дня его рождения. Конструкция deftemplate для управляющего факта, который показывает, что у рассматриваемого человека только что был день рождения, является следующей:

```
(deftemplate had-a-birthday
   (slot name (type STRING)))
```

Но допустимые типы для слотов name в этих двух конструкция deftemplate являются несовместимыми. Как показывает следующий диалог, попытка непосредственно сравнить значения двух указанных слотов приводит к возникновению ошибки:

```
CLIPS>
(defrule update-birthday
   ?f1 <- (had-a-birthday (name ?name))
   ?f2 <- (person (name ?name) (age ?age))
   =>
   (retract ?f1)
   (modify ?f2 (age (+ ?age 1))))↵
[RULECSTR1] Variable ?name in CE #2 slot name has
constraint conflicts which make the pattern unmatchable.
ERROR:
(defrule MAIN::update-birthday
   ?f1 <- (had-a-birthday (name ?name))
   ?f2 <- (person (name ?name) (age ?age))
   =>
   (retract ?f1)
   (modify ?f2 (age (+ ?age 1))))
CLIPS>
```

Слот name для факта had-a-birthday должен представлять собой строку, а слот name для факта person должен быть символом. Переменная ?name не может соответствовать обоим этим ограничениям, и поэтому проверка условия в левой части данного правила никогда не может завершиться успешно.

Статическая и динамическая проверка ограничений

В языке CLIPS предусмотрены два уровня проверки ограничений. Первый уровень, **статическая проверка ограничений**, применяется по умолчанию при выполнении в системе CLIPS синтаксического анализа выражения или конструкции. Использование этого уровня можно проиллюстрировать с помощью приведенных выше примеров нарушения ограничений атрибута `type`. Статическую проверку ограничений можно отменить, вызвав **функцию set-static-constraint-checking** и передав ей в качестве параметра символ `FALSE`. И наоборот, в случае вызова этой функции с символом `TRUE` статическая проверка ограничений активизируется. Функция возвращает значение, соответствующее применявшемуся перед ее выполнением режиму статической проверки ограничений (символ `FALSE`, если такая проверка была перед этим запрещена, и символ `TRUE` — в противном случае). Текущее состояние статической проверки ограничений можно определить, вызвав **функцию get-static-constraint-checking** (которая возвращает символ `TRUE`, если статическая проверка ограничений разрешена, и символ `FALSE` — в противном случае).

На этапе синтаксического анализа не всегда возможно определить все ошибки, связанные с ограничениями. Например, ниже приведено правило `create-person`, в котором переменные `?age` и `?name` могут быть связаны с недопустимыми значениями.

```
(defrule create-person
   =>
   (printout t "What is your name? ")
   (bind ?name (explode$ (readline)))
   (printout t "What is your age? ")
   (bind ?age (read))
   (assert (person (name ?name) (age ?age))))
```

Для ввода полного имени лица в виде строки используется функция `readline`, после чего функция `explode$` преобразует содержимое строки в многозначное значение, которое может быть помещено в слот `name`. Затем для ввода данных о возрасте рассматриваемого лица применяется функция `read`, которая записывает полученное значение в слот `age`. Но не исключена возможность, что в качестве обоих входных значений будут введены недопустимые данные. Например, как показывает следующий диалог, в качестве данных о возрасте лица может введен символ `four`:

```
CLIPS> (reset)↵
CLIPS> (run)↵
What is your name? Fred Smith↵
What is your age? four↵
```

```
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (person (name Fred Smith) (age four))
For a total of 2 facts.
CLIPS>
```

Здесь заслуживает внимания то, что такой же факт person, касающийся лица по имени Fred Smith, который перед этим вызвал нарушение ограничения и не был добавлен к списку фактов, теперь был добавлен успешно. Это связано с тем, что второй уровень проверки ограничений, предусмотренный в системе CLIPS, уровень **динамической проверки ограничений**, запрещен по умолчанию. Динамическая проверка ограничений применяется к фактам в то время, как действительно происходит их ввод в список фактов, что позволяет перехватывать ошибки, которые не могут быть обнаружены во время синтаксического анализа.

Динамическая проверка ограничений может быть разрешена или запрещена с помощью **функции set-dynamic-constraint-checking**, а текущее состояние динамической проверки ограничений можно определить с помощью **функции get-dynamic-constraint-checking**. Ниже приведен диалог, который показывает, какие действия предпринимаются в случае нарушения ограничения, если разрешена динамическая проверка ограничений. Факт person с данными о человеке по имени Fred Smith все равно вводится в список фактов, но обнаруживается нарушение ограничения и выполнение правил прекращается.

```
CLIPS> (set-dynamic-constraint-checking TRUE)↵
FALSE
CLIPS> (reset)↵
CLIPS> (run)↵
What is your name? Fred Smith↵
What is your age? four↵
[CSTRNCHK1] Slot value (Fred Smith) found in fact
f-1 does not match the allowed types for slot age.
[PRCCODE4] Execution halted during the actions of
defrule create-person.
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (person (name Fred Smith) (age four))
For a total of 2
facts.
CLIPS>
```

Атрибуты допустимого значения

Язык CLIPS позволяет не только регламентировать перечень допустимых типов с помощью атрибута `type`, но и дает возможность задавать список допустимых значений для конкретного типа. Например, если в конструкцию `deftemplate` с именем `person` дополнительно вводится слот `gender` (пол), может быть реализована возможность ограничить перечень допустимых символов для этого слота значениями `male` и `female`, как показано ниже.

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER))
   (slot gender (type SYMBOL)
   (allowed-symbols male female)))
```

В языке CLIPS предусмотрено восемь различных атрибутов допустимого значения: **allowed-symbols**, **allowed-strings**, **allowed-lexemes**, **allowed-integers**, **allowed-floats**, **allowed-numbers**, **allowed-instance-names** и **allowed-values**. За каждым из этих атрибутов должно следовать либо обозначение переменной `?VARIABLE` (которое указывает на то, что любые значения заданного типа являются допустимыми), либо список значений этого типа, следующего за префиксом `allowed-`. Например, за атрибутом `allowed-lexemes` должно следовать либо обозначение `?VARIABLE`, либо список символов и (или) строк. По умолчанию атрибут допустимого значения для слотов имеет вид `(allowed-values ?VARIABLE)`.

Следует отметить, что атрибуты допустимого значения не ограничивают состав допустимых типов слота. Например, конструкция `(allowed-symbols male female)` не налагает такого ограничения, чтобы типом слота `gender` был символ. Эта конструкция указывает, что если значением слота является символ, то им должен быть один из двух символов — либо `male`, либо `female`. Если бы атрибут `(type SYMBOL)` был удален, то допустимым значением для слота `gender` были бы любая строка, целое число или число с плавающей точкой.

Атрибут `allowed-values` можно использовать, чтобы полностью ограничить множество допустимых значений для слота заданным списком. Например, после того как конструкция `deftemplate` с именем `person` будет заменена следующей, в результате произойдет то, что пределы допустимых типов для слота `gender` ограничатся символами:

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER))
   (slot gender (allowed-values male female)))
```

Атрибут range

Атрибут range позволяет задавать минимальные и максимальные допустимые числовые значения. Атрибут range имеет общий формат (range <lower-limit> <upper-limit>), в котором параметры <lower-limit> и <upper-limit> представляют собой либо обозначение ?VARIABLE, либо числовое значение. Терм <lower-limit> указывает минимальное значение для слота, а терм <upper-limit> задает максимальное значение для слота. Обозначение ?VARIABLE указывает, что не задано либо минимальное, либо максимальное значение (в зависимости от того, находится ли оно на первом или на втором месте). Например, чтобы предотвратить возможность помещать в слот отрицательные значения, слот age в конструкции deftemplate с именем person можно изменить следующим образом:

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER) (range 0 ?VARIABLE)))
```

А если бы потребовалось явно сформулировать предположение, что никто не сможет прожить больше 125 лет, и ввести тем самым дополнительное ограничение, то можно было бы заменить это определение атрибута range определением (range 0 125). Как и в случае атрибутов допустимого значения, атрибут range не ограничивает тип значения слота таким образом, что если в нем заданы числа, то и тип должен быть числовым. Этот атрибут ограничивает только допустимые числовые значения слота заданным диапазоном, если значение слота является числовым. По умолчанию в качестве атрибута range для слотов применяется (range ?VARIABLE ?VARIABLE).

Атрибут cardinality

Атрибут cardinality позволяет задавать минимальное и максимальное количество значений, которые могут храниться в конструкции multislot. Атрибут cardinality имеет общий формат (cardinality <lower-limit> <upper-limit>), в котором термы <lower-limit> и <upper-limit> представляют собой либо обозначение ?VARIABLE, либо положительное целое число. Терм <lower-limit> показывает минимальное количество значений, которое может содержаться в слоте, а терм <upper-limit> позволяет указать максимально допустимое количество значений, содержащихся в слоте. Обозначение ?VARIABLE показывает, что не задано либо минимальное, либо максимальное количество значений, которое может быть указано в слоте (в зависимости от того, находится это обозначение на первом или втором месте). По умолчанию атрибут cardinality для любого многозначного значения для конструкции multislot имеет вид (cardinality ?VARIABLE ?VARIABLE). Ниже приведена кон-

струкция deftemplate, которая может использоваться для представления состава волейбольной команды, в которую входят сотрудники компании; в этой команде должно быть шесть игроков, а количество запасных игроков может достигать двух. Обратите внимание на то, что к каждому значению, содержащемуся в конструкции multislot, применяются ограничения типа допустимого значения и диапазона.

```
(deftemplate volleyball-team
   (slot name (type STRING))
   (multislot players (type STRING)
                      (cardinality 6 6))
   (multislot alternates (type STRING)
                         (cardinality 0 2)))
```

Атрибут default

В предыдущих главах каждый факт deftemplate, вводимый в список фактов, всегда имел явно заданное значение для каждого слота. Но часто бывает удобно автоматически сохранять в слоте указанное значение, если в команде assert явно не задано какое-либо значение. Возможность задавать применяемое по умолчанию значение обеспечивается **атрибутом default**. Атрибут default имеет общий формат (default <default-specification>), в котором терм <default-specification> может представлять собой обозначение ?DERIVE или ?NONE, единственное выражение (для однозначного слота), либо от нуля и больше выражений (для многозначного слота).

Если в атрибуте default задано обозначение ?DERIVE, то для данного слота должно быть выведено логическим путем определенное значение, которое соответствует всем атрибутам слота. Если для слота не задан атрибут default, то предполагается, что этот атрибут имеет вид (default ?DERIVE). Применительно к однозначному слоту это означает, что выбирается значение, которое удовлетворяет всем требования к атрибутам типа, диапазона и допустимого значения для этого слота. Выведенным логическим путем и заданным по умолчанию значением для многозначного слота становится список идентичных значений, которые имеют минимально допустимую кардинальность для данного слота (по умолчанию равную нулю). Если в заданном по умолчанию значении для многозначного слота содержится одно или несколько значений, то каждое из этих значений должно соответствовать атрибутам типа, диапазона и допустимого значения для этого слота. Ниже приведен пример значений, выведенных логическим путем.

```
CLIPS> (clear)↵
CLIPS>
(deftemplate example
   (slot a)
```

```
   (slot b (type INTEGER))
   (slot c (allowed-values red green blue))
   (multislot d)
   (multislot e (cardinality 2 2)
                (type FLOAT)
                (range 3.5 10.0)))↵
CLIPS> (assert (example))↵
<Fact-0>
CLIPS> (facts)↵
f-0 (example (a nil)
             (b 0)
             (c red)
             (d)
             (e 3.5 3.5))
For a total of 1 fact.
CLIPS>
```

Система CLIPS гарантирует только то, что выведенное логическим путем и заданное по умолчанию значение для слота удовлетворяет атрибутам ограничения для этого слота. Иными словами, программа не должна зависеть от конкретных производных значений (таких как символ типа `nil` для слота `a` или целое число `0` для слота `b` в предыдущем примере), помещаемых в слоты. Если же программа зависит от какого-то конкретного применяемого по умолчанию значения, то должно использоваться некоторое выражение с атрибутом `default` (сведения по этой теме приведены ниже).

Если в атрибуте `default` задано обозначение `?NONE`, то необходимо предусматривать применение некоторого значения для данного слота во время ввода факта в список фактов. Иными словами, в таком случае применяемое по умолчанию значение не предусмотрено. В качестве примера можно привести следующий диалог:

```
CLIPS> (clear)↵
CLIPS>
(deftemplate example
   (slot a)
   (slot b (default ?NONE)))↵
CLIPS> (assert (example))↵
[TMPLTRHS1] Slot b requires a value because of its
(default ?NONE) attribute.
CLIPS> (assert (example (b 1)))↵
<Fact-0>
CLIPS> (facts)↵
```

```
f-0      (example (a nil) (b 1))
For a total of 1 fact.
CLIPS>
```

Если используется одно или несколько выражений с атрибутом `default`, то во время синтаксического анализа слота эти выражения вычисляются и полученное значение сохраняется в слоте каждый раз, когда значение остается не заданным в команде `assert`. Атрибут `default` для однозначного слота должен содержать одно и только одно выражение. Если же в атрибуте `default` для многозначного слота выражения не заданы, то для применяемого по умолчанию значения используется многозначная величина с количеством значений, равным нулю. В противном случае возвращаемые значения всех выражений группируются вместе для формирования одного многозначного значения. Ниже приведен пример, в котором применяются выражения с атрибутом `default`.

```
CLIPS> (clear)↵
CLIPS>
(deftemplate example
   (slot a (default 3))
   (slot b (default (+ 3 4)))
   (multislot c (default a b c))
   (multislot d (default (+ 1 2) (+ 3 4))))↵
CLIPS> (assert (example))↵
<Fact-0>
CLIPS> (facts)↵
f-0      (example (a 3) (b 7) (c a b c) (d 3 7))
For a total of 1 fact.
CLIPS>
```

Атрибут `default-dynamic`

Если используется атрибут `default`, то применяемое по умолчанию значение для слота определяется во время синтаксического анализа объявления слота. Предусмотрена также возможность обеспечить выработку применяемого по умолчанию значения во время ввода в список фактов того факта, в котором будет использоваться это значение, предусмотренное по умолчанию. Для выполнения такой задачи используется **атрибут default-dynamic**. Если значение слота, в котором применяется атрибут `default-dynamic`, остается не заданным в команде `assert`, то вычисляется выражение, заданное с помощью атрибута `default-dynamic`, которое затем используется в качестве значения слота.

В качестве примера рассмотрим, какие проблемы могут быть связаны с решением такой задачи, когда требуется удалить некоторые факты по истечении определенного времени. Прежде всего необходимо найти некоторый способ, поз-

воляющий узнать, когда произошла вставка интересующих нас фактов в список фактов. В данном случае для проставления в фактах отметки времени создания фактов будет использоваться **функция time**, предусмотренная в языке CLIPS. Эта функция возвращает данные о количестве секунд, истекших с начала отсчета времени, зависящего от системы. Как таковое возвращаемое значение функции time не имеет смысла. Оно применимо только при сравнении с другими значениями, возвращаемыми этой функцией. В рассматриваемом примере будет использоваться следующая конструкция deftemplate, в которой содержится слот creation-time, предназначенный для хранения значения времени создания, и слот value, в котором хранится значение, связанное с фактом:

```
(deftemplate data
   (slot creation-time (default-dynamic (time)))
   (slot value))
```

После создания каждого факта data и определения того, что слот creation-time не задан, вызывается функция time и возвращаемое ею значение сохраняется в слоте creation-time следующим образом:

```
CLIPS> (watch facts)↵
CLIPS> (assert (data (value 3)))↵
==> f-0     (data (creation-time 12002.45)
                  (value 3))
<Fact-0>
CLIPS> (assert (data (value b)))↵
==> f-1     (data (creation-time 12010.25)
                  (value b))
<Fact-1>
CLIPS> (assert (data (value c)))↵
==> f-2     (data (creation-time 12018.65)
                  (value c))
<Fact-2>
CLIPS>
```

Если принято предположение, что факт current-time вводится в список фактов и обновляется другими правилами для включения в него текущего системного времени, то следующее правило позволяет извлекать факты data, вставка которых была выполнена минуту назад:

```
(defrule retract-data-facts-after-one-minute
   ?f <- (data (creation-time ?t1))
   (current-time ?t2)
   (test (> (- ?t2 ?t1) 60))
```

```
   =>
   (retract ?f))
```

Обратите внимание на то, что замена правила `retract-data-facts-after-one-minute` следующим правилом не позволяет получить такие же результаты:

```
(defrule retract-data-facts-after-one-minute
   ?f <- (data (creation-time ?t1))
   (test (> (- (time) ?t1) 60))
   =>
   (retract ?f))
```

В этом правиле значение функции `time` проверяется в условном элементе `test` только при согласовании первого шаблона с фактом `data`. А поскольку возвращаемое значение может представлять собой примерно такое же значение времени, как и значение в слоте `creation-time`, то проверка правила не будет завершаться успешно. С другой стороны, система CLIPS не будет непрерывно выполнять повторную проверку условных элементов `test` для определения того, приводит ли их вычисление к получению других значений; они проверяются, только если возникают изменения в предшествующих им условных элементах. Именно поэтому приходится обновлять факт `current-time` в первоначальной версии правила, чтобы периодически повторно проверялся данный условный элемент `test`.

Конфликтующие атрибуты слота

Система CLIPS не позволяет задавать для слота конфликтующие атрибуты. Например, заданное по умолчанию значение слота должно соответствовать типу слота, определению `allowed-...`, а также атрибутам `range` и `cardinality`. Если задан атрибут `allowed-...`, то тип, связанный с этим атрибутом, должен соответствовать атрибуту `type` слота. С атрибутом `range` не могут использоваться атрибуты `allowed-numbers`, `allowed-integers` и `allowed-floats`.

9.3 Значимость

До сих пор для неявного управления выполнением программ использовались управляющие факты. Кроме того, в системе CLIPS предусмотрены два явных метода управления исполнением правил, основанных на применении понятия значимости и модулей. Метод управления исполнением правил с использованием модулей будет рассматриваться ниже в этой главе, а в данном разделе речь идет о применении понятия значимости. **Ключевое слово salience** (значимость) позволяет явно задавать приоритеты правил. При обычных условиях рабочий список

правил действует по принципу стека. Это означает, что запускается активизированное правило, которое было помещено в рабочий список правил самым последним по времени. С другой стороны, средства определения значимости позволяют оставлять в верхней части рабочего списка правил более важные правила независимо от того, когда произошло введение этих правил, а правила с более низкой значимостью задвигаются в рабочем списке правил ниже правил с более высокой значимостью.

Значимость задается с помощью числового значения, изменяющегося в пределах от наименьшего значения, равного `-10000`, до наибольшего, равного `10000`. Если некоторому правилу программистом не было явно присвоено значение значимости, то система CLIPS принимает предположение, что значимость этого правила равна `0`. Заслуживает внимания то, что значимость, равная `0`, занимает среднее положение между максимальным и минимальным значениями значимости. Значение значимости, равное `0`, означает не то, что правило не имеет значимости, а, скорее, то, что данное правило находится на промежуточном уровне приоритета. Вновь активизированное правило помещается в рабочий список правил перед всеми правилами с равной или меньшей значимостью и после всех правил с большей значимостью.

Одно из направлений использования значимости состоит в том, чтобы принудительно обеспечивать запуск правил в определенной последовательности. Рассмотрим следующее множество правил, в которых значения значимости не объявлены:

```
(defrule fire-first
   (priority first)
   =>
   (printout t "Print first" crlf))
(defrule fire-second
   (priority second)
   =>
   (printout t "Print second" crlf))
(defrule fire-third
   (priority third)
   =>
   (printout t "Print third" crlf))
```

Порядок, в котором будет осуществляться запуск этих правил, зависит от того, в каком порядке в список фактов вводились те факты, которые соответствуют условным элементам в левых частях правил. Например, если правила уже введены в систему, то ввод следующих команд приведет к формированию приведенного ниже вывода.

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (assert (priority first))↵
<Fact-1>
CLIPS> (assert (priority second))↵
<Fact-2>
CLIPS> (assert (priority third))↵
<Fact-3>
CLIPS> (run)↵
Print third
Print second
Print first
CLIPS>
```

Обратите внимание на то, какова последовательность операторов вывода. Вначале на устройстве вывода появляется строка "`Print third`", затем "`Print second`" и наконец "`Print first`". Первый факт `(priority first)` активизирует правило `fire-first`. После ввода в список правил второго факта он активизирует правило `fire-second`, которое задвигается в стек поверх активизации, соответствующей правилу `fire-first`. Наконец происходит ввод в список фактов третьего факта, и его активизированное правило, `fire-third`, задвигается в стек поверх активизации для правила `fire-second`.

В системе CLIPS приоритеты правил с равной значимостью, которые активизируются в результате сопоставления с разными шаблонами, определяются с учетом порядка фактов в стеке. А запуск правил из рабочего списка правил происходит от вершины стека в направлении вниз. Поэтому вначале происходит запуск правила `fire-third`, поскольку оно находится в верхней части стека, затем запускается правило `fire-second` и наконец — правило `fire-first`. Если бы порядок вставки фактов был обратным, то порядок, в котором происходит запуск правил, также был бы обратным. В этом можно убедиться, ознакомившись со следующим выводом:

```
CLIPS> (reset)↵
CLIPS> (assert (priority third))↵
<Fact-1>
CLIPS> (assert (priority second))↵
<Fact-2>
CLIPS> (assert (priority first))↵
<Fact-3>
CLIPS> (run)↵
Print first
Print second
```

```
Print third
CLIPS>
```

Необходимо учитывать также еще одно важное соображение: если под действием одного и того же факта активизируется два или несколько правил с одинаковой значимостью, то невозможно гарантировать определенный порядок помещения этих правил в рабочий список правил.

Для принудительного обеспечения запуска правил в последовательности `fire-first`, `fire-second` и `fire-third`, независимо от того, в каком порядке происходил ввод фактов, активизирующих эти правила, в список фактов, можно использовать значимость. Такой подход может быть осуществлен путем объявления значений значимости, например, следующим образом:

```
(defrule fire-first
   (declare (salience 30))
   (priority first)
   =>
   (printout t "Print first" crlf))
(defrule fire-second
   (declare (salience 20))
   (priority second)
   =>
   (printout t "Print second" crlf))
(defrule fire-third
   (declare (salience 10))
   (priority third)
   =>
   (printout t "Print third" crlf))
```

В этом случае, независимо от того, в каком порядке будет происходить ввод в список фактов рассматриваемых фактов `priority`, рабочий список правил всегда будет иметь одну и ту же упорядоченность. Выполнение команды `agenda` после ввода фактов `priority` в список фактов приведет к получению следующего вывода:

```
CLIPS> (reset)↵
CLIPS> (assert (priority second)
                      (priority first)
                      (priority third))↵
<Fact-3>
CLIPS> (agenda)↵
30       fire-first: f-2
20       fire-second: f-1
10       fire-third: f-3
```

```
For a total of 3 activations.
CLIPS>
```

Обратите внимание на то, как были переупорядочены правила в рабочем списке правил согласно приоритетам и благодаря использованию значений значимости. После вызова программы на выполнение порядок запуска правил будет всегда одинаковым: `fire-first`, `fire-second`, `fire-third`.

9.4 Фазы и управляющие факты

Определению понятия экспертной системы, основанной на правилах, в наибольшей степени соответствует такая система, в которой правила действуют оппортунистически (иначе говоря, подчиняясь обстоятельствам), т.е. вызываются в любых обстоятельствах, когда они являются применимыми. Но в большинстве экспертных систем приходится также учитывать некоторые процедурные аспекты. Например, в программе Sticks имеются различные правила, применимость которых зависит от того, принадлежит ли очередность хода игроку-человеку или компьютеру. Управление этой программой осуществляется с помощью фактов, которые указывают, кто должен сделать следующий ход. Такие управляющие факты позволяют включать в состав правил, относящихся к рассматриваемой области знаний, информацию о структуре управления программой. Но такой подход имеет определенный недостаток: знания о том, как организовано управление правилами, смешиваются со знаниями о том, как играть в игру Sticks. В случае программы Sticks указанный недостаток не имеет существенного значения, поскольку эта программа невелика. А что касается программ, состоящих из сотен или тысяч правил, то смешивание знаний предметной области и управляющих знаний приводит к тому, что разработка и сопровождение существенно усложняются.

В качестве примера рассмотрим проблему осуществления этапов **обнаружения неисправностей**, **локализации неисправностей** и **возобновления нормальной работы** в такой системе, как электронное устройство. Обнаружение неисправностей — это процесс, позволяющий признать, что электронное устройство не функционирует должным образом. Локализация — это процесс определения того, какие компоненты устройства вызвали нарушение в работе. А возобновление нормальной работы — это процесс определения этапов, необходимых для устранения нарушения в работе (если это возможно). Вообще говоря, в экспертной системе, предназначенной для решения задач рассматриваемого типа, должны быть одни правила, позволяющие определить, имеет ли место неисправность, другие правила, позволяющие локализовать источник неисправности, а также еще одна категория правил, позволяющих найти способ возобновления нормальной работы после устранения возникшей неисправности. Затем система должна возвратиться в начало цикла, состоящего из этих трех этапов. Пример того, как должно быть

организовано управление ходом выполнения этапов в системе рассматриваемого типа, приведен на рис. 9.1.

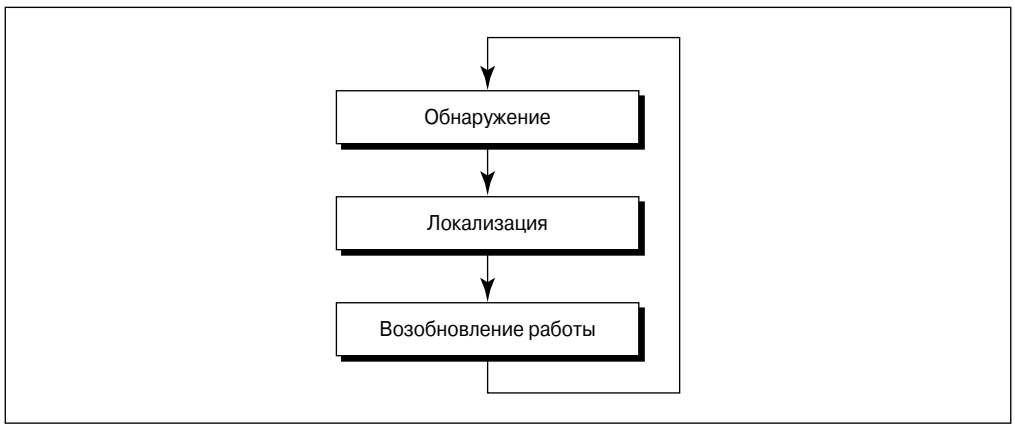


Рис. 9.1. Различные фазы решения задачи обнаружения неисправности, локализации неисправности и возобновления нормальной работы

Реализация способа передачи управления в этой системе может быть осуществлена с использованием по крайней мере четырех подходов. В первых трех подходах используется значимость; эти подходы рассматриваются в настоящем разделе. Четвертый подход, в котором применяются модули, будет рассматриваться ниже в данной главе.

Первый подход к реализации возможностей передачи управления состоит в том, чтобы представить управляющие знания непосредственно в правилах. Например, правила, применяемые в фазе обнаружения неисправностей, могут включать правила, указывающие, когда должен быть осуществлен переход в фазу локализации. В таком случае каждой группе правил присваивается шаблон, указывающий, в какой фазе являются применимыми эти правила. Но такой метод имеет два недостатка. Во-первых, как уже было сказано, управляющие знания вкладываются в правила, которые представляют знания проблемной области, в результате чего эти правила становятся более сложными для понимания. Во-вторых, не всегда легко определить, когда завершилась некоторая фаза. В связи с этим, вообще говоря, требуется предусматривать определенное правило, применимое, только если произошел запуск всех других правил.

Второй подход состоит в том, что для упорядочения правил используется понятие значимости (рис. 9.2). Но этот подход также имеет два существенных недостатка. Во-первых, управляющие знания по-прежнему вкладываются в правила с применением значимости. Во-вторых, этот подход не гарантирует, что исполнение правил будет происходить в правильном порядке. Безусловно, правила фазы обнаружения будут всегда активизироваться перед правилами фазы локализации.

Но, после того как начнется запуск правил фазы локализации, они могут активизировать правила фазы обнаружения и немедленный запуск последних из-за их более высокой значимости.

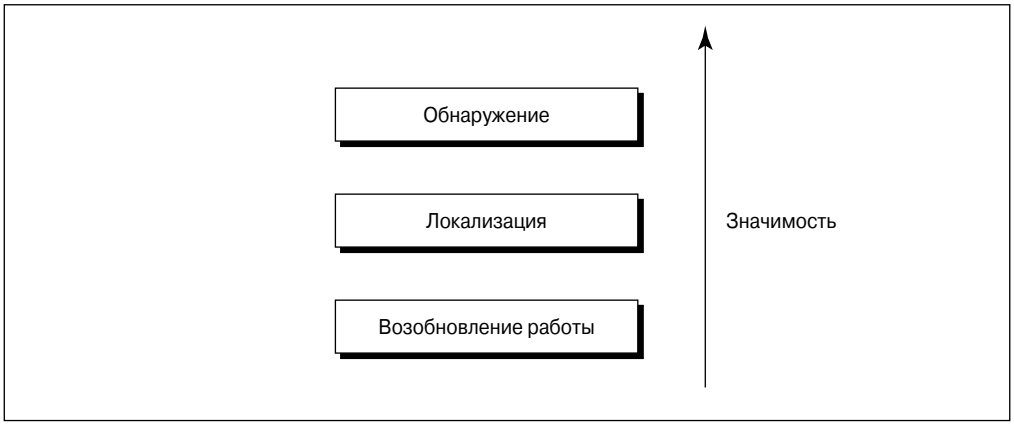


Рис. 9.2. Подход, в котором правилам различных фаз присваиваются разные значения значимости

Третий и лучший подход к управлению потоком исполнения правил состоит в том, чтобы отделить управляющие знания от знаний в проблемной области (рис. 9.3). При использовании этого подхода в каждом правиле предусматривается управляющий шаблон, который показывает, в какой фазе это правило применимо. Затем формулируются управляющие правила для передачи управления между различными фазами, как показано ниже.

```
(defrule detection-to-isolation
   (declare (salience -10))
   ?phase <- (phase detection)
   =>
   (retract ?phase)
   (assert (phase isolation)))
(defrule isolation-to-recovery
   (declare (salience -10))
   ?phase <- (phase isolation)
   =>
   (retract ?phase)
   (assert (phase recovery)))
(defrule recovery-to-detection
   (declare (salience -10))
   ?phase <- (phase recovery)
   =>
```



Рис. 9.3. Отделение экспертных знаний от управляющих

```
   (retract ?phase)
   (assert (phase detection)))
```

Затем каждому из правил, применимому в конкретной фазе, присваивается управляющий шаблон, который позволяет проверить наличие соответствующего управляющего факта. Например, правило фазы возобновления нормальной работы может выглядеть следующим образом:

```
(defrule find-fault-location-and-recovery
   (phase recovery)
   (recovery-solution switch-device
   ?replacement on)
   =>
   (printout t "Switch device" ?replacement "on"
               crlf))
```

Иерархия значимостей представляет собой описание значений значимости, используемых в экспертной системе. Каждый уровень в иерархии значимостей соответствует конкретному множеству правил, всем элементам которого присваивается одинаковая значимость. Если правилам, используемым в фазах обнаружения неисправностей, локализации и возобновления нормальной работы, присваивается применяемое по умолчанию значение значимости, равное нулю, то формируется иерархия значимостей (рис. 9.4). Обратите внимание на то, что правило `detection-to-isolation` будет находиться в рабочем списке правил до тех пор, пока факт `(phase detection)` присутствует в списке фактов. Но

правило detection-to-isolation имеет более низкую значимость по сравнению с правилами фазы обнаружения неисправностей, поэтому его запуск не произойдет до тех пор, пока все правила фазы обнаружения не получат возможность запуска. Ниже приведен вывод, в котором показан пример прогона трех описанных ранее управляющих правил.

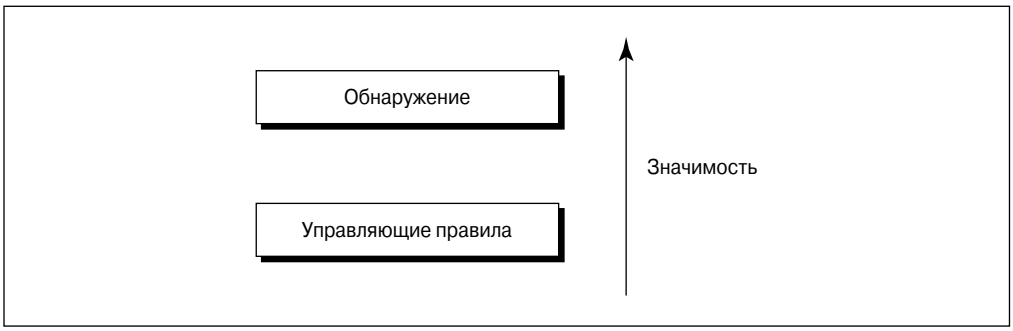


Рис. 9.4. Иерархия значимостей, создаваемая при использовании экспертных и управляющих правил

```
CLIPS> (reset)↵
CLIPS> (assert (phase detection))↵
<Fact-1>
CLIPS> (watch rules)↵
CLIPS> (run 10)↵
FIRE   1 detection-to-isolation: f-1
FIRE   2 isolation-to-recovery: f-2
FIRE   3 recovery-to-detection: f-3
FIRE   4 detection-to-isolation: f-4
FIRE   5 isolation-to-recovery: f-5
FIRE   6 recovery-to-detection: f-6
FIRE   7 detection-to-isolation: f-7
FIRE   8 isolation-to-recovery: f-8
FIRE   9 recovery-to-detection: f-9
FIRE  10 detection-to-isolation: f-10
CLIPS>
```

Следует отметить, что в данном примере происходит запуск управляющих правил одного за другим, поскольку отсутствуют правила со знаниями в проблемной области, которые могли бы быть применены в каждой из рассматриваемых фаз. А если бы такие правила были предусмотрены, то правила со знаниями в проблемной области использовались бы в качестве активизированных правил во время прохождения соответствующих фаз.

Для формулировки приведенных выше управляющих правил можно применить более обобщенную форму записи с конструкцией deffacts и единственным правилом:

```
(deffacts control-information
   (phase detection)
   (phase-after detection isolation)
   (phase-after isolation recovery)
   (phase-after recovery detection))
(defrule change-phase
   (declare (salience -10))
   ?phase <- (phase ?current-phase)
   (phase-after ?current-phase ?next-phase)
   =>
   (retract ?phase)
   (assert (phase ?next-phase)))
```

Еще один вариант состоит в том, что эти правила могут быть записаны с использованием циклически осуществляемой последовательности фаз:

```
(deffacts control-information
   (phase detection)
   (phase-sequence isolation recovery detection))
(defrule change-phase
   (declare (salience -10))
   ?phase <- (phase ?current-phase)
   ?list <- (phase-sequence ?next-phase
   $?other-phases)
   =>
   (retract ?phase ?list)
   (assert (phase ?next-phase))
   (assert (phase-sequence ?other-phases
   ?next-phase)))
```

К такой иерархии значимостей можно легко добавить дополнительные уровни. Пример иерархии с двумя дополнительными уровнями показан на рис. 9.5. Ограничивающие правила представляют собой правила, позволяющие обнаруживать недопустимые или непроизводительные состояния, которые могут возникнуть в экспертной системе. В частности, экспертная система, применяемая для составления графиков, в соответствии с которыми люди выполняют различные задания, может выработать график с нарушением некоторого ограничения. Но ограничивающие правила смогут немедленно устранить нарушения в графике, не позволяя продолжить работу над графиком с помощью правил с более низкой значимостью. В качестве еще одного примера можно указать, что пользователь

может ввести в ответ на ряд вопросов ряд допустимых значений, но программа выработает недопустимое значение. Для обнаружения подобных нарушений могут использоваться ограничивающие правила.

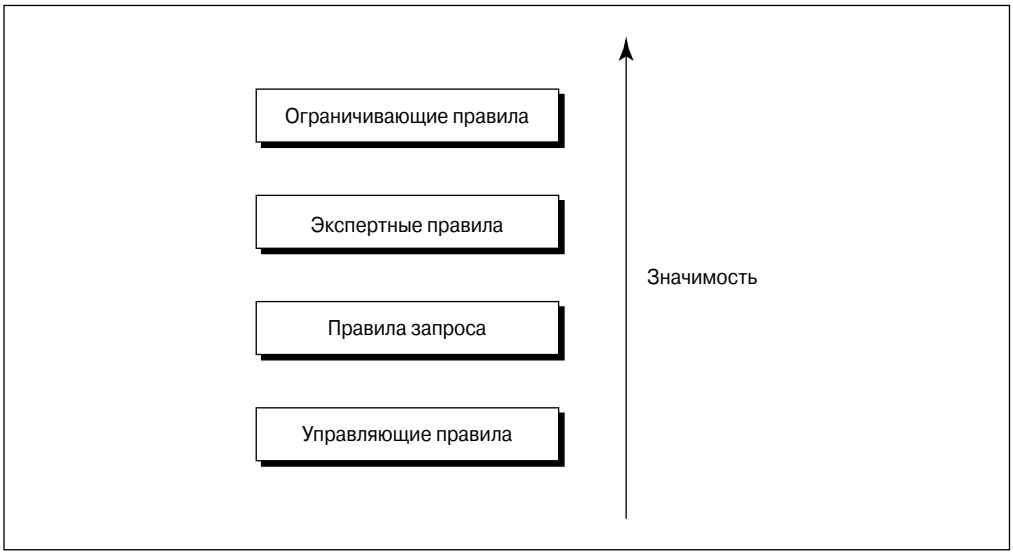


Рис. 9.5. Иерархия значимостей с четырьмя уровнями

Правила запроса (см. рис. 9.5) представляют собой правила, с помощью которых пользователю могут быть заданы конкретные вопросы, чтобы можно было оказать помощь экспертной системе при определении ответа. Эти правила имеют более низкую значимость, чем экспертные правила, поскольку нежелательно задавать пользователю такие вопросы, ответы на которые можно определить с помощью экспертной системы. Поэтому запуск правил запроса происходит, только если с помощью экспертных правил невозможно получить логическим путем дополнительную информацию.

9.5 Неправильное употребление подхода на основе значимости

Безусловно, значимость представляет собой мощное средство управления выполнением программы, но при его использовании можно легко допустить ошибку. Особенно злоупотребляют применением значений значимости те, кто только приступает к изучению программирования на основе правил, поскольку стремятся обеспечить явный контроль над выполнением с помощью значимости. Формируемая при этом программа в большей степени напоминает программы, созданные

по принципам процедурного программирования (к которому привыкли эти программисты), когда операторы программы выполняются последовательно.

В результате злоупотребления подходом на основе значимости создаются плохо закодированные программы. Основным преимуществом программирования на основе правил является то, что программисту не приходится заботиться об управлении выполнением программы. Поэтому правильно спроектированная программа на основе правил действует в естественном режиме выполнения, который позволяет машине логического вывода управлять запуском правил оптимальным образом.

Значимость следует использовать главным образом как механизм определения порядка, в котором должен происходить запуск правил. При этом подразумевается, что правило, помещенное в рабочий список правил, вообще говоря, когда-либо все равно будет запущено. Это означает, что значимость не следует использовать как метод выбора единственного правила из группы правил, поскольку для выражения критериев выбора могут применяться шаблоны. Кроме того, значимость не следует использовать как способ "быстрого исправления" дефектов программы, чтобы обеспечить запуск правил в надлежащем порядке.

Вообще говоря, любое значение значимости, применяемое в правиле, должно соответствовать одному из уровней в иерархии значимостей экспертной системы. То, что диапазон значений значимости составляет от -10000 до 10000, немного сбивает с толку. При разработке программы для любой экспертной системы редко приходится использовать больше семи значений значимости, а в большинстве успешно запрограммированных экспертных систем применяется не больше трех или четырех значений значимости. Настоятельно рекомендуется, чтобы программисты при создании крупных экспертных систем применяли для управления потоком выполнения модули (как описано в следующем разделе) и чтобы количество используемых значений значимости не превышало двух или трех.

В качестве примера того, как можно обойтись без применения значимости, ниже приведено простое множество правил, позволяющее определить, в каких клетках следует ставить отметку в игре в крестики-нолики. Правила перечислены в том порядке, в каком они должны использоваться.

```
IF a winning square is open, THEN take it.
IF a blocking square is open, THEN take it.
IF a square is open, THEN take it.
```

Если факт `choose-move` показывает, что должен быть сделан ход, а факты `open-square` позволяют прийти к выводу, что открыт выигрывающий (`winning`), блокирующий (`blocking`), средний (`middle`), угловой (`corner`) или боковой (`side`) квадрат, то выбор подходящего хода осуществляется с помощью следующих правил:

```
(defrule pick-to-win
   (declare (salience 10))
   ?phase <- (choose-move)
   (open-square win)
   =>
   (retract ?phase)
   (assert (move-to win)))
(defrule pick-to-block
   (declare (salience 5))
   ?phase <- (choose-move)
   (open-square block)
   =>
   (retract ?phase)
   (assert (move-to block)))
(defrule pick-any
   ?phase <- (choose-move)
   (open-square ?any&corner | middle | side)
   =>
   (retract ?phase)
   (assert (move-to ?any)))
```

Следует отметить, что если доступны квадраты больше чем одного типа, то в рабочий список правил помещаются все три правила. А после запуска правила `pick-to-win`, `pick-to-block` или `pick-any` извлечение управляющего факта приводит к удалению из рабочего списка правил всех остальных правил. Таким образом, все три правила очень тесно взаимосвязаны. Поэтому невозможно определить назначение каждого из этих правил, не рассматривая их все вместе. Таким образом, нарушается фундаментальный принцип программирования на основе правил. Любое правило в максимально возможной степени должно представлять какую-то полностью выраженную эвристику. В данном случае для выражения неявных связей между этими правилами используется значимость, тогда как связи между правилами могут быть показаны явно с помощью дополнительных шаблонов, заданных в правилах. Указанные правила могут быть переформулированы без определения значимости следующим образом:

```
(defrule pick-to-win
   ?phase <- (choose-move)
   (open-square win)
   =>
   (retract ?phase)
   (assert (move-to win)))
(defrule pick-to-block
```

```
   ?phase <- (choose-move)
   (open-square block)
   (not (open-square win))
   =>
   (retract ?phase)
   (assert (move-to block)))
(defrule pick-any
   ?phase <- (choose-move)
   (open-square ?any&corner | middle | side)
   (not (open-square win))
   (not (open-square block))
   =>
   (retract ?phase)
   (assert (move-to ?any)))
```

Ввод в эти правила дополнительных шаблонов позволяет явно указать условия, при которых эти правила являются применимыми. Тем самым устраняется необходимость обеспечения тесного взаимодействия правил, а правила приобретают способность действовать оппортунистически (подчиняясь обстоятельствам). Применяемый способ формулировки правил показывает также, что вместо эвристик, использовавшихся первоначально, можно ввести в действие более явно выраженные эвристики, следующим образом:

```
IF a winning square is open, THEN take it.
IF a blocking square is open, and
   a winning square is not open, THEN take it.
IF a corner, middle, or side square is open, and
   a winning square is not open, and
   a blocking square is not open, THEN take it.
```

9.6 Конструкция `defmodule`

До сих пор все конструкции `defrule`, `deftemplate` и `deffacts` находились в одном рабочем пространстве, но в языке CLIPS предусмотрена **конструкция defmodule**, позволяющая секционировать базу знаний путем определения различных модулей. Основной синтаксис для этой конструкции выглядит так:

```
(defmodule <module-name> [<comment>])
```

По умолчанию в программе CLIPS определяется единственный модуль — `MAIN`. В предыдущих примерах имя модуля `MAIN` появлялось в результатах структурированного вывода содержимого конструкций, например, как показано ниже.

```
CLIPS> (clear)↵
```

```
CLIPS> (deftemplate sensor (slot name))↵
CLIPS> (ppdeftemplate sensor)↵
(deftemplate MAIN::sensor
   (slot name))
CLIPS>
```

Символ `::` в имени `MAIN::sensor` называется **отделителем имени модуля**. Справа от отделителя имени модуля находится имя конструкции, а слева — имя модуля, в котором содержится конструкция. Безусловно, до сих пор все определяемые конструкции по умолчанию помещались в модуль `MAIN`, поэтому в результатах структурированного вывода содержимого конструкций и появляется имя модуля `MAIN`.

Синтаксис конструкции `defmodule` позволяет определять новые модули. Используя приведенный выше пример системы обнаружения неисправностей, создадим модули, которые соответствуют фазам `DETECTION` (Обнаружение), `ISOLATION` (Локализация) и `RECOVERY` (Возобновление нормальной работы):

```
CLIPS> (defmodule DETECTION)↵
CLIPS>
(defmodule ISOLATION)↵
CLIPS> (defmodule RECOVERY)↵
```

После определения дополнительных модулей, кроме модуля `MAIN`, необходимо найти ответ на вопрос о том, в какие модули должны быть помещены те или иные новые конструкции. По умолчанию система CLIPS помещает вновь созданные конструкции в текущий модуль. После первоначального запуска или очистки среды системы CLIPS текущим модулем автоматически становится модуль `MAIN`. Таким образом, во всех предыдущих примерах был только один модуль, а использовался лишь текущий модуль, поэтому все определяемые конструкции помещались в модуль `MAIN`.

Сразу после определения нового модуля система CLIPS назначает в качестве текущего модуля именно его. А в приведенном выше диалоге последним был определен модуль `RECOVERY`, поэтому он становится текущим модулем и вновь определяемое правило будет помещено именно в него, как показано ниже.

```
CLIPS> (defrule example1 =>)↵
CLIPS>
(ppdefrule example1)↵
(defrule
RECOVERY::example1
   =>)
CLIPS>
```

Модуль, в который должна быть помещена конструкция, может быть также указан в имени конструкции. В таком случае в имени должен быть вначале указан модуль, затем — отделитель имени модуля, а после этого имя конструкции, например:

```
CLIPS> (defrule ISOLATION::example2 =>)↵
CLIPS>
(ppdefrule example2)↵
(defrule ISOLATION::example2
   =>)
CLIPS>
```

Если в имени конструкции задано имя какого-то модуля, то указанный модуль становится текущим модулем. Имя текущего модуля можно определить с помощью **функции get-current-module**, которая не принимает параметров и возвращает имя текущего модуля. С другой стороны, для изменения текущего модуля используется **функция set-current-module**. Она принимает единственный параметр (имя нового текущего модуля) и возвращает имя модуля, который был текущим перед этим, как показано ниже.

```
CLIPS> (get-current-module)↵
ISOLATION
CLIPS> (set-current-module DETECTION)↵
ISOLATION
CLIPS>
```

Способы задания имен модулей в командах

По умолчанию большинство команд CLIPS, действия которых распространяются на конструкции, работают только с конструкциями, содержащимися в текущем модуле. Например, команда `list-defrules` не сформирует какого-либо вывода, если текущим модулем является `DETECTION`, поскольку в этом модуле еще не содержатся какие-либо правила:

```
CLIPS> (list-defrules)↵
CLIPS>
```

А если бы потребовалось просмотреть конструкции `defrule`, содержащиеся в модуле `ISOLATION`, то можно было бы установить в качестве текущего модуль `ISOLATION`, а затем вызвать на выполнение другую команду `list-defrules`:

```
CLIPS> (set-current-module ISOLATION)↵
DETECTION
CLIPS> (list-defrules)↵
   example2
```

```
For a total of 1 defrule.
CLIPS>
```

Еще один вариант состоит в использовании того свойства команды `list-defrules`, что она принимает имя модуля в качестве необязательного параметра. Этот параметр указывает, для какого модуля должен быть подготовлен список правил:

```
CLIPS> (list-defrules RECOVERY)↵
   example1
For a total of 1 defrule.
CLIPS>
```

Если в качестве параметра команде `list-defrules` передается символ `*`, то создается список всех правил, содержащихся во всех модулях. Каждому списку предшествует имя модуля, а за ним находится список правил, содержащихся в этом модуле:

```
CLIPS> (list-defrules *)↵
MAIN:
DETECTION:
ISOLATION:
   example2
RECOVERY:
   example1
For a total of 2 defrules.
CLIPS>
```

Функции формирования списков `list-deftemplates` и `list-deffacts` действуют аналогично команде `list-defrules`. Команда `show-breaks` также позволяет указать, для какого модуля должны быть выведены все его точки останова. Модифицированный синтаксис для этих функций приведен ниже.

```
(list-defrules [<module-name>])
(list-deftemplates [<module-name>])
(list-deffacts [<module-name>])
(show-breaks [<module-name>])
```

Указывать имя модуля позволяют также команды, применяемые к конкретным конструкциям. Например, команда `ppdefrule` осуществляет поиск только в текущем модуле, если имя модуля не указано:

```
CLIPS> (ppdefrule example2)↵
(defrule ISOLATION::example2
   =>)
CLIPS> (ppdefrule example1)↵
```

```
[PRNTUTIL1] Unable to find defrule example1.
CLIPS>
```

В этом примере в формате структурированного вывода отображено только правило example2, поскольку оно действительно содержится в модуле ISOLATION. Но правило example1 не задано в модуле ISOLATION, поэтому не было найдено командой ppdefrule.

Модуль, в котором необходимо выполнить поиск некоторой конструкции, может быть указан путем задания имени модуля, за которым следует отделитель имени модуля, перед именем конструкции:

```
CLIPS> (ppdefrule RECOVERY::example1)↵
   (defrule RECOVERY::example1
   =>)
CLIPS>
```

В разных модулях могут находиться конструкции с одинаковыми именами. Ниже приведен пример, в котором перед именами конструкций задаются спецификаторы модулей, что позволяет применять в одной команде две разные конструкции с одинаковыми именами.

```
CLIPS> (defrule DETECTION::example1 =>)↵
CLIPS>
(list-defrules *)↵
MAIN:
DETECTION:
   example1
ISOLATION:
   example2
RECOVERY:
   example1
For a total of 3 defrules.
CLIPS> (ppdefrule RECOVERY::example1)↵
(defrule RECOVERY::example1
   =>)
CLIPS> (ppdefrule DETECTION::example1)↵
(defrule DETECTION::example1
   =>)
CLIPS>
```

Кроме того, задавать имя модуля в составе имени конструкции позволяют следующие команды: ppdefrule, undefrule, ppdeftemplate, undeftemplate, ppdeffacts, undeffacts, matches, refresh, remove-break и set-break.

9.7 Импорт и экспорт фактов

В предыдущем разделе было показано, как секционировать в программе конструкции, помещая их в отдельные модули. Предусмотрена также возможность секционировать сами факты. Факты, помещенные в список фактов, автоматически ассоциируются с тем модулем, в котором определены соответствующие им конструкции deftemplate:

```
CLIPS>
(deftemplate DETECTION::fault
   (slot component))↵
CLIPS> (assert (fault (component A)))↵
<Fact-0>
CLIPS> (facts)↵
f-0      (fault (component A))
For a total of 1 fact.
CLIPS>
(deftemplate ISOLATION::possible-failure
   (slot component))↵
CLIPS> (assert (possible-failure (component B)))↵
<Fact-1>
CLIPS> (facts)↵
f-1      (possible-failure (component B))
For a total of 1 fact.
CLIPS> (set-current-module DETECTION)↵
ISOLATION
CLIPS> (facts)↵
f-0      (fault (component A))
For a total of 1 fact.
CLIPS>
```

Обратите внимание на то, что к модулю ISOLATION относится единственный факт — possible-failure, так как в этом модуле содержится соответствующая ему конструкция deftemplate. То же самое относится к факту fault, содержащемуся в модуле DETECTION.

Команда facts, как и list-defrules, и аналогичные команды, может принимать имя модуля в качестве необязательного параметра. Команда facts имеет следующий синтаксис:

```
(facts [<module-name>]
       [<start> [<end> [<maximum>]]])
```

Как и при использовании команды list-defrules, при указании имени модуля в команде facts формируется список только тех фактов, которые содер-

жатся в указанном модуле. Кроме того, как показано ниже, если в качестве имени модуля применяется символ *, то в список включаются все факты.

```
CLIPS> (facts DETECTION)↵
f-0     (fault (component A))
For a total of 1 fact.
CLIPS> (facts ISOLATION)↵
f-1     (possible-failure (component B))
For a total of 1 fact.
CLIPS> (facts RECOVERY)↵
CLIPS> (facts *)↵
f-0     (fault (component A))
f-1     (possible-failure (component B))
For a total of 2 facts.
CLIPS>
```

В отличие от конструкций defrule и deffacts, конструкции deftemplate (а также все факты, в которых используются соответствующие конструкции deftemplate) могут поступать в совместное распоряжение с другими модулями. Факт "принадлежит" модулю, в котором содержится его конструкция deftemplate, но модуль-владелец может **экспортировать** конструкцию deftemplate, связанную с этим фактом, сделав тем самым данный факт (и все другие факты), в котором применяется эта конструкция deftemplate, видимым в других модулях. Но недостаточно просто экспортировать конструкцию deftemplate, чтобы факт стал видимым в другом модуле. Для использования конструкции deftemplate, определенной в другом модуле, необходимо также **импортировать** конструкцию deftemplate в этом модуле.

Если в модуле должен выполняться экспорт конструкций deftemplate, то в определении defmodule этого модуля необходимо использовать атрибут export. Атрибут export должен иметь один из следующих форматов:

```
(export ?ALL)
(export ?NONE)
(export deftemplate ?ALL)
(export deftemplate ?NONE)
(export deftemplate <deftemplate-name>+)
```

При использовании первого формата в модуле экспортируются все экспортируемые конструкции. Из числа конструкций, которые рассматривались до сих пор в данной книге, экспорту подлежат только конструкции deftemplate. Кроме того, могут также экспортироваться некоторые другие конструкции процедурного и объектно-ориентированного программирования языка CLIPS; синтаксис экспорта этих конструкций будет рассматриваться в главах 10 и 11. Второй формат показывает, что никакие конструкции не подлежат экспорту, и является приме-

няемым по умолчанию форматом для конструкции defmodule. Третий формат указывает, что все конструкции deftemplate в модуле подлежат экспорту. Применительно к конструкциям, которые рассматривались до сих пор в настоящей книге, этот формат совпадает с первым форматом. Аналогичным образом, четвертый формат позволяет указать, что никакие конструкции deftemplate не подлежат экспорту. Второй и четвертый форматы предусмотрены в основном для того, чтобы конструкции, экспортируемые некоторым модулем, могли быть явно заданы. Наконец, пятый формат позволяет задать конкретный список конструкций deftemplate, экспортируемых модулем. Атрибут export может быть задан больше одного раза в определении defmodule для указания экспортируемых конструкций различных типов, но до сих пор в качестве единственных экспортируемых конструкций мы рассматривали только конструкции deftemplate, поэтому не было необходимости использовать больше одного оператора с атрибутом export.

Как показано ниже, атрибут import также может иметь пять допустимых форматов.

```
(import <module-name> ?ALL)
(import <module-name> ?NONE)
(import <module-name> deftemplate ?ALL)
(import <module-name> deftemplate ?NONE)
(import <module-name> deftemplate
                      <deftemplate-name>+)
```

Каждый из этих форматов имеет тот же смысл, что и его аналог, предназначенный для экспорта, если не считать того, что с их помощью обеспечивается импорт указанных конструкций. Кроме того, должен быть указан модуль, из которого импортируются конструкции. Как и атрибут export, определение defmodule может иметь несколько атрибутов import.

Прежде чем любая конструкция может быть задана в списке импорта, она должна быть определена, но такое требование, чтобы конструкция была определена, прежде чем ее можно будет указать в списке экспорта, не является обязательным (чтобы можно было поместить конструкцию в модуль, необходимо определить сам модуль, поэтому в действительности невозможно определить конструкцию до определения модуля, в котором она экспортируется). В связи с наличием этого ограничения невозможна такая ситуация, в которой два модуля взаимно импортируют конструкции, заданные в другом модуле (например, если модуль A импортирует конструкции из модуля B, то невозможно добиться того, чтобы модуль B импортировал конструкции из модуля A).

В качестве иллюстрации импорта и экспорта фактов предположим, что в модуль RECOVERY необходимо импортировать конструкцию deftemplate с именем fault из модуля DETECTION, а из модуля ISOLATION — конструкцию

deftemplate с именем possible-failure. В отличие от других конструкций, конструкцию defmodule невозможно переопределить после того, как она определена в программе. Таким образом, для того чтобы заменить атрибуты import и export конструкции defmodule, необходимо вначале выполнить команду clear. Это ограничение распространяется на все модули, за исключением модуля MAIN, который определяется заранее; этот модуль может быть переопределен один раз для включения других атрибутов import и export (по умолчанию модуль MAIN ничего не экспортирует и не импортирует). Следует также отметить, что в применяемом по умолчанию определении модуля MAIN не экспортируется конструкция deftemplate с именем initial-fact. Как было указано в главе 8, при определенных обстоятельствах к левой части некоторых правил добавляется шаблон (initial-fact), в частности, это действие выполняется, если первым условным элементом правила является условный элемент not. Если такое правило помещается в модуль, в котором не импортируется конструкция deftemplate с именем initial-fact из модуля MAIN, то это правило не может быть активизировано. Следует также отметить, что при определении такого правила не возникает даже ошибка, поскольку введение шаблона (initial-fact) вызывает создание подразумеваемой конструкции deftemplate с именем initial-fact в текущем модуле.

Ниже приведены новые определения для модулей DETECTION, ISOLATION и RECOVERY наряду с их конструкциями deftemplate (все эти определения должны быть введены после выполнения команды clear).

```
(defmodule DETECTION
   (export deftemplate fault))
(deftemplate DETECTION::fault
   (slot component))
(defmodule ISOLATION
   (export deftemplate possible-failure))
(deftemplate ISOLATION::possible-failure
   (slot component))
(defmodule RECOVERY
   (import DETECTION deftemplate fault)
   (import ISOLATION deftemplate possible-
   failure))
```

Теперь, после того как определены конструкции defmodule и deftemplate, появляется возможность вводить факты fault в списки фактов модулей DETECTION и RECOVERY, а факты possible-failure — в списки фактов модулей RECOVERY и ISOLATION, как показано ниже.

```
CLIPS> (deffacts DETECTION::start
   (fault (component A)))↵
```

```
CLIPS> (deffacts ISOLATION::start
   (possible-failure (component B)))↵
CLIPS> (deffacts RECOVERY::start
   (fault (component C))
   (possible-failure (component D)))↵
CLIPS> (reset)↵
CLIPS> (facts DETECTION)↵
f-0      (fault (component A))
f-2      (fault (component C))
For a total of 2 facts.
CLIPS> (facts ISOLATION)↵
f-1      (possible-failure (component B))
f-3      (possible-failure (component D))
For a total of 2 facts.
CLIPS> (facts RECOVERY)↵
f-0      (fault (component A))
f-1      (possible-failure (component B))
f-2      (fault (component C))
f-3      (possible-failure (component D))
For a total of 4 facts.
CLIPS>
```

Обратите внимание на то, что после вставки фактов `fault` в список фактов любого из модулей `DETECTION` и `RECOVERY` эти факты становятся видимыми в обоих модулях. Такой же принцип соблюдается и по отношению к фактам `possible-failure`, вставляемым в список фактов модулей `RECOVERY` и `ISOLATION`.

9.8 Модули и управление выполнением программы

Конструкция `defmodule` может использоваться не только для управления тем, какие конструкции `deftemplate` могут импортироваться и экспортироваться модулем, но и для управления исполнением правил. В языке CLIPS каждый определяемый модуль не получает лишь часть одного общего рабочего списка правил, а имеет свой собственный рабочий список правил. Благодаря этому появляется возможность управлять выполнением программы и принимать решение о том, в каком модуле должен быть выбран рабочий список правил для исполнения правил. Например, все следующие конструкции `defrule` должны быть

активизированы с помощью фактов fault и possible-failure, введенных в список фактов в предыдущем примере:

```
(defrule DETECTION::rule-1
   (fault (component A | C))
   =>)
(defrule ISOLATION::rule-2
   (possible-failure (component B | D))
   =>)
(defrule RECOVERY::rule-3
   (fault (component A | C))
   (possible-failure (component B | D))
   =>)
```

Если бы после загрузки всех этих правил команда agenda была вызвана следующим образом, то отобразился бы рабочий список правил модуля RECOVERY, поскольку последнее заданное правило было помещено в этот модуль:

```
CLIPS> (get-current-module)↵
RECOVERY
CLIPS> (agenda)↵
0     rule-3: f-0,f-3
0     rule-3: f-0,f-1
0     rule-3: f-2,f-3
0     rule-3: f-2,f-1
For a total of 4 activations.
CLIPS>
```

Команда agenda, как и команды list-defrules и facts, принимает необязательный параметр, который в случае его определения отображает тот модуль, для которого должен быть выведен рабочий список правил:

```
CLIPS> (agenda DETECTION)↵
0        rule-1: f-2
0        rule-1: f-0
For a total of 2 activations.
CLIPS> (agenda ISOLATION)↵
   0     rule-2: f-3
   0     rule-2: f-1
For a total of 2 activations.
CLIPS> (agenda RECOVERY)↵
   0     rule-3: f-0,f-3
   0     rule-3: f-0,f-1
   0     rule-3: f-2,f-3
```

```
   0     rule-3: f-2,f-1
For a total of 4 activations.
CLIPS> (agenda *)↵
MAIN:
DETECTION:
   0     rule-1: f-2
   0     rule-1: f-0
ISOLATION:
   0     rule-2: f-3
   0     rule-2: f-1
RECOVERY:
   0     rule-3: f-0,f-3
   0     rule-3: f-0,f-1
   0     rule-3: f-2,f-3
   0     rule-3: f-2,f-1
For a total of 8 activations.
CLIPS>
```

Команда `focus`

Итак, теперь все правила распределены по трем отдельным рабочим спискам правил. Что же произойдет после выдачи команды `run` таким образом:

```
CLIPS> (unwatch all)↵
CLIPS> (watch rules)↵
CLIPS> (run)↵
CLIPS>
```

Ни одно из правил не запускается! Дело в том, что в системе CLIPS сопровождается не только информация о текущем модуле, который используется для определения того, в какие списки должны вводиться новые конструкции и какие конструкции применяются или подвергаются воздействию команд, но и **текущий фокус**, который определяет, какой рабочий список правил должен использоваться во время выполнения команды `run`. Команды `reset` и `clear` автоматически переводят текущий фокус на модуль `MAIN`. А после смены текущего модуля текущий фокус не изменяется. Таким образом, по условиям рассматриваемого примера после выдачи команды `run` для выбора правил, подлежащих исполнению, применяется рабочий список правил, связанный с модулем `MAIN`. Тем не менее этот рабочий список правил пуст, поэтому запуск правил не происходит.

Для смены текущего фокуса используется **команда focus**. Команда `focus` имеет следующий синтаксис:

```
(focus <module-name>+)
```

В том простом случае, когда в этой команде задается только одно имя модуля, текущий фокус устанавливается на указанный модуль. После переустановки текущего фокуса в модуль DETECTION и выдачи команды run происходит запуск правил из рабочего списка правил модуля DETECTION:

```
CLIPS> (focus DETECTION)↵
TRUE
CLIPS>
(run)↵
FIRE     1 rule-1: f-2
FIRE     2 rule-1: f-0
CLIPS>
```

При использовании команды focus не только происходит смена текущего фокуса, но и возвращается предыдущее значение текущего фокуса. Текущий фокус в действительности является верхним значением структуры в стековой структуре данных, называемой **стеком фокусов**. После каждой смены текущего фокуса с помощью команды focus фактически новый текущий фокус задвигается в верхнюю часть стека фокусов, смещая вниз все предыдущие текущие фокусы. А после того как в результате исполнения всех правил рабочий список правил модуля, находящегося в текущем фокусе, становится пустым, текущий фокус выталкивается (удаляется) из стека фокусов, и текущим фокусом становится следующий модуль. Затем исполняются правила из рабочего списка правил нового текущего фокуса до тех пор, пока фокус не перейдет на новый модуль или в рабочем списке правил текущего модуля больше не останется правил. Правила продолжают исполняться до тех пор, пока в стеке фокусов не остается больше модулей или не вызывается команда halt.

Продолжая рассматриваемый пример, укажем, что если вначале фокус будет переведен на модуль ISOLATION, а затем на модуль RECOVERY, то это вызовет запуск всех правил из рабочего списка правил модуля RECOVERY, а за этим последует запуск правил из рабочего списка правил модуля ISOLATION. Для отображения перечня модулей, находящихся в стеке фокусов, применяется **команда list-focus-stack** (которая не принимает параметров):

```
CLIPS> (focus ISOLATION)↵
TRUE
CLIPS>
(focus RECOVERY)↵
TRUE
CLIPS>
(list-focus-stack)↵
RECOVERY
ISOLATION
```

```
CLIPS> (run)↵
FIRE     1 rule-3: f-1,f-4
FIRE     2 rule-3: f-1,f-2
FIRE     3 rule-3: f-3,f-4
FIRE     4 rule-3: f-3,f-2
FIRE     5 rule-2: f-4
FIRE     6 rule-2: f-2
CLIPS> (list-focus-stack)↵
CLIPS>
```

Использование двух команд `focus`, позволяющих задвинуть в стек фокусов модули `ISOLATION` и `RECOVERY`, вызовет то, что правила `RECOVERY` будут исполняться прежде, чем правила `ISOLATION`. Но предусмотрена также возможность задать в одной команде `focus` сразу несколько модулей; в таком случае модули задвигаются в стек фокусов справа налево, например, как показано ниже.

```
CLIPS> (focus ISOLATION RECOVERY)↵
TRUE
CLIPS> (list-focus-stack)↵
ISOLATION
RECOVERY
CLIPS> (focus ISOLATION)↵
TRUE
CLIPS> (list-focus-stack)↵
ISOLATION
RECOVERY
CLIPS> (focus RECOVERY)↵
TRUE
CLIPS>
(list-focus-stack)↵
RECOVERY
ISOLATION
RECOVERY
CLIPS>
```

Следует отметить, что один и тот же модуль может быть представлен в стеке фокусов больше одного раза, но перевод фокуса на модуль, который уже находится в текущем фокусе, не приводит к получению каких-либо результатов.

Манипулирование стеком фокусов и его изучение

В языке CLIPS предусмотрено несколько команд и функций для манипулирования текущим фокусом и стеком фокусов. **Команда clear-focus-stack** позволяет

удалить все модули из стека фокусов. **Функция get-focus** возвращает имя модуля, находящегося в текущем фокусе, или символ FALSE, если стек фокусов пуст. **Функция pop-focus** удаляет текущий фокус из стека фокусов (и возвращает имя модуля, который был в текущем фокусе, или символ FALSE, если стек фокусов пуст). **Функция get-focus-stack** возвращает многозначное значение, содержащее имена модулей, находящихся в стеке фокусов. Примеры применения этих команд и функций приведены ниже.

```
CLIPS> (get-focus-stack)↵
(RECOVERY ISOLATION RECOVERY)
CLIPS> (get-focus)↵
RECOVERY
CLIPS> (pop-focus)↵
RECOVERY
CLIPS> (clear-focus-stack)↵
CLIPS>
(get-focus-stack)↵
()
CLIPS>
(get-focus)↵
FALSE
CLIPS>
(pop-focus)↵
FALSE
CLIPS>
```

Для контроля над изменениями в стеке фокусов может использоваться команда watch, в вызове которой в качестве параметра применяется ключевое слово focus:

```
CLIPS> (watch focus)↵
CLIPS> (focus DETECTION ISOLATION RECOVERY)↵
==> Focus RECOVERY
==> Focus ISOLATION from RECOVERY
==> Focus DETECTION from ISOLATION
TRUE
CLIPS>
(run)↵
<== Focus DETECTION to ISOLATION
<== Focus ISOLATION to RECOVERY
<== Focus RECOVERY
CLIPS>
```

В том случае, если выдана команда `run`, а стек фокусов пуст, в стек фокусов автоматически задвигается модуль `MAIN`. Такая возможность предусмотрена в основном для удобства, на тот случай, что ко времени завершения программы добавляются новые активизации, а в стеке фокусов не остается больше модулей, например, как показано ниже.

```
CLIPS> (clear)↵
CLIPS> (watch focus)↵
CLIPS> (watch rules)↵
CLIPS> (defrule example-1 =>)↵
CLIPS> (reset)↵
<== Focus MAIN
==> Focus MAIN
CLIPS> (run)↵
FIRE     1 example-1: f-0
<== Focus MAIN
CLIPS> (defrule example-2 =>)↵
CLIPS> (agenda)↵
0     example-2: f-0
For a total of 1 activation.
CLIPS>
(list-focus-stack)↵
CLIPS>
```

В рабочем списке правил имеется правило `example-2`, но в стеке фокусов отсутствуют какие-либо модули. А после выдачи команды `run` в стек фокусов помещается модуль `MAIN`, поэтому так или иначе появляется возможность запуска правила `example-2`:

```
CLIPS> (run)↵
==> Focus MAIN
FIRE     1 example-2: f-0
<== Focus MAIN
CLIPS>
```

Команда `return`

Один из недостатков способа управления потоком выполнения с использованием управляющих фактов для представления фаз, который описан в разделе 9.4, состоит в том, что он не позволяет выполнить запуск некоторых активизированных правил в конкретной фазе, выйти из этой фазы, а затем еще раз вернуться в ту же фазу и исполнить остальные активизированные правила из рабочего списка правил. Это происходит потому, что после извлечения управляющего факта,

который представляет данную фазу, из рабочего списка правил удаляются все активизированные правила, относящиеся к соответствующей фазе. А после того как в дальнейшем в список фактов вновь будет вставлен управляющий факт, повторно активизируются все ранее активизированные правила, относящиеся к данной фазе, а не только те правила, которые не были запущены (разумеется, это происходит лишь при том условии, что вставляется или удаляется только управляющий факт, но не какие-либо другие факты).

Если для управления потоком выполнения используются модули, то появляется возможность преждевременно прекратить исполнение активизированных правил из рабочего списка правил определенного модуля (в данном случае преждевременно означает — до того, как рабочий список правил модуля станет пустым). Для немедленного прекращения выполнения правой части правила и удаления текущего фокуса из стека фокусов (и передачи управления над выполнением правил в следующий модуль в стеке фокусов) может применяться **команда return**. В команду `return` при ее использовании в правой части правила не должны передаваться параметры (команда `return` может также применяться в конструкциях процедурного программирования, предусмотренных в языке CLIPS). Использование команды `return` иллюстрируется в следующем примере:

```
CLIPS> (clear)↵
CLIPS>
(defmodule MAIN
   (export deftemplate initial-fact))↵
CLIPS>
(defmodule DETECTION
   (import MAIN deftemplate initial-fact))↵
CLIPS>
(defrule MAIN::start
   =>
   (focus DETECTION))↵
CLIPS>
(defrule DETECTION::example-1
   =>
   (return)
   (printout t "No printout!" crlf))↵
CLIPS>
(defrule DETECTION::example-2
   =>
   (return)
   (printout t "No printout!" crlf))↵
CLIPS> (watch rules)↵
```

```
CLIPS> (watch focus)↵
CLIPS> (reset)↵
<== Focus MAIN
==> Focus MAIN
CLIPS> (run)↵
FIRE     1 start: f-0
==> Focus DETECTION from MAIN
FIRE     2 example-1: f-0
<== Focus DETECTION to MAIN
<== Focus MAIN
CLIPS>
```

В этом примере заслуживают внимания две важные особенности. Во-первых, активизация правил в модуле DETECTION с использованием заданного по умолчанию шаблона initial-fact становится возможной, только если конструкция deftemplate с именем initial-fact будет экспортирована модулем MAIN и импортирована модулем DETECTION. Во-вторых, важно то, что команда return немедленно останавливает выполнение правой части правила. Об этом свидетельствует то, что в правиле example-1 не выполняется команда printout, которая следует за командой return (то же касается правила example-2, поскольку это правило не получает возможности запуска). Следует отметить, что по своим функциональным возможностям команда return аналогична, но не идентична команде pop-focus, поскольку последняя удаляет текущий фокус из стека фокусов, но допускает продолжение выполнения действий в правой части правила. Если бы в рассматриваемом примере вместо команды return использовалась команда pop-focus, то во время выполнения действий правила example-1 на внешнее устройство была бы выведена строка "No printout!".

Средство auto-focus

В языке CLIPS не только предоставляется возможность явно переводить фокус на те или иные модули с помощью команды focus, но и обеспечивается автоматический перевод фокуса на конкретный модуль при активизации конкретных правил из этого модуля. По умолчанию на модуль правила при активизации этого правила фокус автоматически не переводится. Такое положение дел можно изменить с использованием **атрибута auto-focus**. Атрибут auto-focus должен быть задан в операторе declare наряду с атрибутом salience. В этом случае задается ключевое слово auto-focus, за которым следует либо символ TRUE (который позволяет разрешить средство auto-focus), либо символ FALSE (запрещающий это средство). Для обеспечения использования средства auto-focus в некоторых правилах модуля не обязательно, чтобы это средство

было разрешено для всех правил модуля. Аналогичным образом, если для правила применяется оператор `declare`, не обязательно объявлять и атрибут `salience`, и атрибут `auto-focus`. Использование средства `auto-focus` иллюстрируется в следующем примере:

```
CLIPS> (clear)↵
CLIPS>
(defmodule MAIN
   (export deftemplate initial-fact))↵
CLIPS>
(defmodule DETECTION
   (import MAIN deftemplate initial-fact))↵
CLIPS>
(defrule DETECTION::example
   (declare (auto-focus TRUE))
   =>)↵
CLIPS> (watch focus)↵
CLIPS> (reset)↵
<== Focus MAIN
==> Focus MAIN
==> Focus
DETECTION from MAIN
CLIPS>
```

После выдачи команды `reset` фокус автоматически переводится на модуль `MAIN`, поскольку стек фокусов пуст. Затем в результате ввода факта `initial-fact` в список фактов активизируется правило `example`. Поскольку для этого правила разрешен атрибут `auto-focus`, то в стек фокусов автоматически задвигается модуль этого правила — модуль `DETECTION`. Средство `auto-focus` особенно удобно при использовании правил, которые обнаруживают нарушения ограничений. Оно позволяет немедленно перевести текущий фокус на модуль ограничивающего правила и поэтому дает возможность предпринять необходимые действия сразу после обнаружения соответствующего нарушения. При этом отсутствует необходимость явно предусматривать отдельную фазу, на которой обнаруживались бы нарушения.

Отказ от фаз и управляющих фактов

Благодаря использованию конструкций `defmodule`, команд `focus` и `return`, а также средства `auto-focus` появилась возможность отказаться от применения фаз и управляющих фактов и создать гораздо более явно выраженный механизм управления потоком выполнения правил. В таком случае вместо описанных в раз-

деле 9.4 конструкций, применяемых для управления выполнением, могут использоваться следующие конструкции:

```
(defmodule DETECTION)
(defmodule ISOLATION)
(defmodule RECOVERY)
(deffacts MAIN::control-information
   (phase-sequence DETECTION ISOLATION RECOVERY))
(defrule MAIN::change-phase
   ?list <- (phase-sequence ?next-phase
   $?other-phases)
   =>
   (focus ?next-phase)
   (retract ?list)
   (assert (phase-sequence ?other-phases
   ?next-phase)))
```

Управление выполнением передается по циклу от модуля DETECTION к модулю ISOLATION, затем к модулю RECOVERY и возвращается в начало. Происходит запуск всех правил каждого модуля, прежде чем появляется возможность запуска правил в следующем модуле (при условии, что не выдается команда return или фокус не переходит на новый модуль в результате применения средства auto-focus), как показано ниже.

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch rules)↵
CLIPS>
(watch focus)↵
CLIPS> (run 5)↵
FIRE 1 change-phase: f-1
==> Focus DETECTION from MAIN
<== Focus DETECTION to MAIN
FIRE     2 change-phase: f-2
==> Focus ISOLATION from MAIN
<== Focus ISOLATION to MAIN
FIRE     3 change-phase: f-3
==> Focus RECOVERY from MAIN
<== Focus RECOVERY to MAIN
FIRE     4 change-phase: f-4
==> Focus DETECTION from
MAIN
<== Focus DETECTION to MAIN
```

```
FIRE     5 change-phase: f-5
==> Focus ISOLATION from MAIN
<== Focus ISOLATION to MAIN
CLIPS>
```

9.9 Rete-алгоритм сопоставления с шаблонами

В таких языках, основанных на правилах, как CLIPS, Jess, Eclipse и OPS5, используется очень эффективный алгоритм сопоставления фактов с шаблонами правил и определения того, условия каких правил удовлетворены. Этот алгоритм называется **rete-алгоритмом сопоставления с шаблонами** [12], [27], [28]. Для написания эффективных правил на языке CLIPS не требуется понимание работы rete-алгоритма. Тем не менее понимание основополагающих алгоритмов, образующих фундамент языка CLIPS и других языков, основанных на правилах, позволяет разобраться в том, почему правила, написанные в одной форме, являются более эффективными, чем написанные в другой форме.

Чтобы разобраться в том, почему rete-алгоритм является таким эффективным, целесообразно рассмотреть проблему согласования фактов с правилами в целом, а затем ознакомиться с другими алгоритмами, которые не являются столь же эффективными. На рис. 9.6 показано, какую задачу решает rete-алгоритм.

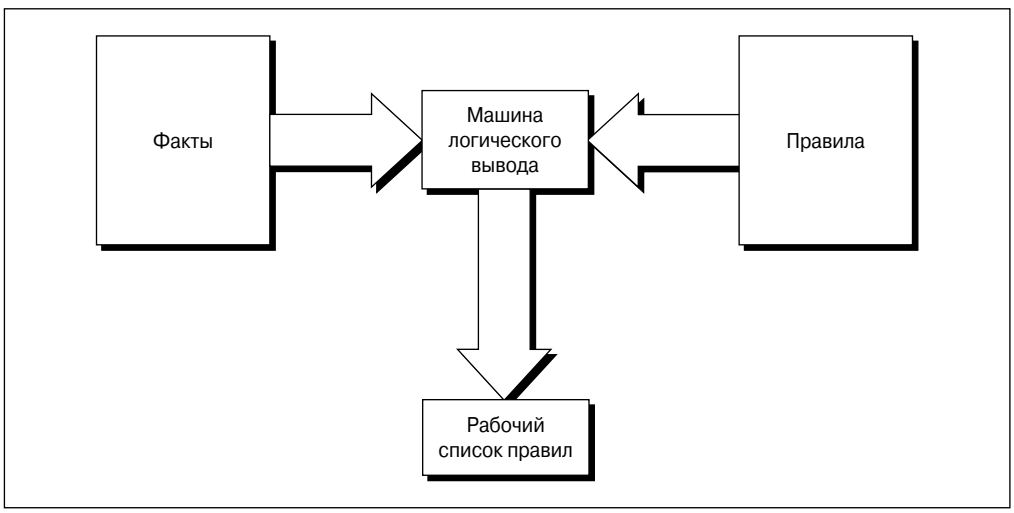


Рис. 9.6. Сопоставление с шаблонами — правила и факты

Если бы процесс согласования должен был произойти только один раз, то решение задачи согласования было бы несложным. Машина логического вывода может исследовать каждое правило, а затем выполнить поиск во множестве

фактов, чтобы определить, удовлетворяются ли шаблоны данного правила. Если проверка условий, заданных шаблонами правила, завершается успешно, то правило можно поместить в рабочий список правил. Этот подход показан на рис. 9.7.

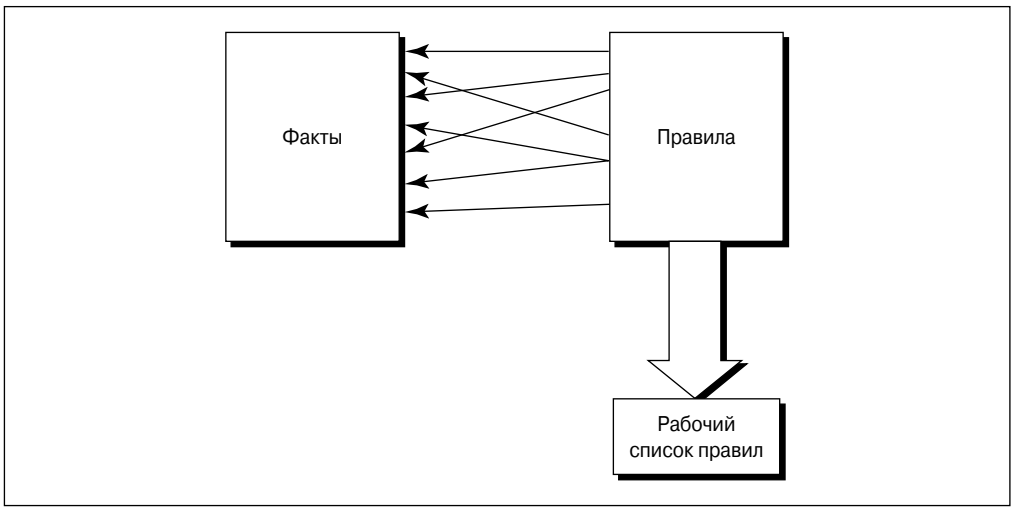


Рис. 9.7. Принцип поиска фактов, удовлетворяющих правилам

Тем не менее в языках, основанных на правилах, процесс согласования должен осуществляться неоднократно. Обычно состав списка фактов изменяется во время каждого цикла выполнения. При этом в список фактов могут быть добавлены новые факты или из этого списка могут быть удалены старые факты. Может оказаться, что в результате таких изменений станут удовлетворяться те шаблоны, проверка условий которых перед этим оканчивалась неудачей, или наоборот. Теперь процесс решения задачи согласования становится осуществляемым неоднократно. При этом во время каждого цикла приходится сопровождать и обновлять множество правил, проверка условий которых завершена успешно, в связи с добавлением и удалением фактов.

Простой и незамысловатый метод решения указанной проблемы мог бы состоять в обеспечении того, чтобы машина логического вывода проверяла каждое правило, позволяя снова осуществить поиск фактов, после каждого цикла выполнения. Но основным недостатком такого подхода является то, что он может показывать очень низкое быстродействие. В большинстве экспертных систем, основанных на правилах, обнаруживается свойство, называемое **временно́й избыточностью**. Это свойство заключается в том, что обычно в результате выполнения действий любого правила изменяется лишь немного фактов в списке фактов. Иными словами, состав фактов в экспертной системе изменяется со временем довольно медленно. Может оказаться, что в каждом цикле выполнения добавляется

или удаляется лишь небольшая процентная доля фактов, и поэтому изменения в списке фактов обычно затрагивают лишь небольшую процентную долю правил. Иначе говоря, подход, требующий повторного просмотра всех правил для поиска необходимых фактов, связан с использованием большого объема ненужных вычислений, поскольку с наибольшей вероятностью для большинства правил в текущем цикле будут найдены те же факты, что и в предыдущем цикле. Причины неэффективности этого подхода иллюстрируются на рис. 9.8. Затененная область представляет те изменения, которые произошли в списке фактов. Ненужных повторных вычислений можно избежать, передавая результаты уже выполненных согласований из одного цикла в другой, а затем проводя только применительно к изменениям вычисления, необходимые в связи с наличием вновь добавленных или вновь удаленных фактов (рис. 9.9). Правила остаются неизменными, изменяется только состав фактов, поэтому не правила должны применяться для поиска фактов, а факты — для поиска правил.

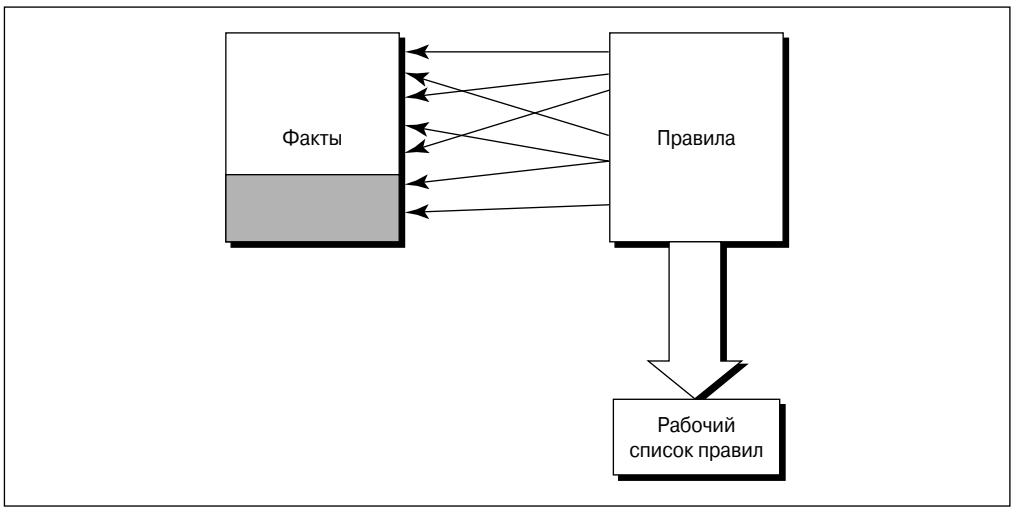


Рис. 9.8. Ненужные вычисления, осуществляемые при использовании правил для поиска фактов

Rete-алгоритм сопоставления с шаблонами позволяет воспользоваться свойством временно́й избыточности, которым обладают экспертные системы, основанные на правилах. В rete-алгоритме такая цель достигается за счет сохранения состояния процесса согласования от цикла к циклу, а затем проведения повторных вычислений в связи с изменениями в этом состоянии, с учетом только тех изменений, которые произошли в списке фактов. Иначе говоря, если в одном цикле обнаружены два из трех необходимых фактов для множества шаблонов, то на следующем цикле нет необходимости выполнять проверку для тех двух фактов, которые уже были найдены, поскольку теперь интерес представляет только тре-

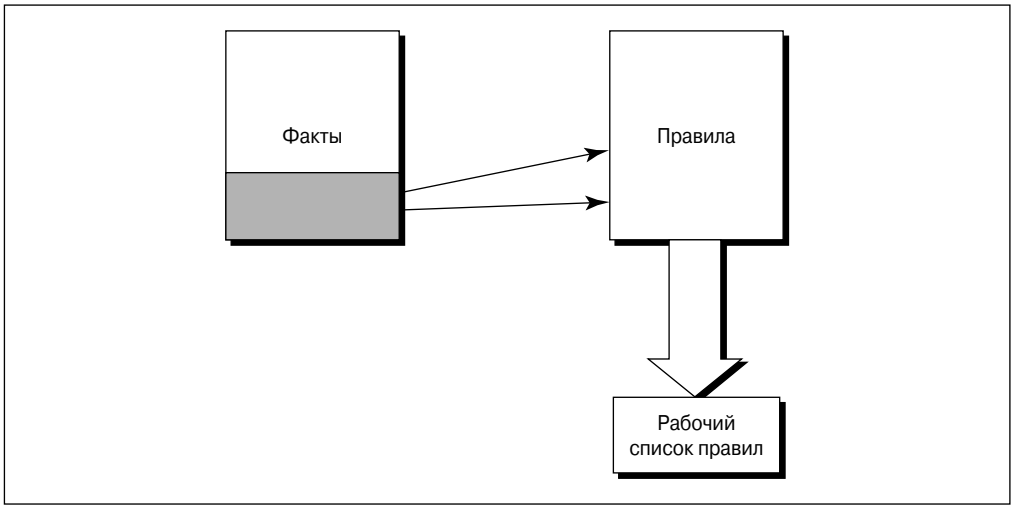


Рис. 9.9. Применение подхода, в котором факты служат для поиска правил

тий факт. Состояние процесса согласования модифицируется лишь в результате добавления и удаления фактов. Это означает, что если количество добавляемых и удаляемых фактов невелико по сравнению с общим количеством фактов и шаблонов, то процесс согласования протекает очень быстро. В наихудшем случае, если происходят изменения во всех фактах, то процесс согласования действует так, как если бы все факты должны быть сопоставлены со всеми шаблонами.

При использовании подхода, в котором обрабатываются только обновления в списке фактов, необходимо предусмотреть хранение в памяти информации о том, какие сопоставления фактов с шаблонами уже были выполнены успешно в каждом правиле. Иными словами, если окажется, что новый факт согласуется с третьим шаблоном правила, то информация о сопоставлениях, относящаяся к первым двум шаблонам, должна быть доступной, чтобы можно было завершить процесс согласования. Информация о состоянии такого типа, которая указывает на факты, сопоставленные ранее с шаблонами правила, называется **частичным соответствием**. Частичным соответствием для правила является любое множество фактов, которые были успешно сопоставлены с шаблонами правила, начиная с первого шаблона правила и заканчивая любым другим шаблоном, включая последний. Таким образом, правило с тремя шаблонами может иметь частичное соответствие с первым шаблоном, с первым и вторым шаблонами, а также с первым, вторым и третьим шаблонами. Частичное соответствие со всеми шаблонами правила становится также активизацией правила. Еще одним типом хранимой информации о состоянии является так называемое **сопоставление с шаблонами**. Сопоставление с шаблоном возникает, если факт удовлетворяет единственному

шаблону любого правила и рассматривается без учета того, что переменные других шаблонов могут ограничивать процесс согласования.

Основной недостаток rete-алгоритма сопоставления с шаблонами состоит в том, что этот алгоритм требует большого объема памяти. Если осуществляется сравнение всех фактов со всеми шаблонами, то память не требуется. Если же приходится хранить информацию о состоянии системы, представленную в виде результатов сопоставлений с шаблонами и частичных соответствий, то потребность в памяти становится существенной. Вообще говоря, чем больший объем памяти занимает заранее подготовленная информация, тем выше быстродействие, но объем потребляемой памяти невозможно увеличивать до бесконечности, поэтому приходится искать компромиссный вариант. Тем не менее важно помнить, что плохо написанные правила не только работают медленно, но и расходуют значительный объем памяти.

Rete-алгоритм позволяет также повысить эффективность систем, основанных на правилах, благодаря тому, что в нем используется свойство **структурного подобия** правил. Структурное подобие часто характеризуется тем, что во многих правилах содержатся одинаковые шаблоны или группы шаблонов. Это свойство используется в rete-алгоритме для повышения эффективности по такому принципу, что одинаковые компоненты объединяются в пулы для того, чтобы вычисления значений этих компонентов не приходилось проводить больше одного раза.

9.10 Сеть шаблонов

Процесс согласования фактов с шаблонами правил можно разделить на два этапа. Во-первых, после добавления и удаления фактов необходимо определить, с какими шаблонами должно быть выполнено сопоставление. Во-вторых, необходимо провести сравнение связываний переменных в различных шаблонах, чтобы определить частичные соответствия для группы шаблонов.

Процесс определения того, какие факты согласуются с шаблонами, осуществляется в **сети шаблонов**. В целях сокращения изложения ограничимся описанием процесса сопоставления с шаблонами в однозначных слотах фактов `deftemplate`. Все операции согласования, в которых не участвуют операции сравнения с переменными, связанными в других шаблонах, могут быть выполнены в сети шаблонов. Сеть шаблонов имеет структуру, подобную дереву, в котором ограничениями первого слота всех шаблонов являются узлы, соединенные с корнем дерева, ограничениями второго слота всех шаблонов — узлы, соединенные с этими узлами, и т.д. Ограничениями последнего слота в шаблоне являются листовые узлы дерева. Узлы в сети шаблонов называются **одновходовыми узлами**, поскольку каждый из них получает информацию только из узла, находящегося над ним. Узлы в сети шаблонов иногда называют также **узлами шаблонов**, а листовые

узлы — **терминальными узлами**. Каждый узел шаблона содержит спецификацию, используемую для определения того, согласовано ли значение слота факта с ограничением слота некоторого шаблона. Например, следующий шаблон был бы представлен только одним узлом, поскольку в нем имеется лишь одно ограничение слота:

```
(data (x 27))
```

В таком случае спецификация согласования для первого узла была бы выражена следующим образом:

```
The slot x value is equal to the constant 27
```

В действительности необходимо также проверить, правильно ли определено то, что факт имеет имя отношения, соответствующее данному шаблону (например, нельзя допускать, чтобы факты `foobar` согласовывались с шаблонами `data` лишь потому, что в них имеются те же имена слотов). В системе CLIPS для проведения такой проверки сопровождается отдельная сеть шаблонов для каждой конструкции `deftemplate`, чтобы можно было выполнять проверку имени отношения ко времени создания факта, но до проведения сопоставления с шаблонами.

Спецификации согласования включают всю информацию, необходимую для согласования отдельного слота. Несколько проверок могут проводиться одновременно. Например, такой шаблон:

```
(data (x ~red&~green))
```

может применяться для выработки следующей спецификации согласования для слота `x`:

```
The slot x value is not equal to the constant red
   and is not equal to the constant green
```

Обычно в сети шаблонов связывание переменных проверяется, только если переменная используется в шаблоне больше одного раза. Например, применение показанного ниже шаблона не приводит к формированию спецификаций согласования ни для слота `x`, ни для слота `y`, поскольку первое связывание переменной со значением слота не влияет на то, будет ли успешно выполнено согласование шаблона с фактом.

```
(data (x ?x) (y ?y) (z ?x))
```

Но слот `z` должен иметь такое же значение, как и слот `x`, поэтому спецификация согласования для слота `z` должна быть следующей:

```
The z slot value is equal to the x slot value
```

В сети шаблонов могут быть проверены такие выражения, которые содержат переменные, полностью находящиеся внутри шаблона. Например, использование приведенного ниже шаблона не приведет к формированию спецификации

согласования для слота x, поскольку переменная ?y не содержится в шаблоне (разумеется, применение такого шаблона возможно, только если переменная ?y определена в одном из предыдущих шаблонов).

```
(data (x ?x&:(> ?x ?y)))
```

Тем не менее слот x в следующем шаблоне:

```
(data (x ?x&:(> ?x 4)))
```

имел бы показанную ниже спецификацию согласования, поскольку единственная переменная, находящаяся в выражении ?x, содержится также в данном шаблоне.

```
The x slot value is greater than the constant 4
```

Как уже было сказано выше, сеть шаблонов имеет иерархическую организацию, в которой узлы шаблона, соответствующие ограничениям первого слота шаблонов, находятся в верхней части. После внесения любого факта в список фактов проверяются узлы шаблонов для поиска ограничений первого слота в сети шаблонов. Любой узел шаблона, для которого проверка спецификации согласования завершается успешно, активизирует находящиеся непосредственно под ним узлы шаблонов. Этот процесс продолжается до тех пор, пока не достигаются терминальные узлы в сети шаблонов. Достижение терминальных узлов в сети шаблонов соответствует окончанию проверки шаблона и успешному согласованию шаблона. С каждым терминальным узлом связана **альфа-память**, или **правая часть памяти**. Альфа-память содержит множество всех фактов, согласованных с шаблоном, который ассоциирован с данным терминальным узлом. Иными словами, альфа-память хранит множество сопоставлений с шаблоном для конкретного шаблона.

В сети шаблонов используется преимущество структурного подобия, поскольку общие узлы шаблонов применяются совместно в разных шаблонах. Дело в том, что хранение узлов шаблонов организовано иерархически, поэтому два шаблона могут совместно использовать свои первые N узлов шаблонов, если спецификации согласования для их первых N ограничений слота являются идентичными. Например, следующие шаблоны могут совместно использовать общие узлы шаблонов для слота x:

```
(data (x red) (y green))
(data (x red) (y blue))
```

Обратите внимание на то, что идентичными должны быть спецификации согласования, но не обязательно ограничения слота, находящиеся в шаблоне. Например, в приведенных ниже шаблонах может совместно использоваться узел шаблона для слота y, даже несмотря на то, что переменные, применяемые в этих двух шаблонах, являются разными.

```
(data (x ?x) (y ?x))
(data (x ?y) (y ?y))
```

Для слотов x не формируется спецификация согласования и поэтому они игнорируются с точки зрения обеспечения совместного использования. Тем не менее изменение порядка следования слотов на противоположный привело бы к тому, что в шаблонах нельзя было совместно использовать узлы, поскольку первый шаблон формировал бы спецификацию согласования для слота y, а второй шаблон — спецификацию согласования для слота x, как показано ниже.

```
(data (x ?x) (y ?x))
(data (y ?y) (x ?y))
```

На рис. 9.10 показана сеть шаблонов, сформированная с приведенными ниже правилами.

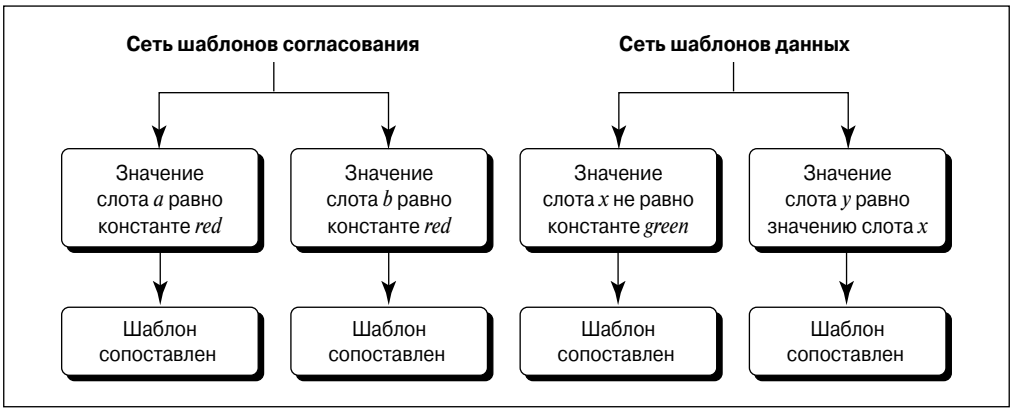


Рис. 9.10. Сеть шаблонов для двух правил

```
(defrule Rete-rule-1
   (match (a red))
   (data (x ?x) (y ?x))
   =>)
(defrule Rete-rule-2
   (match (a ?x) (b red))
   (data (x ~green) (y ?x))
   (data (x ?x) (y ?x))
   =>)
```

9.11 Сеть соединений

После определения того, какие шаблоны согласованы с фактами, необходимо проверить результаты сравнения связываний переменных в различных шаблонах, чтобы убедиться в том, что переменные, используемые более чем в одном

шаблоне, имеют совместимые значения. Такое сравнение выполняется в **сети соединений**. Каждый терминальный узел в сети шаблонов действует в качестве входа в **узел соединения**, или в **двухвходовой узел** сети соединений. Каждое соединение содержит спецификацию согласования для согласований в альфа-памяти, ассоциированной с этим терминальным узлом, и для множества частичных соответствий, которое было согласовано с предыдущими шаблонами. Частичные соответствия для предыдущих шаблонов хранятся в **бета-памяти**, или в **левой части памяти** соединения. Таким образом, для правила с `N` шаблонами требуется `N-1` соединение (но в системе CLIPS для упрощения rete-алгоритма фактически используется всего `N` соединений, что позволяет вводить в действие каждое соединение только с помощью одного шаблона).

В первом соединении сравниваются первые два шаблона, а в оставшихся соединениях сравнивается шаблон, дополнительный по отношению к частичным соответствиям предыдущего соединения. Например, если дано такое правило `Rete-rule-2`, используемое в приведенном выше примере:

```
(defrule Rete-rule-2
   (match (a ?x) (b red))
   (data (x ~green) (y ?x))
   (data (x ?x) (y ?x))
   =>)
```

то первое соединение должно содержать следующую спецификацию согласования:

```
The a slot value of the fact
   bound to the first pattern is equal to
the y slot value of the fact
   bound to the second pattern.
```

В таком случае во втором соединении было бы получено в качестве входных данных множество частичных соответствий из первого соединения, поэтому второе соединение содержало бы следующую спецификацию согласования:

```
The x slot value of the fact
   bound to the third pattern is equal to
the y slot value of the fact
   bound to the second pattern.
```

Обратите внимание на то, что можно было бы сравнить значение переменной `?x` в третьем шаблоне со значением переменной `?x` в первом шаблоне, а не с переменной `?x` во втором шаблоне. Второе вхождение переменной `?x` в третьем шаблоне в этой сети соединений проверять не требуется, поскольку ее значение можно проверить в сети шаблонов. Сети шаблонов и соединений для правила `Rete-rule-2` показаны на рис. 9.11.

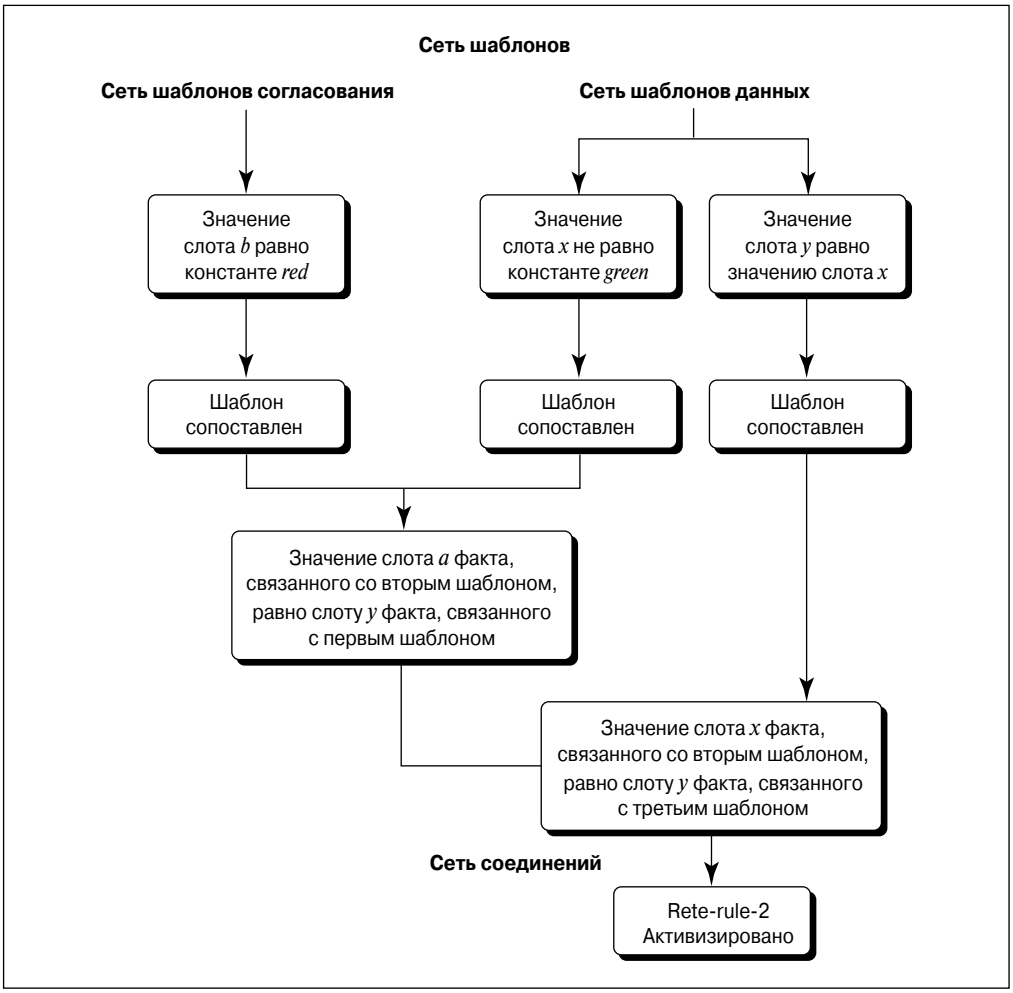


Рис. 9.11. Сети шаблонов и соединений для правила `Rete-rule-2`

В сети соединений можно воспользоваться преимуществом структурного подобия для совместного использования соединений в различных правилах. Соединения в сети соединений могут совместно использоваться в двух правилах, если они имеют идентичные шаблоны и результаты сравнения соединений для двух или нескольких шаблонов, начиная с первого шаблона. Например, в приведенных ниже правилах можно было совместно использовать все их шаблоны в сети шаблонов, а также соединения, созданные для их первых трех шаблонов в сети соединений.

```
(defrule sharing-1
   (match (a ?x) (b red))
```

```
   (data (x ~green) (y ?x))
   (data (x ?x) (y ?x))
   (other (q ?z))
   =>)
(defrule sharing-2
   (match (a ?y) (b red))
   (data (x ~green) (y ?y))
   (data (x ?y) (y ?y))
   (other (q ?y))
   =>)
```

Но соединение для четвертого шаблона нельзя совместно использовать в сети соединений, поскольку спецификации согласования для двух рассматриваемых правил являются разными. В спецификации согласования для четвертого шаблона правила `sharing-1` не требуется выполнять какие-либо сравнения, поскольку переменная `?z` не применяется в других шаблонах. Но в спецификации согласования для четвертого шаблона правила `sharing-2` необходимо предусмотреть сравнение переменной `?y` с переменной `?y` другого шаблона для обеспечения того, чтобы связывания этих переменных являлись совместимыми. Еще раз отметим, что для использования преимущества структурного подобия не требуется, чтобы имена переменных были идентичными. Имеет значение лишь то, являются ли идентичными спецификации согласования.

После выдачи команды `watch compilations` система CLIPS предоставляет полезную информацию о совместном использовании соединений. Например, ниже приведены команды, которые показывают, как отображается информация о совместном использовании соединений. В данном случае предполагается, что правила `sharing-1` и `sharing-2` хранятся в файле "`rules.clp`".

```
CLIPS> (watch compilations)↵
CLIPS>
(load "rules.clp")↵
Defining defrule: sharing-1 +j+j+j+j
Defining defrule: sharing-2 =j=j=j+j
TRUE
CLIPS>
```

Знаки `+j` в этом выводе указывают, что добавляется некоторое соединение, а знаки `=j` говорят о том, что соединение используется совместно. Таким образом, после введения правила `sharing-1` создаются четыре новых соединения. С другой стороны, после введения правила `sharing-2` в нем совместно с правилом `sharing-1` используются первые три соединения, а для последнего шаблона создается новое соединение. Обратите внимание на то, что в системе CLIPS для представления данного правила использовались четыре соединения, а не три, что

потребовалось бы при обычных обстоятельствах. В целях упрощения реализации удобнее иметь только по одному шаблону в расчете на каждое соединение, поэтому вместо применения одного соединения для первых двух шаблонов в системе CLIPS используется два соединения.

9.12 Важность правильного выбора порядка расположения шаблонов

Программисты, впервые осваивающие языки, основанные на правилах, часто не представляют себе, насколько важно добиться должного упорядочения шаблонов в правилах с точки зрения быстродействия и эффективного использования памяти. Поскольку rete-алгоритм предусматривает сохранение информации о состоянии от одного цикла к другому, очень важно добиться того, чтобы правила не вырабатывали большого количества частичных соответствий. Например, рассмотрим следующую простую программу:

```
(deffacts information
   (find-match a c e g)
   (item a)
   (item b)
   (item c)
   (item d)
   (item e)
   (item f)
   (item g))
(defrule match-1
   (find-match ?x ?y ?z ?w)
   (item ?x)
   (item ?y)
   (item ?z)
   (item ?w)
   =>
   (assert (found-match ?x ?y ?z ?w)))
```

Сброс этой программы (переустановка в исходное состояние) происходит очень быстро. В этом позволяет убедиться команда `watch facts`, которая непосредственно следует за командой `reset`. А теперь рассмотрим следующую программу:

```
(deffacts information
   (find-match a c e g)
   (item a)
```

```
   (item b)
   (item c)
   (item d)
   (item e)
   (item f)
   (item g))
(defrule match-2
   (item ?x)
   (item ?y)
   (item ?z)
   (item ?w)
   (find-match ?x ?y ?z ?w)
   =>
   (assert (found-match ?x ?y ?z ?w)))
```

Команда `watch facts`, которая следует за командой `reset`, при выполнении данной программы демонстрирует значительное увеличение продолжительности сброса. По мере выполнения сброса быстро происходит внесение первых нескольких фактов из конструкции `deffacts` в список фактов. Но для внесения последующих фактов в список фактов требуется все более и более продолжительное время. И правило `match-1`, и правило `match-2` имеют одинаковые шаблоны, но сброс правила `match-2` занимает гораздо больше времени. А в действительности на некоторых компьютерах выполнение сброса правила `match-2` может вызвать то, что система CLIPS исчерпает доступную память. Еще более значительное различие в быстродействии может быть продемонстрировано путем добавления большего количества фактов `item` в конструкцию `deffacts` с именем `information` (а на некоторых компьютерах нельзя будет обнаружить заметного различия, не введя дополнительные факты `item` в конструкцию `deffacts` с именем `information`).

Учет характеристик правила `match-1`

Полезная информация может быть получена с помощью подсчета количества сопоставлений с шаблонами и частичных соответствий для правила `match-1`. Для иллюстрации выполняемых при этом действий в данном и следующих разделах используются листинги согласований с шаблонами и частичных соответствий. Для удобства чтения в этих листингах вместо полного текста всего факта применяются идентификаторы фактов. Кроме того, частичные соответствия обозначаются фигурными скобками. Идентификаторы фактов показаны в следующем списке:

```
f-1 (find-match a c e g)
f-2 (item a)
f-3 (item b)
```

```
f-4 (item c)
f-5 (item d)
f-6 (item e)
f-7 (item f)
f-8 (item g)
```

Правило `match-1` имеет такие сопоставления с шаблонами:

```
Pattern 1: f-1
Pattern 2: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 3: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 4: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 5: f-2, f-3, f-4, f-5, f-6, f-7, f-8
```

Правило `match-1` имеет следующие частичные соответствия:

```
Pattern 1: [f-1]
Patterns 1-2: [f-1,f-2]
Patterns 1-3: [f-1,f-2,f-4]
Patterns 1-4: [f-1,f-2,f-4,f-6]
Patterns 1-5: [f-1,f-2,f-4,f-6,f-8]
```

В целом правило `match-1` имеет 29 сопоставлений с шаблонами и 5 частичных соответствий.

Учет характеристик правила `match-2`

Правило `match-2` имеет следующие сопоставления с шаблонами:

```
Pattern 1: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 2: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 3: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 4: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 5: f-1
```

Правила `match-1` и `match-2` имеют одинаковое количество сопоставлений с шаблонами. А теперь рассмотрим только частичные соответствия для шаблона 1:

```
[f-2],[f-3],[f-4],[f-5],[f-6],[f-7],[f-8]
```

Шаблон 1 имеет семь частичных соответствий, и это неудивительно, поскольку для первого шаблона частичные соответствия и сопоставления с шаблонами должны быть одинаковыми. Но, как показано ниже, количество частичных соответствий для шаблонов 1 и 2 является довольно значительным.

```
[f-2,f-2],[f-2,f-3],[f-2,f-4],[f-2,f-5],
          [f-2,f-6],[f-2,f-7],[f-2,f-8],
[f-3,f-2],[f-3,f-3],[f-3,f-4],[f-3,f-5],
```

```
           [f-3,f-6],[f-3,f-7],[f-3,f-8],
[f-4,f-2],[f-4,f-3],[f-4,f-4],[f-4,f-5],
           [f-4,f-6],[f-4,f-7],[f-4,f-8],
[f-5,f-2],[f-5,f-3],[f-5,f-4],[f-5,f-5],
           [f-5,f-6],[f-5,f-7],[f-5,f-8],
[f-6,f-2],[f-6,f-3],[f-6,f-4],[f-6,f-5],
           [f-6,f-6],[f-6,f-7],[f-6,f-8],
[f-7,f-2],[f-7,f-3],[f-7,f-4],[f-7,f-5],
           [f-7,f-6],[f-7,f-7],[f-7,f-8],
[f-8,f-2],[f-8,f-3],[f-8,f-4],[f-8,f-5],
           [f-8,f-6],[f-8,f-7],[f-8,f-8]
```

Общее количество частичных соответствий для шаблонов 1 и 2 равно сорока девяти (семь сопоставлений с шаблонами для шаблона 1 умножаются на семь сопоставлений с шаблонами для шаблона 2). Из-за резкого увеличения потребности в памяти дальнейший рост количества частичных соответствий, которые можно ввести в список применительно к другим шаблонам, вскоре ограничивается, поскольку для шаблонов с 1 по 3 необходимо предусмотреть 343 частичных соответствия, а для шаблонов с 1 по 4 — 2401 частичное соответствие. Как и в случае правила `match-1`, для шаблонов с 1 по 5 имеется только одно частичное соответствие:

```
[f-2,f-4,f-6,f-8,f-1]
```

Важно отметить, что количество сопоставлений с шаблонами и количество активизаций для правил `match-1` и `match-2` одинаково, но правило `match-1` имеет только пять частичных соответствий, а правило `match-2` имеет 2801 частичное соответствие, причем разница между этими значениями количества продолжает расти по мере дальнейшего добавления фактов `item`. Факт `(item h)` не создает новые частичные соответствия для правила `match-1`, а для правила `match-2` создает 1880 новых частичных соответствий. Этот пример показывает, что из-за большого количества создаваемых частичных соответствий может резко снизиться производительность программы. При выборе эффективного набора правил необходимо стремиться не только к тому, чтобы свести к минимуму количество создаваемых новых частичных соответствий, но и количество удаляемых старых частичных соответствий. В действительности следует стремиться к тому, чтобы изменения состояния системы от одного цикла к другому были сведены к минимуму. Конкретные методы минимизации изменений состояния рассматриваются ниже в этой главе.

Команда matches

В языке CLIPS предусмотрена команда отладки, называемая **matches**, которая позволяет отобразить данные о количестве сопоставлений с шаблонами, частичных соответствий и активизаций некоторого правила. Эта команда позволяет находить правила, при использовании которых вырабатывается большое количество частичных соответствий, а также упрощает отладку в тех ситуациях, когда на первый взгляд проверка всех шаблонов правила завершается успешно, но несмотря на это, правило не активизируется. Команда matches имеет следующий синтаксис:

```
(matches <rule-name>)
```

Параметром команды matches является имя правила, для которого должно быть отображено количество сопоставлений и соответствий. Пример вывода команды matches показан в следующем диалоге:

```
CLIPS> (clear)↵
CLIPS>
(defrule match-3
   (find-match ?x ?y)
   (item ?x)
   (item ?y)
   =>
   (assert (found-match ?x ?y)))↵
   CLIPS>
(assert (find-match a b)
   (find-match c d)
   (find-match e f)
   (item a)
   (item b)
   (item c)
   (item f))↵
<Fact-6>
CLIPS> (facts)↵
f-0      (find-match a b)
f-1      (find-match c d)
f-2      (find-match e f)
f-3      (item a)
f-4      (item b)
f-5      (item c)
f-6      (item f)
For a total of
7 facts.
```

```
CLIPS> (matches match-3)↵
Matches for Pattern 1
f-0
f-1
f-2
Matches for Pattern 2
f-3
f-4
f-5
f-6
Matches for Pattern 3
f-3
f-4
f-5
f-6
Partial matches for
CEs 1 - 2
f-1,f-5
f-0,f-3
Partial matches
for CEs 1 - 3
f-0,f-3,f-4
Activations
f-0,f-3,f-4
CLIPS>
```

Первый шаблон правила `match-3` имеет три сопоставления с шаблонами, по одному для каждого факта `find-match`. Аналогичным образом, и второй и третий шаблоны имеют четыре сопоставления с шаблонами, по одному для каждого факта `item`. Первые два шаблона имеют два частичных соответствия: один относится к фактам `(find-match c d)` и `(item c)`, а второй — к фактам `(find-match c d)` и `(item c)`. С фактом `(find-match e f)` не связано ни одного частичного соответствия, поскольку факт `(item e)` не существует. Для всех трех шаблонов имеется только одно частичное соответствие — частичное соответствие для фактов `(fact-match a b)`, `(item a)` и `(item b)`. Для этого частичного соответствия имеется также одна активизация. А после запуска правила `match-3` применительно к этой активизации в выводе команды `matches` соответствующая информация больше не отображается.

Наблюдение за изменяющимся состоянием

Команда matches предоставляет удобный способ исследования частичных соответствий, относящихся к некоторому правилу. Еще одним способом слежения за частичными соответствиями является способ, в котором такие соответствия рассматриваются как частичные активизации правил. Если правило match-1 будет рассматриваться как несколько отдельных правил, в каждом из которых предпринимается попытка вычисления частичных соответствий, то для просмотра этих частичных соответствий по мере их формирования может использоваться команда watch activations. Правило match-1 может быть разбито на несколько правил следующим образом:

```
(defrule m1-pm-1
   "Partial matches for pattern 1"
   (find-match ?x ?y ?z ?w)
   =>)
(defrule m1-pm-1-to-2
   "Partial matches for patterns 1 and 2"
   (find-match ?x ?y ?z ?w)
   (item ?x)
   =>)
(defrule m1-pm-1-to-3
   "Partial matches for patterns 1 to 3"
   (find-match ?x ?y ?z ?w)
   (item ?x)
   (item ?y)
   =>)
(defrule m1-pm-1-to-4
   "Partial matches for patterns 1 to 4"
   (find-match ?x ?y ?z ?w)
   (item ?x)
   (item ?y)
   (item ?z)
   =>)
(defrule match-1 "Activations for the match rule"
   (find-match ?x ?y ?z ?w)
   (item ?x)
   (item ?y)
   (item ?z)
   (item ?w)
   =>
   (assert (found-match ?x ?y ?z ?w)))
```

После загрузки приведенных выше правил и конструкции `deffacts` с именем `information` можно организовать отслеживание частичных соответствий в ходе их создания с помощью команд, показанных в следующем примере диалогового выполнения команд:

```
CLIPS> (watch activations)↵
CLIPS>
(watch facts)↵
CLIPS> (reset)↵
==> f-0     (initial-fact)
==> f-1     (find-match a c e g)
==> Activation 0     m1-pm-1: f-1
==> f-2     (item a)
==> Activation 0     m1-pm-1-to-2: f-1,f-2
==> f-3     (item b)
==> f-4     (item c)
==> Activation 0     m1-pm-1-to-3: f-1,f-2,f-4
==> f-5     (item d)
==> f-6     (item e)
==> Activation 0     m1-pm-1-to-4: f-1,f-2,f-4,f-6
==> f-7     (item f)
==> f-8     (item g)
==> Activation 0     match-1: f-1,f-2,f-4,f-6,f-8
CLIPS>
```

После применения такого же метода для контроля над выполнением правила `matches-2` вырабатываются сотни частичных активизаций. Теперь должно быть ясно, что если речь идет о повышении эффективности, то левую часть правила нельзя рассматривать как единое целое. Каждую левую часть каждого правила следует считать несколькими отдельными правилами и предпринимать предположение, что каждое из этих правил вырабатывает множество частичных соответствий для всего правила в целом. Поэтому задача написания эффективных правил связана с ограничением не только общего количества активизаций для правила, но и количества частичных соответствий для каждого из отдельных субправил, которые формируют левую часть правила.

9.13 Упорядочение шаблонов в целях повышения эффективности

При решении задачи упорядочения шаблонов в целях ограничения количества создаваемых частичных соответствий следует руководствоваться несколькими ре-

комендациями. Иногда бывает нелегко найти наилучший способ упорядочения шаблонов, поскольку одни рекомендации противоречат другим. Вообще говоря, эти рекомендации по упорядочению должны использоваться для предотвращения возникновения наиболее существенных неэффективностей, которые могут проявляться в системе, основанной на правилах. В ходе настройки производительности экспертной системы может потребоваться применить значительное количество вариантов переупорядочения шаблонов по методу проб и ошибок, чтобы определить, в результате каких изменений система будет действовать быстрее. Часто бывает также, что намного лучшие результаты могут быть достигнуты после опробования полностью другого подхода, а не осуществления попыток точной настройки шаблонов.

Первоочередное размещение наиболее конкретных шаблонов

Наиболее конкретные шаблоны должны размещаться ближе к началу левой части правила. Наиболее конкретный шаблон, вообще говоря, характеризуется тем, что к нему относится меньшее количество согласующихся фактов в списке фактов и наибольшее количество связываний переменных, ограничивающих другие шаблоны. Например, конкретным является шаблон `(match ?x ?y ?z ?w)`, показанный в правилах `match-1` и `match-2`, поскольку он ограничивает значения фактов, разрешающие вырабатывать частичные соответствия для четырех других шаблонов правила.

Размещение в последнюю очередь шаблонов, согласующихся с непостоянными фактами

Шаблоны, которые согласуются с фактами, часто добавляемыми и удаляемыми из списка фактов, должны размещаться ближе к концу левой части правила. Благодаря этому количество изменений в частичных соответствиях для данного правила становится наименьшим. Важно отметить, что шаблоны, согласующиеся с непостоянными фактами, часто бывают наиболее конкретными шаблонами в правиле. В связи с этим может возникнуть противоречие при осуществлении попытки переупорядочить шаблоны в правиле для достижения максимальной эффективности. Например, обычно лучше всего размещать управляющие факты в качестве начального шаблона правила, поскольку при отсутствии управляющего факта не будут вырабатываться частичные соответствия для этого правила. Но если внесение управляющих фактов и их извлечение из списка фактов происходят весьма часто, в связи с чем приходится то и дело повторно вычислять большое количество частичных соответствий, то размещение управляющего факта ближе к концу правила может стать более эффективным.

Первоочередное размещение шаблонов, сопоставляющихся с наименьшим количеством фактов

Размещение ближе к началу правила таких шаблонов, которые сопоставляются лишь с немногими фактами в списке фактов, позволяет уменьшить количество частичных соответствий, которые могут быть выработаны. Еще раз отметим, что эта рекомендация может конфликтовать с другими рекомендациями. Шаблон, сопоставляющийся лишь с немногими фактами, не обязательно является наиболее конкретным шаблоном; еще одна особенность этого шаблона может состоять в том, что он согласуется с непостоянными фактами.

9.14 Многозначные переменные и эффективность

Благодаря использованию многозначных подстановочных символов и многозначных переменных возможности сопоставления с шаблонами значительно возрастают. Но непродуманное применение этих возможностей часто приводит к снижению эффективности. Прибегая к использованию многозначных подстановочных символов и переменных, необходимо руководствоваться двумя рекомендациями, описанными в этом разделе. Во-первых, эти конструкции должны применяться, только если они действительно необходимы. Во-вторых, при использовании указанных конструкций нужно следить за тем, чтобы количество многозначных подстановочных символов и переменных в единственном слоте шаблона было ограничено. Ниже приведено правило, которое показывает, что многозначные подстановочные символы и переменные могут оказаться полезными, но вместе с тем весьма дорогостоящими с точки зрения производительности.

```
(defrule produce-twoplets
   (list (items $?b $?m $?e))
   =>
   (assert (front ?b))
   (assert (middle ?m))
   (assert (back ?e)))
```

После внесения в список фактов такого факта, как `(list (items a 4 z 2))`, данное правило вырабатывает факты, представляющие начальную `(front)`, среднюю `(middle)` и конечную `(back)` части списка. Значения длины этих различных частей изменяются от нуля до длины списка. Безусловно, данное правило легко сформулировать с помощью многозначных переменных, но операция его согласования с шаблонами является чрезвычайно дорогостоящей. В табл. 9.1 показаны все согласования, попытки выполнения которых были предприняты, а также продемонстрировано, что многозначные подстановочные символы и переменные

способны выполнить значительную часть работы в составе процесса сопоставления с шаблонами. Вообще говоря, для этого правила `produce-twoplets` происходит `(N2 + 3N + 2) / 2` согласований в расчете на `N` полей, содержащихся в факте `items`.

Таблица 9.1. Анализ попыток согласования для трех многозначных переменных

Попытка согласования	Поля, согласующиеся с переменной `$?b`	Поля, согласующиеся с переменной `$?m`	Поля, согласующиеся с переменной `$?e`
1			a 4 z 2
2		a	4 z 2
3		a 4	z 2
4		a 4 z	2
5		a 4 z 2	
6	a		4 z 2
7	a	4	z 2
8	a	4 z	2
9	a	4 z 2	
10	a 4		z 2
11	a 4	z	2
12	a 4	z 2	
13	a 4 z		2
14	a 4 z	2	
15	a 4 z 2		

9.15 Условный элемент `test` и эффективность программы

Любой условный элемент `test`, находящийся в правиле, должен размещаться как можно ближе к началу правила. Например, рассмотрим следующее правило, в котором осуществляются проверки для поиска трех различных точек:

```
(defrule three-distinct-points
   ?point-1 <- (point (x ?x1) (y ?y1))
   ?point-2 <- (point (x ?x2) (y ?y2))
   ?point-3 <- (point (x ?x3) (y ?y3))
   (test (and (neq ?point-1 ?point-2)
              (neq ?point-2 ?point-3)
              (neq ?point-1 ?point-3)))
```

```
   =>
   (assert (distinct-points (x1 ?x1) (y1 ?y1)
                            (x2 ?x2) (y2 ?y2)
                            (x3 ?x3) (y3 ?y3))))
```

Условный элемент test, позволяющий определить, что адрес факта ?point-1 не совпадает с адресом факта ?point-2, может быть помещен непосредственно после второго шаблона. Размещение условного элемента test в этой точке позволяет уменьшить количество создаваемых частичных соответствий, как показано ниже.

```
(defrule three-distinct-points
   ?point-1 <- (point (x ?x1) (y ?y1))
   ?point-2 <- (point (x ?x2) (y ?y2))
   (test (neq ?point-1 ?point-2))
   ?point-3 <- (point (x ?x3) (y ?y3))
   (test (and (neq ?point-2 ?point-3)
              (neq ?point-1 ?point-3)))
   =>
   (assert (distinct-points (x1 ?x1) (y1 ?y1)
                            (x2 ?x2) (y2 ?y2)
                            (x3 ?x3) (y3 ?y3))))
```

При формировании частичных соответствий в сети соединений всегда происходит вычисление условных элементов test в левой части любого правила. Выражения, используемые с предикативным ограничением или ограничением равенства поля, должны вычисляться в процессе сопоставления с шаблонами, если выполняются некоторые условия. Вычисление выражения в процессе сопоставления с шаблонами в сети шаблонов позволяет добиться повышения эффективности. А выражения, используемые в предикативных ограничениях или ограничениях поля возвращаемого значения, должны осуществляться в процессе сопоставления с шаблонами, если все переменные, упоминаемые в этом выражении, можно найти в том шаблоне, который включает это выражение.

Например, выражение в приведенном ниже правиле будет вычисляться во время формирования частичных соответствий, поскольку оно находится в условном элементе test.

```
(defrule points-share-common-x-or-y-value
   (point (x ?x1) (y ?y1))
   (point (x ?x2) (y ?y2))
   (test (or (= ?x1 ?x2) (= ?y1 ?y2)))
   =>
   (assert (common-x-or-y-value
```

```
            (x1 ?x1) (y1 ?y1)
            (x2 ?x2) (y2 ?y2))))
```

А в следующем примере размещение выражения в шаблоне не приведет к тому, что его вычисление будет осуществляться в процессе сопоставления с шаблонами, поскольку переменные `?x1` и `?y1` не содержатся во втором шаблоне:

```
(defrule points-share-common-x-or-y-value
   (point (x ?x1) (y ?y1))
   (point (x ?x2) (y ?y2&:(or (= ?x1 ?x2)
                              (= ?y1 ?y2))))
   =>
   (assert (common-x-or-y-value
            (x1 ?x1) (y1 ?y1)
            (x2 ?x2) (y2 ?y2))))
```

Еще раз отметим, что выражение в следующем правиле будет вычисляться во время формирования частичных соответствий, поскольку оно находится в условном элементе `test`:

```
(defrule point-not-on-x-y-diagonals ""
   (point (x ?x1) (y ?y1))
   (test (and (<> ?x1 ?y1) (<> ?x1 (- 0 ?y1))))
   =>
   (assert (non-diagonal-point (x ?x1) (y ?y1))))
```

Но на этот раз размещение выражения в шаблоне обеспечивает возможность его вычисления в процессе сопоставления с шаблонами, поскольку обе переменные, `?x1` и `?y1`, находятся в шаблоне, который включает это выражение:

```
(defrule point-not-on-x-y-diagonals
   (point (x ?x1)
   (y ?y1&:(and (<> ?x1 ?y1)
   (<> ?x1 (- 0 ?y1)))))
   =>
   (assert (non-diagonal-point (x ?x1) (y ?y1))))
```

9.16 Встроенные ограничения сопоставления с шаблонами

Встроенные ограничения сопоставления с шаблонами всегда являются более эффективными по сравнению с эквивалентным выражением, которое должно быть вычислено. Например, такое правило:

```
(defrule primary-color
```

```
   (color ?x&:(or (eq ?x red)
                  (eq ?x green)
                  (eq ?x blue)))
   =>
   (assert (primary-color ?x)))
```

не должно использоваться, если для достижения того же результата можно применить ограничения сопоставления с шаблонами, как показано ниже.

```
(defrule primary-color
   (color ?x&red | green | blue)
   =>
   (assert (primary-color ?x)))
```

9.17 Сравнение общих правил с конкретными правилами

Не всегда возможно сразу же дать ответ на вопрос о том, будут ли конкретные правила, представленные в большом количестве, функционировать эффективнее по сравнению с более общими, но менее многочисленными правилами. В результате применения конкретных правил процесс сопоставления с шаблонами в большей степени сосредоточивается в сети шаблонов, а объем работы в сети соединений уменьшается. Общие правила часто предоставляют больше возможностей для совместного использования в сетях шаблонов и соединений. Кроме того, одно общее правило может предоставлять большее удобство сопровождения по сравнению с более крупной группой конкретных правил. Но написание общих правил должно осуществляться весьма продуманно. Дело в том, что общие правила должны выполнять такую же работу, как и несколько конкретных правил, поэтому гораздо легче упустить что-то важное и написать неэффективное общее правило, чем неэффективное конкретное правило. Для иллюстрации различий между этими двумя методами рассмотрим приведенную ниже конструкцию `deftemplate` и четыре правила, которые должны обновлять факт, содержащий прямоугольные координаты объекта, который может перемещаться на север (`north`), юг (`south`), восток (`east`) или запад (`west`). Координаты `x` и `y` объекта находятся в факте `location`. После перемещения на север увеличивается значение координаты `y`, а после перемещения на восток увеличивается значение координаты `x`.

```
(deftemplate location (slot x) (slot y))
(defrule move-north
   (move north)
   ?old-location <- (location (y ?old-y))
```

```
   =>
   (modify ?old-location (y (+ ?old-y 1))))
(defrule move-south
   (move south)
   ?old-location <- (location (y ?old-y))
   =>
   (modify ?old-location (y (- ?old-y 1))))
(defrule move-east
   (move east)
   ?old-location <- (location (x ?old-x))
   =>
   (modify ?old-location (x (+ ?old-x 1))))
(defrule move-west
   (move west)
   ?old-location <- (location (x ?old-x))
   =>
   (modify ?old-location (x (- ?old-x 1))))
```

Четыре приведенных выше правила можно заменить одним более общим правилом, дополнительной конструкцией `deftemplate` и конструкцией `deffacts`:

```
(deftemplate direction
   (slot which-way)
   (slot delta-x)
   (slot delta-y))
(deffacts direction-information
   (direction (which-way north)
              (delta-x 0) (delta-y 1))
   (direction (which-way south)
              (delta-x 0) (delta-y -1))
   (direction (which-way east)
              (delta-x 1) (delta-y 0))
   (direction (which-way west)
              (delta-x -1) (delta-y 0)))
(defrule move-direction
   (move ?dir)
   (direction (which-way ?dir)
              (delta-x ?dx)
              (delta-y ?dy))
   ?old-location <- (location (x ?old-x)
                              (y ?old-y))
   =>
```

```
(modify ?old-location (x (+ ?old-x ?dx))
                      (y (+ ?old-y ?dy))))
```

В переменных `?dx` и `?dy` используются значения дельта `x` и дельта `y`, которые должны суммироваться со значениями `x` и `y`, относящимися к прежнему местонахождению, для получения значений `x` и `y`, относящихся к новому местонахождению.

При использовании этого нового правила требуется больше работы по созданию частичных соответствий для определения значений дельта `x` и дельта `y`, которые должны суммироваться с координатами текущего местонахождения. Но такой подход позволяет создать дополнительный уровень абстракции, благодаря чему упрощается возможность введения новых направлений перемещения. Например, чтобы обеспечить перемещение на северо-восток, юго-восток, северо-запад и юго-запад, потребовалось бы добавить четыре новых конкретных правила, а в обобщенном примере требуется лишь ввести четыре новых факта в конструкцию `deffacts` следующим образом:

```
(deffacts direction-information
   (direction (which-way north)
              (delta-x 0) (delta-y 1))
   (direction (which-way south)
              (delta-x 0) (delta-y -1))
   (direction (which-way east)
              (delta-x 1) (delta-y 0))
   (direction (which-way west)
              (delta-x -1) (delta-y 0))
   (direction (which-way northeast)
              (delta-x 1) (delta-y 1))
   (direction (which-way southeast)
              (delta-x 1) (delta-y -1))
   (direction (which-way northwest)
              (delta-x -1) (delta-y 1))
   (direction (which-way southwest)
              (delta-x -1) (delta-y -1)))
```

9.18 Сравнение простых правил со сложными правилами

Языки, основанные на правилах, позволяют сформулировать многие задачи в простой, но изящной форме. В частности, с помощью языка CLIPS можно легко найти наибольшее число из группы чисел, хотя он не предназначен специаль-

но для решения алгоритмических или вычислительных задач. Ниже приведены правило и относящаяся в нему конструкция deffacts, обеспечивающие внесение в список фактов группы фактов с примерами чисел, которые используются в качестве проверочных данных в правиле, отыскивающем наибольшее число.

```
(deffacts max-num
   (loop-max 100))
(defrule loop-assert
   (loop-max ?n)
   =>
   (bind ?i 1)
   (while (<= ?i ?n) do
      (assert (number ?i))
      (bind ?i (+ ?i 1))))
```

Простейший способ поиска наибольшего числа был продемонстрирован в главе 8. Как показано ниже, чтобы найти наибольшее число, достаточно применить лишь одно правило.

```
(defrule largest-number
   (number ?number1)
   (not (number ?number2&:(> ?number2 ?number1)))
   =>
   (printout t "Largest number is " ?number1 crlf))
```

Безусловно, это правило является несложным, но не представляет собой самый быстрый способ поиска наибольшего числа. Если N показывает количество фактов в отношении number, то и в первом, и во втором шаблоне должно быть проведено количество сопоставлений с шаблонами, равное N. Даже несмотря на то, что количество частичных соответствий для первых двух шаблонов равно лишь единице, количество операций сравнения, которые должны быть выполнены для нахождения этого частичного соответствия, равно N в квадрате. Увеличение значения факта loop-max до 200, 300, 400 и т.д. показывает, что затраты времени на решение данной задачи пропорциональны квадрату N.

Операция сравнения такого типа является чрезвычайно неэффективной, поскольку после добавления каждого числа оно сравнивается со всеми другими числами для определения того, не является ли оно наибольшим. Например, если факты, представляющие числа от 2 до 100, уже были внесены в список фактов, после чего внесен факт, представляющий число 1, то необходимо выполнить 199 операций сравнения для определения того, является ли число 1 наибольшим. Дело в том, что факт (number 1) согласуется с первым шаблоном, поэтому его необходимо будет сравнить со всеми 99 фактами, соответствующими второму шаблону, чтобы определить, представляет ли он наибольшее число. Первый ряд операций сравнения окончится неудачей (поскольку все числа от 2 до 100

больше 1), но все равно придется провести сравнение для всех оставшихся чисел. Аналогичным образом, факт (number 1) согласуется со вторым шаблоном, поэтому его необходимо будет сравнить со 100 фактами, согласующимися с первым шаблоном (который теперь включает также факт (number 1)).

Ключом к ускорению работы этой программы является предотвращение возможности выполнения ненужных сравнений. Такую задачу можно решить, используя дополнительный факт для отслеживания значения наибольшего числа и сравнивая с ним факты number. Ниже приведены правила, которые показывают, как осуществить этот замысел.

```
(defrule try-number
   (number ?n)
   =>
   (assert (try-number ?n)))
(defrule largest-unknown
   ?attempt <- (try-number ?n)
   (not (largest ?))
   =>
   (retract ?attempt)
   (assert (largest ?n)))
(defrule largest-smaller
   ?old-largest <- (largest ?n1)
   ?attempt <- (try-number ?n2&:(> ?n2 ?n1))
   =>
   (retract ?old-largest ?attempt)
   (assert (largest ?n2)))
(defrule largest-bigger
   (largest ?n1)
   ?attempt <- (try-number ?n2&:(<= ?n2 ?n1))
   =>
   (retract ?attempt))
(defrule print-largest
   (declare (salience -1))
   (largest ?number)
   =>
   (printout t "Largest number is " ?number crlf))
```

Вызов данной программы на выполнение со значениями max-loop, равными 100, 200, 300, 400 и т.д., показывает, что продолжительность прогона программы пропорциональна N. Этот результат становится еще более любопытным, если учесть, что в первой программе происходил запуск только двух правил, тогда как во второй программе запускаются приблизительно 2N правил. Вторая группа пра-

вил показывает, что в правиле следует пытаться ограничивать не только количество имеющихся в нем частичных соответствий, но и количество сравнений, необходимых для определения частичных соответствий. Как показало первое правило, если для первого шаблона имеется N согласований и для второго шаблона имеется N согласований, то в правиле выполняется N в квадрате операций сравнения при определении частичных соответствий для первых двух шаблонов. Даже если частичные соответствия не вырабатываются, время, затраченное на вычисления, будет примерно эквивалентно тому времени, которое требуется для выработки частичных соответствий в количестве N в квадрате. Вторая группа правил уменьшает количество сравнений, которые должны быть выполнены, до N. Одновременно существует только один факт `largest` и один факт `try-number`, поэтому для правил `largest-unknown`, `largest-smaller` и `largest-bigger` в любой момент времени имеется лишь одно частичное соответствие. А поскольку вырабатываются N фактов `try-number`, то время вычисления ограничивается формированием N частичных соответствий. На первый взгляд может показаться, что возможна выработка больше одного факта `try-number` (что вызовет появление N в квадрате частичных соответствий). Но, как было сказано в разделе 9.3, рабочий список правил действует по принципу стека. Это означает, что для всех активизаций `try-number`, помещенных в рабочий список правил с помощью правила `loop-assert`, запуск всегда будет осуществляться после любых активизаций правил `largest-unknown`, `largest-smaller` и `largest-bigger`. А поскольку эти правила всегда удаляют текущий факт `try-number`, то количество фактов такого типа никогда не превысит единицу.

Этот пример демонстрирует два важных понятия. Во-первых, самый простой способ разработки кода для решения задачи на языке, основанном на правилах, не всегда является наилучшим. Во-вторых, количество выполняемых сравнений часто можно уменьшить, используя для хранения данных временные факты. В рассматриваемой задаче для хранения значения, применяемого в последующих операциях сравнения, служил факт `largest`, поэтому во время сравнения не приходилось выполнять поиск среди всех фактов `number`.

9.19 Резюме

В настоящей главе приведено вводное описание различных средств CLIPS, позволяющих обеспечить разработку надежных экспертных систем. Атрибуты `deftemplate` обеспечивают принудительное применение ограничений типов и значений, что позволяет предотвратить возникновение не только опечаток, но и семантических ошибок. Проверка соблюдения ограничений может осуществляться статически (на этапе определения выражений или конструкций) или динамически (на этапе вычисления выражений). Атрибут `type` позволяет налагать

ограничения на значения, разрешенные для применения в определенном слоте. Атрибуты допустимого значения позволяют ограничивать значения, разрешенные для использования в слоте, указанным списком. Атрибут `range` дает возможность ограничивать числовые значения заданным диапазоном. Атрибут `cardinality` позволяет регламентировать минимальное и максимальное количества полей, хранимых в многозначном слоте. Два других атрибута конструкции `deftemplate`, атрибуты `default` и `default-dynamic`, не ограничивают значения слотов, но позволяют задать начальное значение слота конструкции `deftemplate`.

Еще более сложные структуры управления могут быть созданы с помощью такого механизма, как значимость. Значимость используется для такого определения приоритетов правил, чтобы запуск активизированных правил с наивысшей значимостью осуществлялся в первую очередь. Средства определения значимости могут применяться в сочетании с управляющими фактами для отделения экспертных знаний от управляющих знаний.

Конструкция `defmodule` позволяет секционировать базу знаний. В свою очередь, модули позволяют явно указывать, какие конструкции `deftemplate` импортируются из других модулей и экспортируются в другие модули, поэтому дают возможность управлять тем, какие факты окажутся видимыми в этих модулях. Применение команды `focus` позволяет управлять исполнением программы без использования значимости; для этого правила секционируются на группы и размещаются в отдельных модулях.

В настоящей главе показано, насколько важно обеспечить эффективное согласование фактов с правилами. Эффективным средством осуществления такого процесса согласования является rete-алгоритм, поскольку в этом алгоритме продуктивно применяются свойства временно́й и структурной избыточности, проявляющиеся в экспертных системах, основанных на правилах.

Правила преобразуются в структуры данных в сети правил. Такая сеть состоит из сети шаблонов и сети соединений. В сети шаблонов факты согласуются с шаблонами, а сеть соединений обеспечивает единообразие связываний переменных в различных шаблонах. На производительность любого правила может оказывать существенное влияние порядок расположения шаблонов в этом правиле. Вообще говоря, наиболее конкретные шаблоны и шаблоны, согласующиеся с наименьшим количеством фактов, должны размещаться ближе к началу левой части правила, а шаблоны, согласующиеся с непостоянными фактами, должны размещаться ближе к концу левой части правила. Для отображения сопоставлений с шаблонами, частичных соответствий и активизаций правила используется команда `matches`.

Для повышения эффективности правил применяется несколько методов, включая надлежащее использование многозначных переменных, правильное позиционирование проверочных шаблонов и использование встроенных ограничений сопоставления с шаблонами. Кроме того, при достижении оптимальной эффек-

тивности необходимо учитывать компромиссы, связанные с применением общих или конкретных правил, а также простых или сложных правил.

Задачи

9.1. Модифицируйте программу Sticks, описанную в главе 8, так, чтобы управляющие правила были отделены от правил ведения игры. Для назначения управляющим правилам более низкого приоритета используйте значимость (атрибут `salience`).

9.2. Добавьте к программе для "мира блоков", приведенной в разделе 7.23, такое правило, которое удаляет цель хода, если эта цель уже выполнена.

9.3. Создайте правила для реализации процедуры принятия решений, позволяющей определить, является ли силлогизм действительным. Проверьте разработанную программу на силлогизме, приведенном в задаче 8.24 (см. с. 688).

9.4. Напишите программу, позволяющую определить простые множители числа. Например, простыми множителями числа 15 являются 3 и 5.

9.5. По приведенным в табл. 9.2 данным о расстояниях между городами в штате Техас найдите решение задачи коммивояжера (см. с. 105 в разделе 1.13) для указанных городов. Напишите программу, позволяющую найти самый короткий маршрут посещения всех городов. В качестве входных данных для программы должен применяться город, с которого начинается маршрут, и список посещаемых городов. Для проверки программы определите самый короткий маршрут, начинающийся в г. Хьюстон (Houston).

Таблица 9.2. Расстояния между городами в штате Техасе

	Houston	**Dallas**	**Austin**	**Abilene**	**Waco**
Houston	—	241	162	351	183
Dallas	241	—	202	186	97
Austin	162	202	—	216	106
Abilene	351	186	216	—	186
Waco	183	97	106	186	—

9.6. При условии, что дана следующая информация, напишите программу, которая запрашивает у пользователя данные о типе видимых облаков и направлении ветра, а затем выдает прогноз с указанием вероятности дождя. Кучевые облака указывают на то, что стоит хорошая погода, но они могут превратиться в слоисто-дождевые облака, если дует ветер, имеющий

направление от северо-восточного до южного. Перисто-кучевые облака указывают на то, что в течение суток пройдет дождь, если дует ветер, имеющий направление от северо-восточного до южного. Если дует ветер, имеющий направление от северного до западного, то может быть предсказана пасмурная погода. Слоисто-кучевые облака могут превратиться в кучево-дождевые облака, если дует ветер, имеющий направление от северо-восточного до южного. Слоистые облака указывают на возможность легкого дождя. Если дует ветер, имеющий направление от северо-восточного до южного, то пройдет продолжительный дождь. Слоисто-дождевые облака указывают на возможность кратковременного дождя, если дует ветер, имеющий направление от юго-западного до северного. Продолжительный дождь возможен, если дует ветер, имеющий направление от северо-восточного до южного. Кучево-дождевые облака свидетельствуют о возможности ливня, если они появляются перед полуднем. Перисто-слоистые облака указывают на возможность дождя в течение 15–24 часов, если дует ветер, имеющий направление от северо-восточного до южного. Высокослоистые облака указывают на возможность дождя в течение суток, если дует ветер, имеющий направление от северо-восточного до южного, а в ином случае будет пасмурная погода. Высококучевые облака указывают на возможность дождя в течение 15–20 часов, если дует ветер, имеющий направление от северо-восточного до южного.

9.7. Напишите программу для преобразования сообщения, заданного в виде азбуки Морзе, в эквивалентный этому сообщению ряд знаков алфавита. Ниже приведен пример входных и выходных данных этой программы (где * и – обозначают точки и тире, а знак / используется для отделения друг от друга символов азбуки Морзе):

```
Enter a message (<CR> to end): * * * / - - - /
   * * *↵
The message is S O S
Enter a message (<CR> to end): ↵
CLIPS>
```

Коды азбуки Морзе и эквивалентные им знаки алфавита приведены на рис. 9.12.

9.8. Напишите программу, которая после ввода в нее выражения, состоящего из чисел и обозначений единиц измерения, преобразует все единицы в этом выражении в набор базовых единиц измерения (таких как метры, секунды, килограммы, пенсы и амперы). Ниже приведен пример входных и выходных данных этой программы.

```
Enter an expression (<CR> to end): 30 meters /
   minute↵
```

A	• –	H	• • • •	O	– – –	V	• • • –
B	– • • •	I	• •	P	• – – •	W	• – –
C	– • – •	J	• – – –	Q	– – • –	X	– • • –
D	– • •	K	– • –	R	• – •	Y	– • – –
E	•	L	• – • •	S	• • •	Z	– – • •
F	• • – •	M	– –	T	–		
G	– – •	N	– •	U	• • –		

Рис. 9.12. Коды азбуки Морзе и эквивалентные им знаки алфавита

```
Conversion is 0.5 m / s
Enter an expression (<CR> to end): ↵
CLIPS>
```

9.9. Напишите программу для ведения игры Life (широко распространенная игра, в которой моделируются клеточные автоматы). Предположим, что имеется двумерный массив клеток, в котором каждая клетка является либо мертвой, либо живой. Состояние клеток следующего поколения определяется на основе следующих правил. Любая живая клетка, которая является смежной точно с двумя или тремя другими живыми клетками, продолжает жить. Любая живая клетка, которая является смежной меньше чем с двумя или больше чем с тремя другими живыми клетками, умирает. Любая мертвая клетка, которая является смежной точно с тремя другими живыми клетками, становится живой. Например, предположим, что первое поколение было представлено в виде массива 5×5, в котором живые клетки обозначены точками (рис. 9.13).

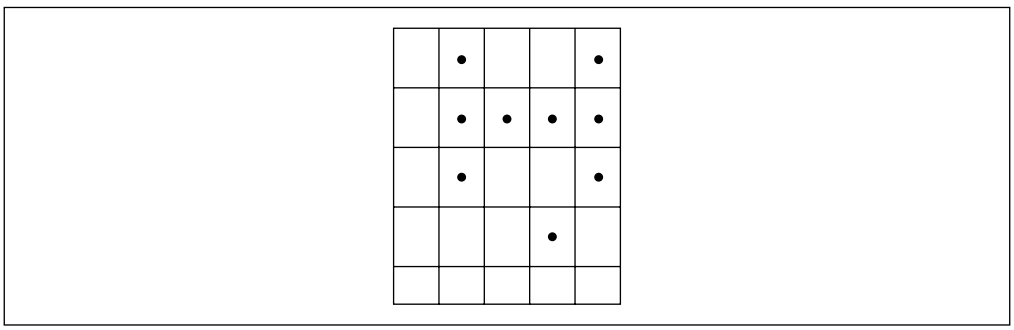


Рис. 9.13. Первое поколение клеток в рассматриваемом примере игры Life

В таком случае следующее поколение выглядело бы так, как показано на рис. 9.14.

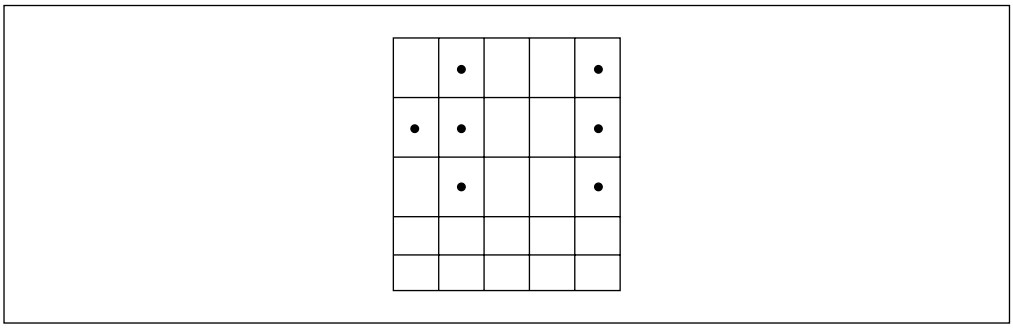


Рис. 9.14. Второе поколение клеток в рассматриваемом примере игры Life

Вычислите состояние игры для следующих четырех поколений, используя начальную конфигурацию, приведенную на рис. 9.13.

9.10. Четырехугольник — это фигура с четырьмя сторонами. Четырехугольник именуется равнобедренным, если он имеет две разные пары сторон одинаковой длины, следующих друг за другом по периметру. Четырехугольник называется трапецоидом, если он имеет по крайней мере одну пару параллельных сторон. Четырехугольник называется параллелограммом, если обе пары его противоположных сторон параллельны. Четырехугольник называется ромбом, если все его четыре стороны равны по длине. Четырехугольник называется прямоугольником, если он имеет четыре прямых угла. Четырехугольник называется квадратом, если он имеет четыре равных стороны и четыре прямых угла. Обратите внимание на то, что ромб представляет собой сочетание равнобедренного четырехугольника и параллелограмма, параллелограмм представляет собой трапецоид, прямоугольник представляет собой параллелограмм, а квадрат представляет собой сочетание ромба и прямоугольника.

Напишите программу, которая после получения координат четырех точек, образующих четырехугольник, определяет тип четырехугольника. В программе должна учитываться возможная ошибка округления (примите предположение, что две стороны равны, если разница между значениями их длины не превышает 0,00001). Проверьте разработанную вами программу по приведенным ниже данным о четырехугольниках.

а) Точки $(0,0)$, $(2,4)$, $(6,0)$ и $(3,2)$.

б) Точки $(0,3)$, $(2,5)$, $(4,3)$ и $(2,0)$.

в) Точки $(0,0)$, $(3,2)$, $(4,2)$ и $(9,0)$.

г) Точки $(0,0)$, $(1,3)$, $(5,3)$ и $(4,0)$.

д) Точки $(0,0)$, $(3,5.196152)$, $(9,5.196152)$ и $(6,0)$.

е) Точки $(0,0)$, $(0,4)$, $(2,4)$ и $(2,0)$.

ж) Точки $(0,2)$, $(4,6)$, $(6,4)$ и $(2,0)$.

з) Точки $(0,2)$, $(2,4)$, $(4,2)$ и $(2,0)$.

Подсказка. Если сторона 1 содержит точки $(x1,y1)$ и $(x2,y2)$ и сторона 2 содержит точки $(x3,y3)$ и $(x4,y4)$, то сторона 1 параллельна стороне 2, если $(x2-x1)*(y4-y3)$ равно $(x4-x3)*(y2-y1)$.

9.11. Напишите программу CLIPS, которая может определить, является ли простое предложение грамматически правильным. Грамматически правильные предложения должны соответствовать следующему определению, представленному в виде нормальной формы Бэкуса–Наура:

```
<sentence> ::= <verb> <direct-object>
                       [<indirect-object>]
<direct-object> ::= [<determiner>] <adjective>* <noun>
<indirect-object> ::= <preposition> <direct-object>
<determiner> ::= a | an | the
<adjective> ::= red | shiny | heavy
<noun> ::= ball | wrench | gun | pliers
<preposition> ::= with  in | at
<verb> ::= get | throw | shoot
```

В качестве примера можно привести следующий диалог:

```
Enter a sentence (<CR> to end): shoot the red shiny
gun at the pliers↵
OK
Enter a sentence (<CR> to end): gun shoot↵
I don't understand.
Enter a sentence (<CR> to end):
CLIPS>
```

9.12. Модифицируйте программу, разработанную в результате решения задачи 8.12 (см. с. 683), таким образом, чтобы в список включались только кустарники, имеющие все необходимые характеристики. Например, если пользователь указывает, что растение должно обладать устойчивостью к холоду и засухе, то в список следует включить только войлочную восковницу и обыкновенный можжевельник. Если всем требованиям не удовлетворяет ни один из кустарников, то для указания на это должно быть выведено сообщение.

9.13. Предположим, что имеется множество генераторов, вырабатывающих электроэнергию, и множество устройств, потребляющих электроэнергию. Напишите программу, позволяющую подключать устройства к генераторам таким образом, чтобы было сведено к минимуму количество используемых генераторов и количество электроэнергии, не потребляемой от каждого используемого генератора. Например, если есть четыре генератора, вырабатывающие 5, 6, 7 и 10 ватт электроэнергии, и четыре устройства, потребляющие 4, 5, 6 и 7 ватт электроэнергии, то подключение устройства на 5 ватт к генератору на 5 ватт, устройства на 7 ватт к генератору на 7 ватт и устройств на 4 ватта и 6 ватт к генератору на 10 ватт позволяет минимизировать и количество используемых генераторов, и количество незатребованной электроэнергии. Для ввода и вывода данных в программе могут служить последовательности фактов. Проверьте разработанную вами программу на приведенном выше примере, а также проведите проверку для такого случая, когда генераторы остаются теми же самыми, как и в указанном примере, но подключаемые устройства потребляют 1, 3, 4, 5 и 9 ватт электроэнергии.

9.14. Напишите программу, которая функционирует как операционная система компьютера и определяет подходящие адреса в памяти для загрузки приложений, выделяя для них постоянные объемы памяти. В качестве входных данных для программы должны использоваться последовательности фактов, подобные следующим:

```
(launch (application word-processor)
(memory-needed 2))
(launch (application spreadsheet)
(memory-needed 6))
(launch (application game) (memory-needed 1))
(terminate (application word-processor))
(launch (application game) (memory-needed 1))
(terminate (application spreadsheet))
(terminate (application game))
```

Предположим, что компьютер имеет память объемом восемь мегабайтов и что количество мегабайтов, требуемое для любого приложения, измеряется целым числом. Если в памяти имеется участок, точно соответствующий требованиям приложения к объему памяти, этот участок и должен использоваться. Например, если имеются два участка свободной памяти, причем один из них имеет объем шесть мегабайтов, а второй — четыре мегабайта, и произошел запуск приложения, для которого требуется четыре мегабайта, то приложение должно быть загружено в свободный участок с объемом четыре мегабайта, а не в свободный участок с объемом шесть мегабайтов. После обработки команды `launch` (запустить приложение) или `terminate` (за-

вершить работу приложения) на внешнее устройство должно быть выведено сообщение. Если нет достаточно большого участка памяти для запуска приложения, должно быть выведено соответствующее сообщение. Проверьте разработанную вами программу, добавив к базе знаний приведенные выше факты и введя команду `run` после внесения в список фактов всех фактов. Вывод программы должен выглядеть примерно так:

```
Application word-processor memory location is 1 to 2.
Application spreadsheet memory location is 3 to 8.
Unable to launch application game.
Terminating word-processor.
Application game memory location is 1 to 1.
Terminating spreadsheet.
Terminating game.
```

9.15. Разработайте интерфейс текстового меню, который является автономным в пределах модуля и подходит для повторного использования в других программах. Пункты меню должны быть представлены как факты. Для пунктов меню должны поддерживаться два типа действий. Одно из этих действий должно обеспечивать прекращение выполнения программы. После выбора пункта меню этого типа программа должна прекратить выполнение. Пункт меню другого типа должен обеспечивать внесение факта в список фактов и перевод фокуса на конкретный модуль. Для этого действия факты `menu-item` должны содержать слоты, которые указывают модуль, переходящий в фокус, и значение факта, которое вносится в список фактов после выбора пункта меню. Например, в приведенном ниже диалоге результатом выбора пункта меню "`Option A`" должно быть следующее: перевод фокуса на модуль `A` и внесение в список фактов указанного факта `(menu-select (value option-a))`, где `A` и `option-a` — значения, указанные в факте `menu-item`. Модуль `A` содержит правило, которое согласуется с фактом `menu-select`, что приводит к выводу на внешнее устройство сообщения "`Executing Option A`".

```
CLIPS> (run)
Select one of the following options:
   1 - Option A
   2 - Option B
   9 - Quit Program
Your choice: 1
Executing Option A
Select one of the following options:
   1 - Option A
   2 - Option B
```

```
   9 - Quit Program
Your choice: 9
CLIPS>
```

9.16. Модифицируйте программу, разработанную в результате решения задачи 8.33 (см. с. 691), таким образом, чтобы для вывода информации обо всех звездах, имеющих указанный спектральный класс, всех звездах, имеющих указанную величину, и всех звездах, соответствующих и указанному спектральному классу, и указанный величине, наряду с расстоянием этих звезд от Земли в световых годах использовались отдельные модули.

9.17. Модифицируйте программу, разработанную в результате решения задачи 8.15 (см. с. 685), чтобы включить в нее правила, касающиеся остальных драгоценных камней, перечисленных в табл. 7.2 с описанием драгоценных камней в задаче 7.13 (см. с. 620). Для драгоценных камней, имеющих единственное числовое значение с указанием их твердости или плотности, измените правила так, чтобы было приемлемо любое значение, находящееся в пределах 0,01 от указанного значения. Включите правила, позволяющие проверить введенные данные и повторно передать запрос пользователю, если твердость не находится в пределах 1–10 включительно, а плотность — в пределах 1–6 включительно.

9.18. В табл. 9.3 перечислены доступные предметы и имена преподавателей, а также указано время проведения занятий, в течение которых ведется преподавание каждого предмета в средней школе с факультативным посещением. Ученик сам выбирает предметы, преподавателей и время проведения занятий. Напишите программу, которая предлагает наиболее подходящий вариант выбора преподавателя и времени проведения занятий для изучения выбранного учеником предмета. Входными данными для программы является факт, в котором указан предмет, планируемый для изучения, а также обозначены приоритеты, позволяющие выяснить, какие преподаватели и какое время проведения занятий являются более или менее предпочтительными. Для определения “наилучшего” преподавателя и времени проведения занятий оцените каждый возможный вариант следующим образом: начальная оценка — нуль; выбор более предпочтительного преподавателя или времени проведения занятий приводит к добавлению к оценке одного пункта; выбор менее предпочтительного преподавателя или времени проведения занятий приводит к вычитанию из оценки одного пункта; выбор преподавателя или времени проведения занятий, в отношении которых не определены предпочтения, приводит к тому, что оценка остается неизменной; “наилучшими” являются преподаватель и время проведения занятий с самой высокой оценкой.

Таблица 9.3. Предметы, имена преподавателей и время проведения занятий

Предмет	Преподаватель	Время проведения занятий
Алгебра	Джоунз	1, 2, 3
Алгебра	Смит	3, 4, 5, 6
История Америки	Вейл	5
История Америки	Хилл	1, 2
Искусство	Дженкинс	1, 3, 5
Биология	Долби	1, 2, 5
Химия	Долби	3, 6
Химия	Винсон	6
Французский язык	Блэйк	2, 4
Геология	Винсон	1
Геометрия	Джоунз	5, 6
Геометрия	Смит	1
Немецкий язык	Блэйк	5
Литература	Хеннинг	2, 3, 4, 5, 6
Литература	Дэвис	1, 2, 3, 4, 5
Музыка	Дженкинс	2, 4
Физкультура	Мак	1, 2, 3, 4, 5
Физкультура	Кинг	1, 2,3, 4, 6
Физкультура	Симпсон	2, 3, 4, 5, 6
Физика	Винсон	2, 3, 5
Испанский язык	Блэйк	1, 3
История Техаса	Вейл	2, 3, 4
История Техаса	Хилл	5, 6
Мировая история	Вейл	2, 3, 4
Мировая история	Хилл	4

9.19. Модифицируйте программу, разработанную в результате выполнения задачи 8.14 (см. с. 684), таким образом, чтобы был предусмотрен вывод на внешнее устройство информации об общей стоимости конфигурации. Если количество выбранных приспособлений превышает количество доступных отсеков или если количество электроэнергии, требуемое для приспособлений, превышает количество электроэнергии, предоставляемое блоком, то должно быть выведено предупреждающее сообщение.

9.20. Перечислите спецификации узла шаблона, сформированные для слотов с перечисленными ниже шаблонами.

а) `(blip (altitude 100))`.

б) `(blip (altitude ?x&:(> ?x 100)))`.

в) `(stop-light (color ~red))`.

г) `(balloon (color blue|white))`.

9.21. Нарисуйте сеть шаблонов для следующей группы шаблонов:

```
(data (x red) (y ?y) (z ?y))
(data (x ~red))
(item (b ?y) (c ?x&:(> ?x ?y)))
(item (a red) (b blue|yellow))
```

9.22. Нарисуйте сеть шаблонов и сеть соединений для приведенных ниже правил. Составьте список выражений, вычисляемых в каждом узле.

```
(defrule rule1
   (phase (name testing))
   (data (x ?x) (y ?y))
   (data (x ?y) (y ?x&:(> ?x ?y)))
   =>)
(defrule rule2
   (phase (name testing))
   (data (x ?x) (y ?y))
   (data (x ?y&~red) (y ?x&~green))
   =>)
```

9.23. Перезапишите следующее правило в целях повышения его эффективности:

```
(defrule bad-rule
   (items (x ?x1) (y ?y1) (z ?z1))
   (items (x ?x2) (y ?y2) (z ?z2))
   (items (x ?x3) (y ?y3) (z ?z3))
   (test (and (or (eq ?x3 green)
                  (eq ?x3 red))
                  (eq ?z2 ?y3)
   (> ?y1 ?x1)
   (< ?z1 ?x1)
   (neq ?z3 ?x1)))
   =>)
```

9.24. На рис. 9.15 показана сеть диагностирования неисправностей для гипотетических компонентов оборудования. Стрелки указывают направление, в котором распространяются неисправности. Например, при возникновении неисправности в компоненте A неисправности возникают и в компонентах B и C.

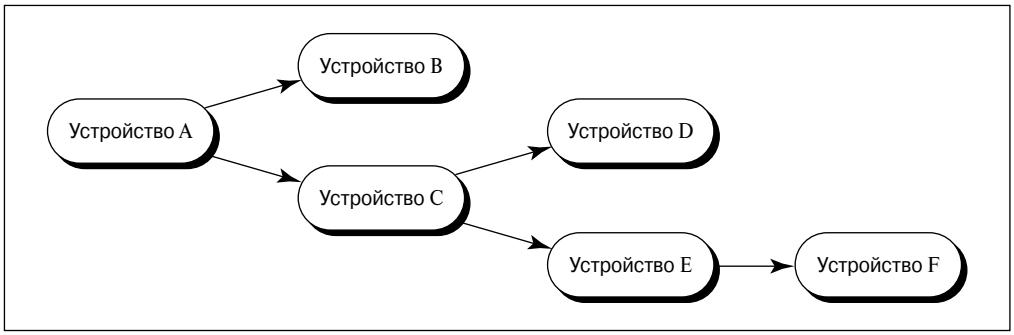


Рис. 9.15. Сеть диагностирования неисправностей гипотетических компонентов оборудования

Ниже приведены правила, которые описывают распространение неисправностей по сети.

```
(defrule propagate-device-A-fault
   (fault device-A)
=>
   (assert (fault device-B))
   (assert (fault device-C)))
   (defrule propagate-device-C-fault
      (fault device-C)
=>
   (assert (fault device-D))
   (assert (fault device-E)))
   (defrule propagate-device-E-fault
      (fault device-E)
=>
   (assert (fault device-F)))
```

Перепишите эти три правила в виде одного общего правила и конструкции `deffacts`, которые будут описывать распространение неисправностей так же, как и три приведенных выше правила. Объясните, какие изменения потребуется внести при использовании каждого из этих двух вариантов организации программы после добавления нового компонента к сети диагностирования неисправностей.

Глава 10

Процедурное программирование

10.1 Введение

В языке CLIPS предусмотрены средства процедурного программирования, аналогичные тем, которые применяются в таких языках, как C, Ada и Pascal. В настоящей главе приведено вводное описание этих средств. Иногда бывает более удобно (и более эффективно) осуществлять некоторые операции с использованием подхода, базирующегося на процедурном программировании, а не подхода, основанного на правилах. В настоящей главе вначале рассматривается ряд процедурных функций, включая определенные функции, которые обеспечивают организацию циклов и выбор путей передачи управления по условию. Для определения новых функций, которые могут вызываться из правил или других функций, предназначена конструкция `deffunction`. Конструкция `defglobal` позволяет определять глобальные переменные. В отличие от локальных переменных, определяемых в правилах или функциях, глобальные переменные всегда сохраняют свои значения. Конструкции `defgeneric` и `defmethod` обеспечивают создание универсальных функций, которые подобны обычным функциям, за исключением того, что выполняемый в них код зависит от количества и типов параметров, передаваемых в функцию. Наконец, в данной главе дано вводное описание целого ряда полезных вспомогательных функций.

10.2 Процедурные функции

В языке CLIPS предусмотрено несколько функций, предназначенных для управления потоком выполнения действий. Функции **while**, **if**, **switch**, **loop-for-count**, **progn$** и **break** предоставляют функциональные возможности, аналогичные управляющим структурам, которые предусмотрены в таких современных язы-

ках высокого уровня, как Ada, Pascal и C. Кроме того, функция **halt**, заданная в правой части любого правила, позволяет остановить выполнение правил.

Язык CLIPS в первую очередь предназначен для использования в качестве эффективного языка, основанного на правилах, а процедурные функции, рассматриваемые в этой главе, предусмотрены лишь для ограниченного применения. Попытка написать объемистую процедурную программу, включив ее в правую часть правила, противоречит общему назначению языка, основанного на правилах. Поэтому, вообще говоря, такие функции должны использоваться для выполнения простых проверок и циклов в правой части правила, и следует избегать применения в правых частях правил сложных вложенных конструкций, состоящих из процедурных функций.

Функция `if`

Функция `if` имеет следующий синтаксис:

```
(if <predicate-expression>
   then <expression>+
   [else <expression>+])
```

В этом определении `<predicate-expression>` — это единственное выражение (такое как предикативная функция или переменная), а параметр `<expression>+`, который следует за ключевыми словами `if` и `then`, представляет собой одно или несколько выражений, которые должны быть вычислены с учетом значения, полученного в результате вычисления выражения `<predicate-expression>`. Обратите внимание на то, что выражение `else` является необязательным.

При выполнении функции `if` вначале проверяется условие, представленное выражением `<predicate-expression>`, для определения того, должны ли быть выполнены действия, заданные в конструкции `then` или `else`. Если проверка условия приводит к получению любого символа, отличного от `FALSE`, то выполняются действия, заданные в конструкции `then` этой функции, а если проверка условия приводит к получению символа `FALSE`, то выполняются действия, заданные в конструкции `else`. Если же конструкция `else` не включена, то после получения ложных результатов проверки условия никакие действия не выполняются. Сразу после завершения выполнения функции `if` система CLIPS переходит к выполнению следующего действия в правой части правила, если оно имеется.

Функцию `if` удобно использовать для проверки значений в правой части правила, поскольку это позволяет избавиться от необходимости осуществлять проверку с помощью других правил. Например, с помощью следующего правила можно определить, должно ли быть продолжено выполнение программы:

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? ")
   (bind ?answer (read))
   (if (or (eq ?answer y) (eq ?answer yes))
   then (assert (phase continue))
   else (assert (phase halt))))
```

Следует отметить, что функция `if` позволяет преобразовать положительный или отрицательный ответ, `yes` или `no`, в факт, указывающий, какого типа действие должно быть предпринято. В рассматриваемом случае таким действием становится либо `continue`, либо `halt`.

Возвращаемым значением функции `if` является результат вычисления последнего выражения в части `then` или `else` функции. Если результатом вычисления выражения `<predicate-expression>` становится `FALSE` и в функции отсутствует часть `then`, то функция возвращает символ `FALSE`.

Функция `while`

Функция `while` имеет следующий синтаксис:

```
(while <predicate-expression> [do]
   <expression>*)
```

В этом определении `<predicate-expression>` представляет собой единственное выражение (такое как предикативная функция или переменная), а параметр `<expression>*`, который следует за необязательным ключевым словом `do`, обозначает от нуля или больше выражений, которые должны быть вычислены с учетом значения, возвращаемого в результате вычисления выражения `<predicate-expression>`. Эти выражения составляют **тело цикла**.

Та часть функции `while`, которая представлена выражением `<predicate-expression>`, вычисляется до выполнения действий, предусмотренных в теле цикла. Если вычисление выражения `<predicate-expression>` приводит к получению любого значения, отличного от символа `FALSE`, то выполняются выражения, представленные в теле цикла. Если же вычисление выражения `<predicate-expression>` приводит к получению символа `FALSE`, то программа переходит к выполнению оператора, который следует за функцией `while`, если таковой имеется. Условие функции `while` проверяется каждый раз перед выполнением операторов в теле цикла для определения того, должны ли они быть снова выполнены.

Функция while может использоваться вместе с функцией if для проверки ошибок ввода в правой части правила. Ниже приведена еще одна модификация правила continue-check, в которой применяется функция while для определения условия продолжения цикла до тех пор, пока не будет получен приемлемый ответ.

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? ")
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no))
      do
      (printout t "Continue? ")
      (bind ?answer (read)))
   (if (eq ?answer yes)
      then (assert (phase continue))
      else (assert (phase halt))))
```

Функция switch

Функция switch имеет следующий синтаксис:

```
(switch <test-expression>
   <case-statement>*
   [<default-statement>])
```

В этом определении оператор <case-statement> имеет следующую форму:

```
(case <comparison-expression> then <expression>*)
```

а оператор <default-statement> определен таким образом:

```
(default <expression>*)
```

Часть <expression>*, которая следует за ключевыми словами then и default, состоит из одного или нескольких выражений, которые должны быть вычислены с учетом возвращаемого значения выражения <comparison-expression>, сравниваемого с возвращаемым значением выражения <test-expression>. Следует учитывать, что необязательный вариант case, обозначаемый как default, должен быть расположен после всех других конструкций case.

После передачи управления функции switch вначале вычисляется часть, представленная выражением <test-expression>. После этого вычисляется каждое из выражений <comparison-expression> в том порядке, в котором они заданы в определении функции, и если результат <test-expression> сов-

падает с результатом вычисления выражения <comparison-expression>, то выполняются действия, заданные после ключевого слова then, и работа функции switch завершается. Если не найдено ни одно выражение <comparison-expression>, значение которого совпадало бы со значением <test-expression>, и задан вариант <default-statement>, то выполняются действия, определяемые вариантом <default-statement>.

В приведенном ниже коде функция switch используется для установления соответствия между символическими именами и знаками арифметических функций.

```
(defrule perform-operation
   (operation ?type ?arg1 ?arg2)
   =>
   (switch ?type
      (case times then
         (printout t ?arg1 " times " ?arg2
                           " is " (* ?arg1 ?arg2)
                                  crlf))
   (case plus then
      (printout t ?arg1 " plus " ?arg2
                        " is " (+ ?arg1 ?arg2)
                                  crlf))
   (case minus then
      (printout t ?arg1 " minus " ?arg2
                        " is " (- ?arg1 ?arg2)
                                  crlf))
   (case divided-by then
      (printout t ?arg1 " divided by " ?arg2
                        " is " (/ ?arg1 ?arg2)
                                  crlf))))
```

В качестве примера можно привести следующий диалог:

```
CLIPS> (assert (operation plus 3 4))↵
<Fact-1>
CLIPS> (run)↵
3 plus 4 is 7
CLIPS>
```

Символ plus в факте operation вызывает переход к варианту case со значением plus в операторе switch, после чего выводится результат сложения чисел 3 и 4.

Приведенное выше правило perform-operation можно записать иначе, в виде четырех отдельных правил, т.е. без использования функции switch, следующим образом:

```
(defrule perform-operation-times
   (operation times ?arg1 ?arg2)
   =>
   (printout t ?arg1 " times " ?arg2
                    " is " (* ?arg1 ?arg2) crlf))
(defrule perform-operation-plus
   (operation plus ?arg1 ?arg2)
   =>
   (printout t ?arg1 " plus " ?arg2
                    " is " (+ ?arg1 ?arg2) crlf))
(defrule perform-operation-minus
   (operation minus ?arg1 ?arg2)
   =>
   (printout t ?arg1 " minus " ?arg2
                    " is " (- ?arg1 ?arg2) crlf))
(defrule perform-operation-divided-by
   (operation divided-by ?arg1 ?arg2)
   =>
   (printout t ?arg1 " divided by " ?arg2
                    " is " (/ ?arg1 ?arg2) crlf))
```

Функция loop-for-count

Функция loop-for-count имеет следующий синтаксис:

```
(loop-for-count <range-spec> [do] <expression>*)
```

В этом определении конструкция <range-spec> имеет такую форму:

```
<end-index> |
(<loop-variable> <end-index>) |
(<loop-variable> <start-index> <end-index>)
```

В данном случае <end-index> и <start-index> представляют собой выражения, которые возвращают целые числа. Если задано только выражение <end-index>, то оно рассматривается как количество итераций выполнения тела функции, определяемого параметром <expression>*. Если заданы выражения <loop-variable> и <end-index>, то осуществляется указанное количество итераций выполнения тела функции, а текущий номер итерации, возрастающий в пределах от 1 до <end-index>, сохраняется в переменной <loop-variable> на каждой итерации. Если кроме этого задано выражение

<start-index>, то отсчет количества итераций начинается со значения <start-index>, а не с 1, и количество выполняемых итераций определяется как разность между значениями <end-index> и <start-index> плюс один. Если значение <start-index> больше чем <end-index>, то никакие итерации не выполняются. Ниже приведен пример, в котором используются все три составляющие конструкции <range-spec> и вложенные вызовы функции loop-for-count.

```
CLIPS>
(loop-for-count (?cnt1 2 4) do
   (loop-for-count (?cnt2 3) do
      (printout t ?cnt1 " ")
      (loop-for-count 3 do
         (printout t "."))
         (printout t " " ?cnt2 crlf)))↵
2 ... 1
2 ... 2
2 ... 3
3 ... 1
3 ... 2
3 ... 3
4 ... 1
4 ... 2
4 ... 3
FALSE
CLIPS>
```

Переменная <loop-variable> маскирует любые переменные, объявленные вне выражения loop-for-count и имеющие такое же имя, например, как показано ниже.

```
CLIPS>
(defrule masking-example
   =>
   (bind ?x 4)
   (loop-for-count (?x 2) do
      (printout t "inside ?x is " ?x crlf))
      (printout t "outside ?x is " ?x crlf))↵
CLIPS> (reset)↵
CLIPS> (run)↵
inside ?x is 1
inside ?x is 2
outside ?x
```

```
is 4
CLIPS>
```

Функция progn$

Функция progn$ имеет следующий синтаксис:

```
(progn$ <list-spec> <expression>*)
```

В этом определении параметр <list-spec> задается в такой форме:

```
<multifield-expression> |
(<list-variable> <multifield-expression>)
```

Если задан не только параметр <list-spec>, но и параметр <multifield-expression>, то тело функции, представленное как <expression>*, выполняется по одному разу для каждого поля в результирующем многозначном значении, которое получено в результате вычисления выражения <multifield-expression>. Применение параметра <list-variable> вместе с параметром <multifield-expression> позволяет осуществлять выборку поля текущей итерации, ссылаясь на переменную. Кроме того, создается специальная переменная путем добавления суффикса -index к имени параметра <list-variable>. Эта переменная содержит индекс текущей итерации. Возвращаемым значением данной функции является возвращаемое значение последнего выражения <expression>, вычисленного для последнего поля параметра <multifield-expression>. Вызовы функции loop-for-count, как и вызовы функции progn$, могут вкладываться, а переменные, созданные для выражения progn$, маскируют переменные, объявленные вне выражения progn$, имеющие то же имя. Ниже приведены примеры использования обоих вариантов представления параметра <list-spec>.

```
CLIPS>
(progn$ (create$ 1 2 3)
   (printout t . crlf))↵
.
.
.
CLIPS>
(progn$ (?v (create$ a b c))
   (printout t ?v-index " " ?v crlf))↵
1 a
2 b
3 c
CLIPS>
```

Функция break

Функция break имеет следующий синтаксис:

```
(break)
```

Функция break прекращает выполнение той функции while, loop-for-count или progn$, в которой она непосредственно обнаруживается. Обычно функция break используется, чтобы вызвать преждевременное завершение цикла при обнаружении некоторого условия. Например, правило print-list, показанное в приведенном ниже диалоге, выводит первые пять элементов из списка, а затем выводит многоточие, ..., если есть еще оставшиеся элементы.

```
CLIPS>
(defrule print-list
   (print-list $?list)
   =>
   (progn$ (?v ?list)
      (if (<= ?v-index 5)
         then
         (printout t ?v " ")
         else
         (printout t "...")
         (break)))
   (printout t crlf))↵
CLIPS> (reset)↵
CLIPS> (assert (print-list a b c d e))↵
<Fact-1>
CLIPS> (run)↵
a b c d e
CLIPS> (assert (print-list a b c d e f g h))↵
<Fact-2>
CLIPS> (run)↵
a b c d e ...
CLIPS>
```

Функция halt

Функция halt может использоваться в правой части любого правила для останова выполнения правил, находящихся в рабочем списке правил. Эта функция не требует параметров. После ее вызова прекращается выполнение каких-либо действий, заданных в правой части запущенного правила, и управление передается в приглашение верхнего уровня. В рабочем списке правил продолжают находиться все оставшиеся правила, активизированные ко времени вызова функции halt.

Например, в правиле `continue-check` следующее действие:

```
(assert (phase halt))
```

можно было бы заменить таким действием, которое прекращает выполнение правил:

```
(halt)
```

Функция `halt` становится особенно удобной, если требуется прекратить выполнение программы после того, как пользователь выразит намерение снова вызвать в дальнейшем программу на выполнение с помощью команды `run`. Рассмотрим следующую модификацию правила `continue-check`:

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? ")
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no))
      do
      (printout t "Continue? ")
      (bind ?answer (read)))
   (assert (phase continue))
   (if (neq ?answer yes)
      then (halt)))
```

Обратите внимание на то, что в этом правиле в список фактов вносится факт `(phase continue)`, невзирая на то, что ответил пользователь на вопрос о продолжении, "`Continue? `". Внесение этого факта в список фактов приводит к тому, что в рабочий список правил помещаются правила, необходимые для продолжения выполнения. Если ответ на вопрос о продолжении не является положительным, то выполнение программы прекращается с помощью функции `halt`. После этого пользователь может проверить правила и факты, а затем снова перейти к выполнению программы с того места, где она была остановлена, воспользовавшись командой `run`.

10.3 Конструкция `deffunction`

Язык CLIPS позволяет определять новые функции по такому же принципу, как и в других процедурных языках. Такие функции, применяемые наряду с правилами, позволяют уменьшить количество повторяющихся выражений и в левых, и правых частях. Новые функции определяются с использованием **конструкции deffunction**. Конструкция `deffunction` имеет следующий общий формат:

```
(deffunction <deffunction-name> [<optional-comment>]
(<regular-parameter>* [<wildcard-parameter>])
<expression>*)
```

В этом определении параметр <regular-parameter> представляет собой однозначную переменную, а <wildcard-parameter> — многозначную переменную. Имя конструкции deffunction, <deffunction-name>, должно быть уникальным, в частности, не должно совпадать с одной из уже существующих, заранее определенных функций CLIPS. Тело конструкции deffunction, представленное параметром <expression>*, состоит из ряда выражений, подобных выражениям в правой части правила, которые выполняются последовательно при вызове конструкции deffunction. Определяемые пользователем конструкции deffunction действуют аналогично заранее определенным функциям, предусмотренным в языке CLIPS. В любых условиях, допускающих вызов заранее определенной системной функции CLIPS, можно вызвать конструкцию deffunction. Но, в отличие от заранее определенных функций, допускается удаление конструкций deffunction, а для отслеживания их выполнения может использоваться команда watch.

Объявления <regular-parameter> и <wildcard-parameter> позволяют задавать параметры, передаваемые в конструкцию deffunction при ее вызове. Такой способ вызова в определенной степени аналогичен активизации правил, если переменные, связанные со значениями в левой части правила, которые применяются в правой части правила, рассматриваются как объявления параметров. Кроме того, в теле конструкции deffunction для создания локальных переменных может использоваться команда bind, в полном соответствии с тем, как это действие выполняется в правой части правила.

Конструкция deffunction может возвращать значения, по аналогии с тем, как возвращают значения некоторые заранее определенные функции. Возвращаемым значением конструкции deffunction является значение последнего выражения, вычисленного в теле конструкции deffunction.

В качестве примера рассмотрим конструкцию deffunction, в которой вычисляется длина гипотенузы прямоугольного треугольника с помощью теоремы Пифагора. Допустим, что a и b — стороны, образующие прямой угол, а c — гипотенуза. В таком случае согласно данной теореме можно записать следующее:

$$a^2 + b^2 = c^2$$

В результате преобразования определим длину гипотенузы таким образом:

$$c = \sqrt{a^2 + b^2}$$

Преобразование этой формулы в конструкцию deffunction приводит к получению следующего определения:

```
(deffunction hypotenuse-length (?a ?b)
   (** (+ (* ?a ?a) (* ?b ?b)) 0.5))
```

Двумя параметрами этой функции являются `?a` и `?b`; эти параметры используются для передачи в функцию значений длины двух сторон треугольника, примыкающих к прямому углу. Функция `**` со вторым параметром 0.5 применяется для вычисления квадратного корня, поскольку значение выражения `(** <numeric-expression> <numeric-expression>)` представляет собой результат возведения первого параметра в степень, заданную вторым параметром. После того как функция `hypotenuse-length` будет определена, ее можно вызывать из приглашения к вводу команд, следующим образом:

```
CLIPS> (hypotenuse-length 3 4)↵
5.0
CLIPS>
```

В данном случае значения длины сторон, прилегающих к прямому углу, равны 3 и 4, поэтому длина гипотенузы вычисляется правильно, как равная 5.

Поскольку в конструкциях `deffunction` можно использовать локальные переменные, применяемый способ вычисления длины гипотенузы можно представить в формате, более удобном для чтения, как показано ниже.

```
(deffunction hypotenuse-length (?a ?b)
   (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
   (** ?temp 0.5))
```

Функция `return`

Функция `return` не только позволяет прекратить выполнение правой части правила, но и дает возможность прекратить работу выполняемой в данный момент конструкции `deffunction`. Функция `return`, применяемая вместе с конструкцией `deffunction`, имеет следующий синтаксис:

```
(return [<expression>])
```

Если в этом определении задано выражение `<expression>`, то в качестве возвращаемого значения конструкции `deffunction` используется результат вычисления этого выражения. В функции `hypotenuse-length` можно было бы явно применить функцию `return` для возврата вычисленной длины гипотенузы одним из следующих способов:

```
(deffunction hypotenuse-length (?a ?b)
   (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
   (return (** ?temp 0.5)))
```

или:

```
(deffunction hypotenuse-length (?a ?b)
```

```
   (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
   (bind ?c (** ?temp 0.5))
   (return ?c))
```

Но возвращаемым значением указанной конструкции deffunction является результат вычисления последнего выражения, а вычисление всех выражений в конструкции deffunction с именем hypotenuse-length осуществляется последовательно, поэтому нет необходимости применять явно заданный оператор return. Необходимость в использовании функции return возникает главным образом, если удовлетворяется условие, согласно которому должно быть завершено выполнение конструкции deffunction, или если возвращаемое значение вычисляется на основания выражения, не являющегося последним выражением, которое подлежит вычислению. Например, ниже приведена конструкция deffunction, которая определяет, является ли заданное число простым.

```
(deffunction primep (?num)
   (loop-for-count (?i 2 (- ?num 1))
   (if (= ?num (* (div ?num ?i) ?i))
      then
      (return FALSE)))
   (return TRUE))
```

Конструкция deffunction с именем primep возвращает символ TRUE, если параметр ?num — простое число; в противном случае она возвращает символ FALSE. Число является простым, только если его делителем служит число 1 или оно само. Таким образом, если число делится без остатка на любое другое число, то не может быть простым. В конструкции deffunction с именем primep используется функция loop-for-count для итерации по всем числам от 2 до того числа, которое на единицу меньше числа, проверяемого в качестве "кандидата на звание" простого числа, ?num (данная программа немного упрощена, так как было бы достаточно проверить возможные делители, не превышающие корня квадратного из числа ?num). Если окажется, что какое-либо из этих чисел делит без остатка число ?num, то функция завершает свою работу и возвращает символ FALSE, поскольку рассматриваемое число не является простым. Для определения статуса числа ?num как простого используется функция div. Функция div выполняет целочисленное деление и дает в результате целочисленное возвращаемое значение. Поэтому выражение (div 5 2) возвращает 2, а не значение 2.5, которое было бы возвращено при использовании выражения (/ 5 2). Если выражение (* (div ?num ?i) ?i) возвращает первоначальное значение ?num, то число ?num делится без остатка на ?i и поэтому число ?num не является простым. А в том случае, если ни одно из значений ?i, применяемых в качестве делителя в функции loop-for-count, не делит число?num без остатка, то

?num — простое число, и значение, возвращаемое конструкцией deffunction, становится равным TRUE.

Еще один вариант программы Sticks

Как было указано в разделе 8.7, для определения того, должен ли первым ходить компьютер или человек, применяются конструкция deffacts и три правила:

```
(deffacts initial-phase
   (phase choose-player))
(defrule player-select
   (phase choose-player)
   =>
(printout t "Who moves first (Computer: c "
                  "Human: h)? ")
      (assert (player-select (read))))
(defrule good-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&c | h)
   =>
   (retract ?phase ?choice)
   (assert (player-move ?player)))
(defrule bad-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&~c&~h)
   =>
   (retract ?phase ?choice)
   (assert (phase choose-player))
   (printout t "Choose c or h." crlf))
```

Очевидно, что необходимость проверять введенные данные встречается часто, поэтому рассмотрим следующую функцию, которая позволила бы в данном случае исключить необходимость проводить такую проверку во всех трех правилах:

```
(deffunction check-input (?question ?values)   ;; Строка 1
   (printout t ?question " " ?values " ")      ;; Строка 2
   (bind ?answer (read))                       ;; Строка 3
   (while (not (member$ ?answer ?values))      ;; Строка 4
      (printout t ?question " " ?values " ")   ;; Строка 5
      (bind ?answer (read)))                   ;; Строка 6
   (return ?answer))                           ;; Строка 7
```

Строка 1 начинается с определения конструкции deffunction с именем check-input. Определяемая здесь функция принимает два параметра: ?question — приглашение с вопросом, отображаемое для пользователя, и ?values — список значений, которые допускается вводить в ответ на вопрос. В строке 2 выводятся приглашение с вопросом, а также список допустимых значений. Ответ пользователя на вопрос перехватывается в строке 3 с помощью функции read. В строке 4 начинается цикл while, итерации которого выполняются до тех пор, пока пользователь не введет допустимый ответ. Функция member$ возвращает символ TRUE, если ее первым параметром является один из элементов многозначного значения, представленного в виде второго параметра. В данном случае, если ответ, предоставленный пользователем, ?answer, не является элементом списка допустимых значений, ?values, то выполняется тело цикла while. Строки 5 и 6, формирующие тело цикла while, обеспечивают передачу пользователю повторяющихся вопросов. После того как пользователь предоставит допустимый ответ, цикл while завершается и выполняется строка 7, которая возвращает допустимый ответ, введенный пользователем.

Пример использования конструкции deffunction с именем check-input приведен ниже. Для создания многозначного значения из однозначных параметров, которое может быть передано в качестве второго параметра в указанную конструкцию deffunction, применяется функция create$.

```
CLIPS>
(check-input "Who moves first, Computer or Human?"
                     (create$ c h))↵
Who moves first, Computer or Human? (c h) x↵
Who moves first, Computer or Human? (c h) computer↵
Who moves first, Computer or Human? (c h) c↵
c
CLIPS>
```

С помощью функции check-input правило player-select можно перезаписать следующим образом и тем самым исключить необходимость использования правил good-player-choice и bad-player-choice:

```
(defrule player-select
   ?f <- (phase choose-player)
   =>
   (retract ?f)
   (bind ?player
      (check-input
      "Who moves first, Computer or Human?"
      (create$ c h)))
   (assert (player-move ?player)))
```

Рекурсия

Конструкции `deffunction` могут вызывать в своем теле другие конструкции `deffunction`, включая самих себя. В качестве примера укажем, что значение факториала от положительного целого числа n определяется следующим образом:

$$\text{factorial}(n) = \begin{cases} n * \text{factorial}(n-1) & \text{если } n > 1 \\ 1 & \text{если } n = 0 \end{cases}$$

Таким образом, чтобы вычислить факториал от n, необходимо вычислить факториал от $n-1$. Поэтому факториал числа 3 (обозначаемый как 3!) равен $3*2!$, что, в свою очередь, равно $3*2*1!$ и составляет $3*2*1$ или 6. Ниже приведена конструкция `deffunction`, позволяющая вычислить факториал любого целого числа.

```
(deffunction factorial (?n)
   (if (>= ?n 1)
      then (* ?n (factorial (- ?n 1)))
      else 1))
```

Как указано выше, значение (factorial 3) можно вычислить с помощью выражения $3*$(factorial 2), затем $3*2*$(factorial 1) и, наконец, $3*2*1$, что составляет 6. В том, что эти вычисления действительно проводятся правильно, можно убедиться, непосредственно вызывая конструкцию `deffunction` с именем `factorial`, как показано ниже.

```
CLIPS> (factorial 3)↵
6
CLIPS> (factorial 2)↵
2
CLIPS> (factorial 1)↵
1
CLIPS>
```

Предварительные объявления

Иногда в конструкциях `deffunction` применяются циклические ссылки друг на друга. Например, в конструкции `deffunction A` может быть предусмотрен вызов конструкции `deffunction B`, которая вызывает конструкцию `deffunction C`, а последняя содержит вызов конструкции `deffunction A`, как в следующем примере:

```
(deffunction A (?n)
   (if (<= ?n 0)
      then 1
```

```
      else (+ 2 (B (- ?n 1)))))
(deffunction B (?n)
   (if (<= ?n 0)
      then 1
      else (* 2 (C (- ?n 1)))))
(deffunction C (?n)
   (if (<= ?n 0)
      then 1
      else (- 2 (A (- ?n 1)))))
```

Ниже приведены значения, вычисляемые этими функциями, в более удобном для чтения виде.

$$A(n) = \begin{cases} 1 & \text{если } n \leq 0 \\ 2 + B(n-1) & \text{в ином случае} \end{cases}$$

$$B(n) = \begin{cases} 1 & \text{если } n \leq 0 \\ 2 * C(n-1) & \text{в ином случае} \end{cases}$$

$$C(n) = \begin{cases} 1 & \text{если } n \leq 0 \\ 2 - A(n-1) & \text{в ином случае} \end{cases}$$

Все упоминаемые конструкции deffunction должны быть представлены в программе к тому времени, как в процессе синтаксического анализа встречается ссылка на эти конструкции, поэтому возникает проблема. Дело в том, что невозможно откомпилировать ни одну из этих функций, поскольку каждая из них зависит от другой, как показано ниже.

```
CLIPS>
(deffunction A (?n)
   (if (<= ?n 0)
      then 1
      else (+ 2 (B (- ?n 1)))))↵
[EXPRNPSR3] Missing function declaration for B.
ERROR:
(deffunction MAIN::A
   (?n)
   (if (<= ?n 0)
      then
      1
      else
      (+ 2 (B
CLIPS>
```

Один из способов устранения указанной проблемы состоит в том, чтобы подготовить предварительные объявления для некоторых из функций. В предварительном объявлении дается описание, состоящее из имени и списка параметров функции, но тело функции остается пустым. Функция объявлена, и это означает, что на нее можно сделать ссылку, но функция не имеет тела, поэтому не может ссылаться на другие функции. Предварительное объявление может быть в дальнейшем заменено такой версией, которая включает тело, например, как показано ниже.

```
CLIPS> (deffunction B (?n))↵
CLIPS>
(deffunction A (?n)
   (if (<= ?n 0)
      then 1
      else (+ 2 (B (- ?n 1)))))↵
CLIPS> (deffunction C (?n))↵
CLIPS>
(deffunction B (?n)
   (if (<= ?n 0)
      then 1
      else (* 2 (C (- ?n 1)))))↵
CLIPS>
(deffunction C (?n)
   (if (<= ?n 0)
      then 1
      else (- 2 (A (- ?n 1)))))↵
CLIPS>
```

В данном примере предварительные объявления были введены непосредственно перед конструкцией `deffunction`, требующей объявления. Обычно, если применяемые в программе правила загружаются из текстового файла, все предварительные объявления сосредоточиваются в начале файла, чтобы проще было изучать и сопровождать код, но единственным обязательным требованием является то, чтобы такие объявления были сделаны до того, как будет определена конструкция `deffunction`, в которой они требуются.

Отслеживание работы конструкций `deffunction`

Если для отслеживания работы конструкций `deffunction` используется команда `watch`, то каждый раз, когда начинается или заканчивается выполнение конструкции `deffunction`, выводится информационное сообщение, например, как показано ниже.

```
CLIPS> (watch deffunctions)↵
CLIPS>
(factorial 2)↵
DFN >> factorial ED:1 (2)
DFN >> factorial ED:2 (1)
DFN >> factorial ED:3 (0)
DFN << factorial ED:3 (0)
DFN << factorial ED:2 (1)
DFN << factorial ED:1 (2)
2
CLIPS>
```

Обозначение DFN, приведенное в начале информационного сообщения, указывает, что оно относится к конструкции deffunction. Символ >> показывает, что происходит переход в конструкцию deffunction, а символ << говорит о том, что осуществляется выход из конструкции deffunction. Следующий символ представляет собой имя конструкции deffunction, в которую входит или из которой выходит программа; в данном случае упоминается конструкция deffunction с именем factorial. Символ ED является сокращением от "Evaluation Depth" (Глубина вложенности); за этим символом следует двоеточие и значение текущей глубины вложенности в виде целого числа. Значение глубины вложенности показывает, как вкладываются друг в друга вызовы конструкций deffunction. Отсчет этого значения начинается с нуля, а затем увеличивается на единицу после перехода в каждую очередную конструкцию deffunction. После выхода из очередной конструкции deffunction значение глубины вложенности уменьшается на единицу. Наконец, последним фрагментом информации, отображаемым в данной строке, являются фактические параметры, передаваемые в конструкцию deffunction.

В данном примере конструкция deffunction с именем factorial первоначально вызывается с параметром 2. Глубина вложенности для этого вызова равна 1. Конструкция deffunction с именем factorial должна быть вызвана снова для вычисления факториала числа 1. Этот вызов имеет глубину вложенности 2, а его параметр равен 1. Следующий вызов применяется для вычисления факториала числа 0. Данный вызов имеет глубину вложенности 3, и его параметром является 0. Но в конструкции deffunction с именем factorial не требуется рекурсия для определения факториала числа 0, поэтому нет необходимости снова вызывать конструкцию deffunction и можно начать выходить из вложенных вызовов конструкции deffunction. После выхода из каждой очередной конструкции deffunction глубина вложенности вызова уменьшается на 1 до тех пор, пока не происходит выход из первоначального вызова конструкции deffunction и не осуществляется возврат значения 2.

Обеспечивается также возможность отслеживать только определенные конструкции deffunction; для этого необходимо указать имена этих конструкций в конце команды watch deffunctions, например, как показано ниже.

```
CLIPS> (unwatch deffunctions)↵
CLIPS>
(watch deffunctions C)↵
CLIPS> (A 2)↵
DFN >> C ED:3 (0)
DFN << C ED:3 (0)
4
CLIPS> (watch deffunctions A B)↵
CLIPS> (A 2)↵
DFN >> A ED:1 (2)
DFN >> B ED:2 (1)
DFN >> C ED:3 (0)
DFN << C ED:3 (0)
DFN << B ED:2 (1)
DFN << A ED:1 (2)
4
CLIPS>
```

Обратите внимание на то, что глубина вложенности для вызова конструкций deffunction все еще вычисляется, даже несмотря на то, что не предусмотрено отслеживание работы конструкций deffunction.

Параметр с подстановочным символом

Если все параметры в списке параметров конструкции deffunction являются однозначными переменными, то имеет место взаимно однозначное соответствие между количеством этих параметров и количеством параметров, которые должны быть переданы в конструкцию deffunction во время ее вызова. Иными словами, если предусмотрено три формальных параметра, то во время вызова конструкции deffunction должны быть переданы три значения.

Если же последним параметром, объявленным в конструкции deffunction, является многозначная переменная (такой параметр называется *параметром с подстановочным символом*), то конструкция deffunction может быть вызвана с бóльшим количеством параметров, чем указано в списке параметров. Если в списке формальных параметров имеется M параметров, а в вызове конструкции deffunction передаются N параметров, то первые M–1 параметров в вызове конструкции deffunction отображаются на первые M–1 параметры в списке параметров. А параметры от M до N в вызове конструкции deffunction отображаются на M-й параметр в списке параметров в качестве многозначного значения.

В качестве примера еще раз рассмотрим конструкцию deffunction с именем check-input и преобразуем ее последний параметр, ?values, в параметр с подстановочным символом, как показано ниже.

```
(deffunction check-input (?question $?values)
   (printout t ?question " " ?values " ")
   (bind ?answer (read))
   (while (not (member$ ?answer ?values))
      (printout t ?question " " ?values " ")
      (bind ?answer (read)))
   (return ?answer))
```

После преобразования последнего параметра в параметр с подстановочным символом отпадает необходимость в использовании функции create$ для создания многозначного значения и передачи его в конструкцию deffunction, как показывает следующий пример:

```
CLIPS>
(check-input "Who moves first, Computer or Human?" c h)↵
Who moves first, Computer or Human? (c h) computer↵
Who moves first, Computer or Human? (c h) human↵
Who moves first, Computer or Human? (c h) h↵
h
CLIPS>
```

В данном случае после вызова конструкции check-input строка “Who moves first, Computer or Human?” связывается с параметром ?question, а оставшиеся параметры, c и h, преобразуются в многозначное значение и сохраняются в параметре с подстановочным символом, $?values.

Команды для работы с конструкцией deffunction

Команда ppdeffunction (сокращение от pretty print deffunction — структурированный вывод конструкции deffunction) используется для отображения текстового представления конструкции deffunction. **Команда undeffunction** применяется для удаления конструкции deffunction. **Команда list-deffunctions** служит для отображения списка конструкций deffunction, предусмотренных в языке CLIPS. **Функция get-deffunction-list** возвращает многозначное значение, содержащее список объявленных конструкций deffunction. Эти команды имеют следующий синтаксис:

```
(ppdeffunction <deffunction-name>)
(undeffunction <deffunction-name>)
(list-deffunctions [<module-name>])
(get-deffunction-list [<module-name>])
```

Ниже приведены примеры применения данных функций.

```
CLIPS> (list-deffunctions)↵
hypotenuse-length
primep
check-input
factorial
A
B
C
For a total of 7 deffunctions.
CLIPS> (get-deffunction-list)↵
(hypotenuse-length primep check-input factorial A B C)
CLIPS> (undeffunction primep)↵
CLIPS>
(get-deffunction-list)↵
(hypotenuse-length check-input factorial A B C)
CLIPS> (ppdeffunction
factorial)↵
(deffunction MAIN::factorial
   (?n)
   (if (>= ?n 1)
      then
      (* ?n (factorial (- ?n 1)))
      else
      1))
CLIPS>
```

Если на некоторую конструкцию `deffunction` ссылаются другие конструкции `deffunction` или конструкции каких-то других типов, то данная конструкция не может быть удалена. В такой ситуации единственным способом удалении конструкции `deffunction` является уничтожение всех других ссылок или выдача команда `clear`, как показано в следующем примере:

```
CLIPS> (undeffunction A)↵
[PRNTUTIL4] Unable to delete deffunction A.
CLIPS> (deffunction C (?n))↵
CLIPS> (undeffunction A)↵
CLIPS> (undeffunction C)↵
[PRNTUTIL4] Unable to delete deffunction C.
CLIPS> (clear)↵
CLIPS>
```

Функции, определяемые пользователем

В языке CLIPS предусмотрена возможность не только объявлять конструкции `deffunction`, но и применять способ вызова функций, написанных на языке программирования C. Функции, определяемые с использованием этой методологии, называются **функциями, определяемыми пользователем**. Данный термин был введен еще тогда, когда в языке CLIPS не была доступна конструкция `deffunction` и единственным способом введения новых функций служило написание этих функций на языке C. Поэтому указанный термин применяется только к функциям, написанным на языке C, даже несмотря на то, что конструкции `deffunction` объявляются пользователем и вполне могут рассматриваться как функции, определяемые пользователем.

Подробная информация о том, как создать и ввести в язык CLIPS функцию, определяемую пользователем, приведена в документе *Advanced Programming Guide*, который включен в компакт-диск, прилагаемый к этой книге. Как правило, заниматься написанием функций, определяемых пользователем, а не применять конструкцию `deffunction`, целесообразно только по двум причинам. Первой причиной может стать наличие кода, уже написанного на языке C, который желательно объединить с кодом на языке CLIPS. Во многих случаях повторное написание больших объемов кода на другом языке является нежелательным, поэтому проще оставить этот код написанным на языке C и предусмотреть возможность вызывать его из программы CLIPS с помощью определяемой пользователем функции. Второй причиной может стать быстродействие. Выполнение функции C, откомпилированной в собственный код компьютера, происходит гораздо быстрее по сравнению с конструкцией `deffunction`, эксплуатируемой в интерпретирующей среде, которая предоставляется системой CLIPS, особенно если речь о больших фрагментах кода. Но, к счастью, в большинстве задач удобства, создаваемые благодаря тому, что есть возможность определять конструкции `deffunction` непосредственно в среде CLIPS, компенсируют снижение быстродействия, вызванное эксплуатацией интерпретируемого кода. Для добавления определяемых пользователем функций к системе CLIPS требуется создание новой исполняемой программы CLIPS, а для тех, кто не знаком с использованием компиляторов или с программированием на языке C, такая задача требует настолько значительных усилий, что не каждый на это решится. Тем не менее в документе *Advanced Programming Guide* приведено много примеров определяемых пользователем функций; кроме того, многочисленные примеры определяемых пользователем функций можно найти в исходном коде CLIPS, который также включен в указанный компакт-диск. Все функции и команды, описанные в главах 7–12, были введены в среду CLIPS с использованием методологии создания определяемых пользователем функций.

10.4 Конструкция defglobal

Язык CLIPS позволяет определять переменные, сохраняющие свои значения вне области определения той или иной конструкции. Эти переменные называются **глобальными переменными**. До сих пор все переменные, применявшиеся в конструкциях, представляли собой **локальные переменные**. Эти переменные являются локальными применительно к конструкциям, в которых они упоминаются. Например, рассмотрим переменную ?x, используемую в следующих двух правилах:

```
(defrule example-1
   (data-1 ?x)
   =>
   (printout t "?x = " ?x crlf))
(defrule example-2
   (data-2 ?x)
   =>
   (printout t "?x = " ?x crlf))
```

Значение переменной ?x в правиле example-1 не ограничивает каким-либо образом значение переменной ?x в правиле example-2. Если в список фактов будет внесен факт (data-1 3), то переменной ?x в правиле example-1 будет присвоено значение 3. Но это не означает, что тем самым на шаблон правила example-2 будет наложено такое ограничение, что он должен согласовываться только с таким фактом data-2, в котором значение переменной ?x равно 3. Более того, внесение в список фактов еще одного факта, например (data-1 4), не приведет к появлению такого ограничения значения ?x, что оно должно совпадать со значениями двух активизаций правила example-1. По существу, каждое частичное соответствие или каждая активизация правила имеет свое собственное множество локальных переменных. То же относится к каждому вызову конструкции deffunction.

Глобальные переменные определяются с помощью **конструкции defglobal**. Конструкция defglobal имеет следующий общий формат:

```
(defglobal [<defmodule-name>] <global-assignment>*)
```

В этом определении параметр <global-assignment> задается следующим образом:

```
<global-variable> = <expression>
```

а параметр <global-variable> определяется так:

```
?*<symbol>*
```

Терм <defmodule-name> представляет собой модуль, в котором должны быть определены глобальные переменные. Если этот параметр не задан, то гло-

бальные переменные помещаются в текущий модуль. Имена глобальных переменных начинаются и заканчиваются знаком *, поэтому можно легко определить, что ?x является локальной переменной, а ?*x* представляет собой глобальную переменную. В определении конструкции defglobal начальное значение для каждой переменной defglobal задается путем вычисления выражения <expression> и присваивания полученного значения переменной defglobal, например, как показано ниже.

```
CLIPS>
(defglobal ?*x* = 3
           ?*y* = (+ ?*x* 1))↵
CLIPS> ?*x*↵
3
CLIPS> ?*y*↵
4
CLIPS>
```

Обратите внимание на то, что мы смогли использовать значение глобальной переменной ?*x* для вычисления начального значения глобальной переменной ?*y*, поскольку переменная ?*x* определена прежде, чем ?*y*. Кроме того, глобальные переменные сохраняют свои значения за пределами всех конструкций, поэтому достаточно просто ввести имя глобальной переменной в приглашении к вводу команды, чтобы определить ее значение. Ссылки на глобальные переменные defglobal могут использоваться не только в командной строке, но и везде, где может быть задано любое выражение <expression>. Кроме всего прочего, это означает, что такие ссылки могут использоваться в теле любой конструкции deffunction и в правой части любого правила. Но если указанные ссылки содержатся в вызове функции, то их нельзя применять в качестве параметра конструкции deffunction и допускается их использование только в левой части некоторого правила. Например, все приведенные ниже конструкции являются недопустимыми.

```
(deffunction illegal-1 (?*x* ?y)
   (+ ?*x* y))
(defrule illegal-2
   (data-1 ?*x*)
   =>)
(defrule illegal-3
   (data-1 ?x&~?*x*)
   =>)
```

Команды, применяемые наряду с конструкциями defglobal

Значение переменной defglobal можно изменить с помощью команды bind. Достаточно просто вместо указания локальной переменной указать глобальную переменную. Для манипулирования конструкциями defglobal предусмотрено несколько команд. Для вывода на внешнее устройство текстового представления конструкции defglobal используется **команда ppdefglobal** (сокращение от pretty print defglobal — структурированный вывод конструкции defglobal). Для удаления конструкции defglobal применяется **команда undefglobal**. Для отображения списка конструкций defglobal, которые определены в системе CLIPS, служит **команда list-defglobals**. А для отображения имен и значений переменных defglobal, которые определены в системе CLIPS, применяется **команда show-defglobals**. **Функция get-defglobal-list** возвращает многозначное значение, содержащее список конструкций defglobal. Эти команды имеют следующий синтаксис:

```
(ppdefglobal <defglobal-name>)
(undefglobal <defglobal-name>)
(list-defglobals [<module-name>])
(show-defglobals [<module-name>])
(get-defglobal-list [<module-name>])
```

В качестве терма <defglobal-name> для команд ppdefglobal и undefglobal должно использоваться имя глобальной переменной без начальных и конечных знаков * (например, x, а не *x*). Ниже приведены примеры применения этих функций.

```
CLIPS> (ppdefglobal y)↵
(defglobal MAIN ?*y* = (+ ?*x* 1))
CLIPS> (list-defglobals)↵
x
y
For a total of 2 defglobals.
CLIPS> (get-defglobal-list)↵
(x y)
CLIPS> (show-defglobals)↵
?*x* = 3
?*y* = 4
CLIPS> (bind ?*y* 5)↵
5
CLIPS> (show-defglobals)↵
?*x* = 3
```

```
?*y* = 5
CLIPS> (undefglobal y)↵
CLIPS> (list-defglobals)↵
x
For a total of 1 defglobal.
CLIPS>
```

Принципы сброса (переустановки) значения переменной defglobal

Значение переменной defglobal восстанавливается, принимая первоначальное значение, заданное в определении, после выдачи каждой команды reset или использования команды bind для изменения значения глобальной переменной в такой форме, в которой не задано новое значение, как показано в следующем примере:

```
CLIPS> (bind ?*x* some-value)↵
some-value
CLIPS> ?*x*↵
some-value
CLIPS> (reset)↵
CLIPS> ?*x*↵
3
CLIPS> (bind ?*x* another-value)↵
another-value
CLIPS> ?*x*↵
another-value
CLIPS> (bind ?*x*)↵
3
CLIPS> ?*x*↵
3
CLIPS>
```

Такой применяемый по умолчанию принцип переустановки значений глобальных переменных можно отменить путем вызова **функции set-reset-globals** с параметром, имеющим значение FALSE, а после передачи в эту функцию значения TRUE указанный принцип переустановки снова вступает в действие. Если применяемый по умолчанию принцип переустановки значений глобальных переменных отменен, то команда reset не восстанавливает значения переменных defglobal таким образом, чтобы они принимали первоначальное значение, заданное в определении, как показано ниже.

```
CLIPS> (bind ?*x* 5)↵
5
CLIPS> (set-reset-globals FALSE)↵
TRUE
CLIPS> (reset)↵
CLIPS> ?*x*↵
5
CLIPS> (set-reset-globals TRUE)↵
FALSE
CLIPS> (reset)↵
CLIPS> ?*x*↵
3
CLIPS>
```

Отслеживание значений переменных `defglobal`

Если с помощью команды `watch` осуществляется отслеживание значений переменных `defglobal`, то при каждом изменении значения любой переменной `defglobal` выводится информационное сообщение, например, как показано ниже.

```
CLIPS> (watch globals)↵
CLIPS> (bind ?*x* 6)↵
:== ?*x* ==> 6 <== 3
6
CLIPS> (bind ?*x* 7)↵
:== ?*x* ==> 7 <== 6
7
CLIPS> (reset)↵
:== ?*x* ==> 3 <== 7
CLIPS> (unwatch globals)↵
CLIPS> (bind ?*x* 8)↵
8
CLIPS>
```

Знаки `:==` в начале информационного сообщения указывают на то, что глобальной переменной присвоено значение. Следующий символ представляет собой имя модифицируемой переменной `defglobal`. За ним следуют знаки `==>`, после чего показано новое значение глобальной переменной. За этим значением следуют знаки `<==`, а после показано старое значение глобальной переменной.

Переменные defglobal и сопоставление с шаблонами

Переменные defglobal могут использоваться в выражениях в левых частях правил, но изменения значений переменных defglobal не активизируют сопоставление с шаблонами. Например, рассмотрим следующие конструкции defglobal и defrule:

```
(defglobal ?*z* = 4)
(defrule global-example
   (data ?z&:(> ?z ?*z*))
   =>)
```

Теперь проанализируем, что произойдет после внесения в список фактов таких фактов data, которые согласуются с единственным шаблоном в правиле global-example:

```
CLIPS> (reset)↵
CLIPS> ?*z*↵
4
CLIPS> (assert (data 5) (data 6))↵
<Fact-2>
CLIPS> (facts)↵
f-0   (initial-fact)
f-1   (data 5)
f-2   (data 6)
For a total of 3 facts.
CLIPS> (agenda)↵
0     global-example: f-1
0     global-example: f-2
For a total of 2 activations.
CLIPS>
```

После внесения факта (data 5) в список фактов значение 5 связывается с локальной переменной ?z, а затем это значение сравнивается со значением глобальной переменной ?*z*, которое равно 4. Поскольку 5 больше, чем 4, то правило global-example активизируется с помощью факта f-1. Аналогичным образом, поскольку 6 больше, чем 4, то правило активизируется также под действием факта f-2. К этому моменту сопоставление с шаблонами закончено и изменение значения переменной ?*z* не приводит к повторному вычислению шаблона применительно к активизациям, сформированным для данного правила, как показано ниже.

```
CLIPS> (bind ?*z* 5)↵
5
CLIPS> (agenda)↵
```

```
0     global-example: f-1
0     global-example: f-2
For a total of 2 activations.
CLIPS>
```

Изменение значения переменной `?*z*`, которое становится равным 5, не приводит к удалению активизации, вызванной действием факта `f-1`, даже несмотря на то, что значение 5, находящееся в факте `data` и согласованное с правилом `global-example`, больше не превышает значение, хранящееся в глобальной переменной `?*z*`. Но извлечение этого факта и последующее его повторное внесение вызывает повторную активизацию процесса сопоставления с шаблонами и применение нового значения глобальной переменной, так же, как и внесение любого нового факта в список фактов:

```
CLIPS> (retract 1)↵
CLIPS> (assert (data 5))↵
<Fact-3>
CLIPS> (agenda)↵
0     global-example: f-2
For a total of 1 activation.
CLIPS> (assert (data 7))↵
<Fact-4>
CLIPS> (agenda)↵
0     global-example: f-2
0     global-example: f-4
For a total of 2 activations.
CLIPS>
```

После извлечения и последующего внесения в список фактов того же факта `(data 5)` с индексом `f-1` условие правила `global-example` больше не удовлетворяется под действием этого факта, поэтому активизация не формируется. Но внесение факта `(data 7)` в список фактов вызывает активизацию, поскольку значение переменной `?z` равно 7 и поэтому больше нового значения глобальной переменной `?*z*`, равного 5.

Использование конструкций `defglobal`

Переменные `defglobal` в наибольшей степени подходят для применения в правилах в качестве констант; кроме того, глобальные переменные могут служить для передачи информации, которая используется только в правой части правила и должна активизировать сопоставление с шаблонами. Переменные `defglobal`, применяемые в качестве констант, позволяют упростить понимание программы. Рассмотрим следующее правило:

```
(defrule plant-advisory
   (temperature ?value Fahrenheit)
   (test (<= ?value 32))
   =>
   (printout t "It's freezing." crlf)
   (printout t "Bring your plants inside." crlf))
```

Безусловно, большинство людей знают, что вода замерзает при температуре 0° по Цельсию, или при 32° по Фаренгейту, поэтому легко понять, почему так важна константа 32 в правиле plant-advisory. Но не все знают, почему важны также другие константы. Поскольку значение 32 может быть присвоено глобальной переменной, то появляется возможность воспользоваться осмысленным символическим именем в правиле plant-advisory, чтобы смысл этого правила было легче понять:

```
(defglobal ?*water-freezing-point-Fahrenheit* = 32)
(defrule plant-advisory
   (temperature ?value Fahrenheit)
   (test (<= ?value ?*water-freezing-point-Fahrenheit*))
   =>
   (printout t "It's freezing." crlf)
   (printout t "Bring your plants inside." crlf))
```

Одно из направлений использования переменных defglobal для предотвращения активизации сопоставления с шаблонами состоит в осуществлении отладки. Предположим, что требуется вывести некоторую дополнительную информацию, кроме той, что предоставляется командой watch. Один из способов выполнения этой задачи состоит во введении команды printout или format в правила примерно по такому принципу:

```
(defrule debug-example
   (data ?x)
   =>
   (printout t "Debug-example ?x = " ?x crlf))
```

Но при этом возникает проблема, связанная с тем, что такие отладочные сообщения могут не требоваться постоянно, поэтому команды их вывода либо потребуется удалять, либо обозначать комментариями на то время, когда эти команды не должны действовать. Но более удобный способ решения этой задачи состоит в том, что логическое имя устройства, в которое направляется отладочный вывод, можно сохранить в переменной defglobal. Если в качестве параметра с логическим именем в команде printout или format используется nil, то вывод не формируется (хотя команда format все еще передает отформатированную строку в качестве возвращаемого значения). Этим свойством указанных команд можно

воспользоваться, чтобы уничтожать отладочную информацию, когда вывод ее не требуется. Ниже показано, как откорректировать правило `debug-example` для использования в нем переменной `defglobal`.

```
(defglobal ?*debug-print* = nil)
(defrule debug-example
   (data ?x)
   =>
   (printout ?*debug-print* "Debug-example ?x = " ?x
             crlf))
```

По умолчанию отладочная информация передается в устройство с логическим именем `nil`, поэтому вывод команды `printout` не отображается на экране. А для того чтобы обеспечить вывод отладочной информации на экран, достаточно задать символ `t` в качестве значения глобальной переменной `?*debug-print*`, как показано в следующем диалоге:

```
CLIPS> (reset)↵
CLIPS> (assert (data a) (data b) (data c))↵
<Fact-3>
CLIPS>
(agenda)↵
0     debug-example: f-1
0     debug-example: f-2
0     debug-example: f-3
For a total of 3 activations.
CLIPS> (run 1)↵
CLIPS> (bind ?*debug-print* t)↵
t
CLIPS> (run 2)↵
Debug-example ?x = b
Debug-example ?x = c
CLIPS>
```

Обратите внимание на то, что в этом примере вывод отладочной информации был разрешен после того, как была сформирована одна из активизаций правила `debug-example`. А если бы отладочная информация была сохранена в виде факта, то возможность сделать так же не была бы предоставлена. В таком случае правило `debug-example` должно было выглядеть следующим образом:

```
(defrule debug-example
   (debug-print ?debug-print)
   (data ?x)
```

```
=>
(printout ?debug-print "Debug-example ?x = " ?x crlf))
```

Извлечение факта `debug-print` и его последующее внесение в список фактов с новым значением привели бы к повторной активизации всех тех активизаций правила `debug-example`, которые уже были запущены, а это может оказаться весьма нежелательным.

10.5 Конструкции `defgeneric` и `defmethod`

В языке CLIPS предусмотрена возможность не только определять функции с использованием конструкции `deffunction`, но и объявлять универсальные функции. **Универсальная функция** по существу представляет собой группу взаимосвязанных функций (называемых *методами*), в которых совместно применяется общее имя. В действительности универсальная функция, которая включает больше одного метода, называется **перегруженной** (поскольку в принципе предоставляет возможность ссылаться больше чем на один метод). Каждый метод в группе имеет свою собственную сигнатуру; *сигнатура* характеризует количество и типы параметров, принимаемых методом. Во время обработки вызова универсальной функции в системе CLIPS анализируются параметры и вызывается на выполнение метод с сигнатурой, соответствующей параметрам, если есть таковой. Этот процесс известен под названием **вызова перегруженного метода** (generic dispatch). Для определения общего имени универсальной функции, используемого группой методов, служит **конструкция defgeneric**. Конструкция `defgeneric` имеет следующий общий формат:

```
(defgeneric <defgeneric-name> [<optional-comment>])
```

Единственная причина, по которой когда-либо следует явно определять конструкцию `defgeneric`, состоит в том, чтобы использовать ее в качестве предварительного объявления при применении ссылки на универсальную функцию, прежде чем фактически определены какие-либо методы универсальной функции. Если определяется некоторый метод универсальной функции, прежде чем фактически задается конструкция `defgeneric`, то происходит автоматическое создание конструкции `defgeneric` от имени программиста.

Конкретные методы для универсальной функции определяются с использованием **конструкции defmethod**. Конструкция `defmethod` имеет такой общий формат:

```
(defmethod <defgeneric-name> [<index>]
   [<optional-comment>]
   (<regular-parameter-restriction>*
```

```
   [<wildcard-parameter-restriction>])
<expression>*)
```

Конкретные конструкции `defmethod` не имеют уникальных имен. Один и тот же параметр `<defgeneric-name>` используется для всех методов, из которых состоит универсальная функция. Но каждому конкретному методу присваивается уникальный целочисленный индекс, который может использоваться для ссылки на метод. Предусмотрена также возможность присвоить методу определенный индекс, задав значение `<index>` при определении метода. А если значение `<index>` не задано, то система CLIPS автоматически создает его от имени программиста. В составе каждой определенной универсальной функции может быть только один метод, имеющий отличную от других сигнатуру параметров (значений, передаваемых в качестве термов `<regular-parameter-restriction>` и `<wildcard-parameter-restriction>`). Если же объявляется метод универсальной функции, имеющий такую же сигнатуру параметров, что и у существующего метода этой универсальной функции, а значение `<index>` не задается, то существующий метод заменяется новым методом и для него используется тот же индекс. А если при определении метода задается значение `<index>` и существует метод универсальной функции, имеющий такую же сигнатуру параметра, то значение `<index>` должно совпадать с индексом существующего метода, имеющего такую же сигнатуру; в результате этого существующий метод заменяется новым методом.

Каждый параметр `<regular-parameter-restriction>` может иметь одну из двух форм. Он может представлять собой либо однозначную переменную (как в конструкции `deffunction`), либо иметь следующую форму:

```
(<single-field-variable> <type>* [<query>])
```

Во второй форме, которая характеризуется использованием круглых скобок, за однозначной переменной следуют от нуля и больше обозначений типов, а затем представлен необязательный запрос. Допустимыми значениями для термов `<type>` могут быть любые допустимые имена классов. Эта тема рассматривается более подробно в главе 11, а в примерах данной главы эти обозначения ограничиваются символами, уже применявшимися с атрибутом `type`, которые (как следует отметить) являются также именами классов `SYMBOL`, `STRING`, `LEXEME`, `INTEGER`, `FLOAT`, `NUMBER`, `INSTANCE-NAME`, `INSTANCE-ADDRESS`, `INSTANCE`, `FACT-ADDRESS` и `EXTERNAL-ADDRESS`. Значением `<query>`, которое в случае его использования должно находиться на последнем месте, должна быть либо глобальная переменная, либо вызов функции.

Терм `<wildcard-parameter-restriction>` имеет формат, аналогичный параметру `<regular-parameter-restriction>`, за исключением того, что в нем вместо однозначных переменных используются многозначные перемен-

ные. Поэтому параметр <wildcard-parameter-restriction> имеет такие две формы: либо многозначную переменную, либо следующую форму:

```
(<multifield-variable> <type>* [<query>])
```

Если не считать того, что в этой форме вместо терма <single-field-variable> (однозначная переменная) используется параметр <multifield-variable> (многозначная переменная), значения, которые могут быть подставлены вместо термов <type> и <query>, подчиняются таким же ограничениям, как и в форме <regular-parameter-restriction>. При вызове метода многозначная переменная, заданная в качестве параметра <wildcard-parameter-restriction>, действует аналогично параметру с подстановочным символом конструкции deffunction. При этом все оставшиеся параметры в вызове метода, которые выходят за пределы количества параметров для данного метода, группируются в виде одного многозначного значения и присваиваются переменной, заданной в позиции параметра с подстановочным символом.

Последняя часть определения defmethod, представленная термом <expression>*, является телом метода. Тело метода, как и правая часть конструкции defrule или тела конструкции deffunction, — это ряд выражений, которые выполняются в указанном порядке после вызова метода.

Еще один вариант конструкции deffunction с именем check-input

Вернемся снова к конструкции deffunction с именем check-input, которая рассматривалась в разделе 10.3. Первоначальным определением этой конструкции было следующее:

```
(deffunction check-input (?question $?values)
   (printout t ?question " " ?values " ")
   (bind ?answer (read))
   (while (not (member$ ?answer ?values))
      (printout t ?question " " ?values " ")
      (bind ?answer (read)))
   (return ?answer))
```

Данное объявление deffunction можно непосредственно преобразовать в объявление defmethod, заменив имя конструкции, таким образом:

```
(defmethod check-input (?question $?values)
   (printout t ?question " " ?values " ")
   (bind ?answer (read))
   (while (not (member$ ?answer ?values))
      (printout t ?question " " ?values " ")
```

```
      (bind ?answer (read)))
   (return ?answer))
```

Следует отметить, что объявление конструкции `deffunction` нельзя заменить объявлением универсальной функции с тем же именем, или наоборот, поэтому, чтобы иметь возможность определить конструкцию `defmethod` с именем `check-input`, необходимо вначале удалить конструкцию `deffunction` с именем `check-input`. Конструкция `defmethod` с именем `check-input`, объявленная выше, действует полностью идентично ранее рассматривавшейся в данной главе конструкции `deffunction`. Тем не менее, если не будут определены еще какие-либо методы с тем же именем, то в действительности нет смысла использовать универсальную функцию, а не применявшуюся ранее конструкцию `deffunction`. Поэтому рассмотрим еще одну часть программы Sticks. Как было отмечено в разделе 8.9, для определения количества палочек, удаленных игроком-человеком, используются следующие три правила:

```
(defrule get-human-move
   (player-move h)
   (pile-size ?size)
   (test (> ?size 1))
   =>
   (printout t
   "How many sticks do you wish to take? ")
   (assert (human-takes (read))))
(defrule good-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (and (integerp ?choice)
              (>= ?choice 1)
              (<= ?choice 3)
              (< ?choice ?size)))
   =>
   (retract ?whose-turn ?number-taken)
   (printout t "Human made a valid move" crlf))
(defrule bad-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (or (not (integerp ?choice))
                 (< ?choice 1)
                 (> ?choice 3)
```

```
                (>= ?choice ?size)))
   =>
   (printout t "Human made an invalid move" crlf)
   (retract ?whose-turn ?number-taken)
   (assert (player-move h)))
```

Безусловно, количество палочек, которое может быть разрешено удалить человеку, не всегда равно 1, 2 или 3, поэтому невозможно просто вывести для пользователя приглашение с указанием количества палочек, как в следующем вызове:

```
(check-input " How many sticks do you wish to take?"
 1 2 3)
```

Вместо этого количество удаляемых палочек должно вычисляться динамически, как показывает следующее модифицированное правило get-human-move:

```
(defrule get-human-move
   (player-move h)
   (pile-size ?size)
   (test (> ?size 1))
   =>
   (bind ?responses (create$))
   (bind ?upper-choice (min (- ?size 1) 3))
   (loop-for-count (?i ?upper-choice)
      (bind ?responses (create$ ?responses ?i)))
      (bind ?answer
      (check-input "How many sticks do you wish to take?"
      ?responses))
   (assert (human-takes ?answer)))
```

В первых четырех строках правила get-human-move динамически создается многозначное значение, содержащее множество допустимых значений, из которых может выбирать пользователь. Это множество может состоять только из чисел 1, 2 или 3, но должно также быть меньше количества оставшихся палочек. Но в программе было бы неудобно вычислять все допустимые ответы, а поскольку все возможные ответы отображаются при выводе вопроса пользователю, то если бы в правиле check-input пришлось передать просьбу пользователю выбрать одно значение от 1 до 100, то было бы необходимо вывести все 100 значений. Эту проблему можно устранить, введя дополнительный метод. В частности, ниже приведен метод, в котором используются ограничения типа для осуществления особого принципа действия, в котором после вопроса в вызове правила check-input задаются два целых числа.

```
(defmethod check-input ((?question STRING)
```

```
            (?value1 INTEGER)
            (?value2 INTEGER))
   (printout t ?question " (" ?value1 "-" ?value2 ") ")
   (bind ?answer (read))
   (while (or (not (integerp ?answer))
      (< ?answer ?value1)
      (> ?answer ?value2))
      (printout t ?question " (" ?value1 "-" ?value2 ") ")
      (bind ?answer (read)))
   (return ?answer))
```

В этом методе частично используется такой же код, как и в первоначальном методе `ask-user`, но имеются существенные различия, из которых наиболее важным является список параметров. Каждый из параметров заключается в круглые скобки, а также содержит спецификацию типа. Спецификацией типа для переменной `?question` является `STRING`, для переменной `?value1` — `INTEGER`, а для переменной `?value2` — также `INTEGER`. В связи с наличием таких ограничений данный конкретный метод вызывается, только если функции `ask-user` передаются три параметра, причем первый из них имеет тип `STRING`, второй — `INTEGER` и третий — также `INTEGER`.

Еще одной отличительной особенностью данного метода является то, что вслед за вопросом в нем предусматривается вывод информации о диапазоне допустимых целых чисел, а не выводится каждое допустимое целое число. Наконец, еще одной особенностью данного метода является то, что проверка значения `member$` в цикле `while` заменена в нем тремя проверками, позволяющими убедиться в том, что в качестве ответа введено целое число и что это число находится в допустимом диапазоне. После определения обоих этих методов экспериментальный прогон, результаты которого приведены ниже, показал, что оба метода работают должным образом.

```
CLIPS> (check-input "Pick a number" 1 3)↵
Pick a number (1-3) a↵
Pick a number (1-3) 34↵
Pick a number (1-3) 3↵
3
CLIPS> (check-input "Pick a number" 1 2 3)↵
Pick a number (1 2 3) a↵
Pick a number (1 2 3) 34↵
Pick a number (1 2 3) 3↵
3
CLIPS>
```

Обратите внимание на то, что после выбора первого метода был отображен допустимый диапазон, (1-3), а после вызова второго метода на внешнее устройство выведен список допустимых значений, (1 2 3).

Приоритеты методов

После вызова универсальной функции в системе CLIPS выполняется только один из ее методов. Процесс определения того, какой метод должен быть выполнен, называется **вызовом перегруженного метода**. В предыдущем примере было показано, что в универсальной функции check-input применяется другой принцип действия при ее вызове с указанием целочисленного диапазона, а не списка допустимых значений. Вторым выражением в рассматриваемом примере было следующее:

```
(check-input "Pick a number" 1 2 3)
```

Две сигнатуры методов функции check-input были определены таким образом:

```
(?question $?values)
(?question STRING) (?value1 INTEGER)
(?value2 INTEGER)
```

Очевидно, что сигнатура первого метода может использоваться для вычисления этого выражения, поскольку она требует один или больше параметров, но сигнатура второго метода неприменима для вычисления этого выражения, так как она требует точно три параметра, а задано четыре. С другой стороны, анализ первого выражения в рассматриваемого примере:

```
(check-input "Pick a number" 1 3)
```

и его сравнение с сигнатурами объявленных методов показывает, что данное выражение может быть обработано с помощью обеих сигнатур методов. Еще раз отметим, что сигнатура первого метода просто принимает от одного или больше параметров любого типа, а сигнатура второго метода предусматривает применение точно трех параметров, причем первый параметр должен иметь тип STRING, второй — INTEGER и третий — INTEGER. Поскольку выражение содержит параметры точно в таком количестве и такого типа, как требуется, данный метод также является применимым. Каким же образом система CLIPS определяет метод, подлежащий вызову на выполнение, если применимыми являются несколько методов?

Ответ на этот вопрос состоит в том, что система CLIPS определяет порядок приоритетов методов, относящихся к данной конкретной универсальной функции. Если применимыми являются несколько методов, то на выполнение вызывается метод с самым высоким приоритетом. Для ознакомления с порядком приоритетов методов, применимых для обработки конкретного множества параметров, может

использоваться **команда preview-generic**. Эта команда имеет следующий синтаксис:

```
(preview-generic <defgeneric-name> <expression>*)
```

Параметр `<defgeneric-name>` представляет собой имя универсальной функции, а параметр `<expression>*` соответствует параметрам в количестве от нуля и больше, которые должны быть переданы в универсальную функцию. При вызове функции `preview-generic` универсальная функция фактически не выполняется. Определяются лишь все применимые методы, а затем отображаются в виде списка в порядке приоритетов, например, как показано ниже.

```
CLIPS>
(preview-generic check-input "Pick a number" 1 3)↵
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #1 () $?
CLIPS>
```

Этот вывод подтверждает то, что было уже определено опытным путем, — метод `check-input`, в котором проверяется принадлежность к целочисленному диапазону, имеет более высокий приоритет, чем метод `check-input`, в котором проверяется значение с использованием параметра с подстановочным символом. В первую очередь в списке отображается метод `#2`, т.е. метод проверки принадлежности к целочисленному диапазону, а это означает, что данный метод имеет более высокий приоритет. В этом списке вначале отображается имя универсальной функции, а затем индекс метода. После этого выводятся допустимые типы для каждого параметра в круглых скобках. Имена переменных в списке параметров конструкции `defmethod` не влияют на приоритет, поэтому включать их в список нет необходимости. Следующим по порядку в списке находится метод `#1`. На первый параметр не налагаются какие-либо ограничения типа, поэтому отображается пустая пара круглых скобок, `()`. Последним параметром в методе `#1` является параметр с подстановочным символом. Если для параметра с подстановочным символом не заданы какие-либо ограничения типа, то команда `preview-generic` отображает для представления этого параметра символ `$?`.

Если в системе CLIPS приходится проводить различия между двумя методами универсальной функции, то для определения метода с более высоким приоритетом используются описанные ниже этапы.

1. Сравнить самые левые неисследованные ограничения параметров этих двух методов. Если очередной параметр в списке параметров имеется только в одном методе, перейти к шагу 6. Если неисследованные параметры не остались ни в одном методе, перейти к шагу 7. В противном случае перейти к шагу 2.

2. Если один параметр является обычным параметром, а другой — параметром с подстановочным символом, то метод с обычным параметром имеет более высокий приоритет. Если оба параметра не различаются по данному признаку, перейти к шагу 3.
3. Если в одном из параметров заданы ограничения типа, а в другом параметре — нет, то метод с ограничениями типа имеет более высокий приоритет. Если оба параметра не различаются по данному признаку, перейти к шагу 4.
4. Сравнить крайние левые неисследованные ограничения типа двух параметров. Если ограничение типа в одном параметре является более конкретным по сравнению с ограничением типа во втором параметре, то метод с более конкретным ограничением типа имеет более высокий приоритет. Например, ограничение типа `INTEGER` является более конкретным, чем `NUMBER`. Если ни одно ограничение типа не является более конкретным (например, ограничение типа `INTEGER` не более конкретно, чем `LEXEME`) и в обоих параметрах есть еще неисследованные ограничения типа, то повторить шаг 4. Если же в одном из параметров есть еще неисследованные ограничения типа, а в другом параметре — нет, то метод без дополнительных ограничений типа имеет более высокий приоритет. Если оба параметра не различаются по данному признаку, перейти к шагу 5.
5. Если в одном параметре имеется ограничение запроса, а в другом — нет, то метод с ограничением запроса имеет более высокий приоритет. В противном случае возвратиться к шагу 1 и сравнить следующее множество параметров.
6. Если следующим параметром метода, в котором остались неисследованные параметры, является обычный параметр, то данный метод имеет более высокий приоритет. В противном случае более высокий приоритет имеет другой метод.
7. Более высокий приоритет имеет метод, который первым прошел проверку.

Блок-схема процедуры, позволяющей получить результат, эквивалентный тому, который может быть получен с помощью шагов от 1 до 7, показан на рис. 10.1. Шаг 1 начинается с действия, обозначенного прямоугольником в левом верхнем углу рисунка. А на рис. 10.2 показана блок-схема процедуры, позволяющей получить результат, эквивалентный тому, который может быть получен с помощью шагов 3 и 4, касающийся определения того, имеет ли один параметр более конкретные ограничения типа по сравнению с другим параметром.

Используя описанные выше шаги, можно легко определить, почему метод `#2` универсальной функции `check-input` имеет более высокий приоритет, чем метод `#1`. Начиная с шага 1, вначале сравним крайние левые неисследованные параметры обоих методов. Ограничением параметра для метода `#1` является `()`, а ограничением параметра для метода `#2` — `(STRING)`. Поскольку оба метода

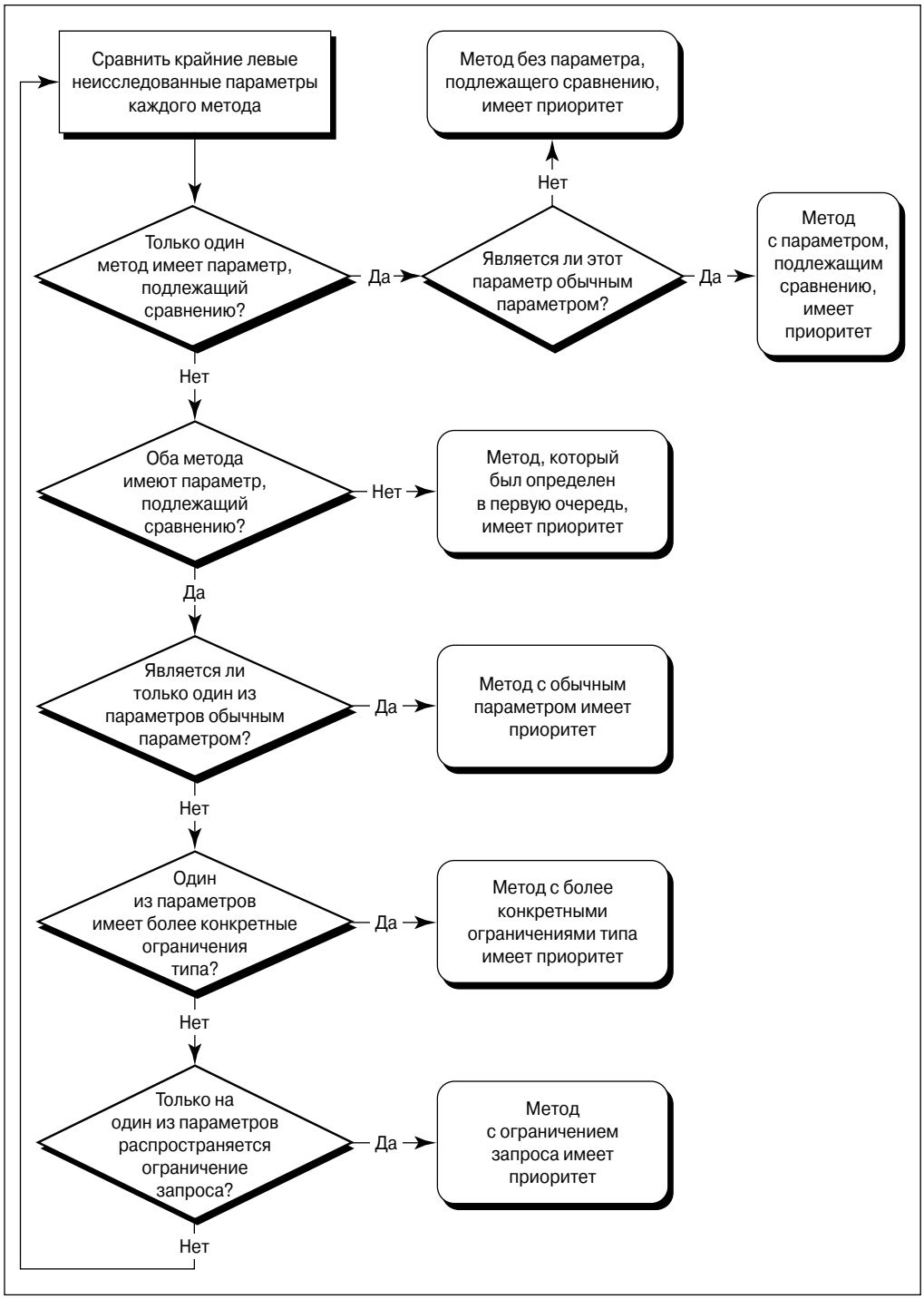


Рис. 10.1. Определение приоритета метода

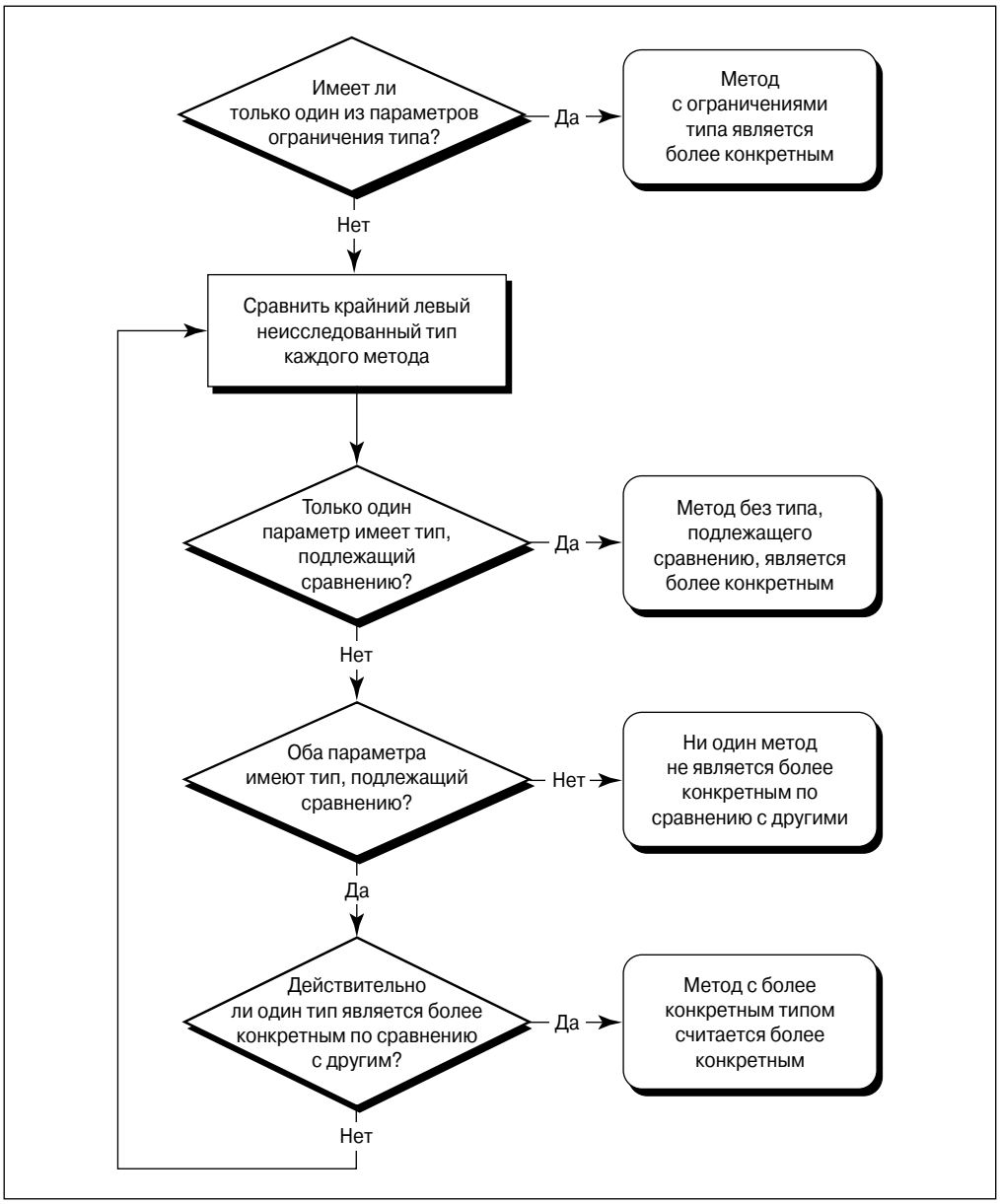


Рис. 10.2. Определение того, являются ли ограничения типа одного параметра более конкретными по сравнению с другими параметрами

имеют и другие параметры, переходим к шагу 2. Оба этих параметра являются обычными параметрами, поэтому переходим к шагу 3. На этом шаге выясняем,

что ограничение типа имеет только метод #2, и это означает, что метод #2 имеет более высокий приоритет по сравнению с методом #1.

Итак, метод #2 универсальной функции check-input позволяет вводить только целочисленные значения, поэтому создадим еще один метод, который даст возможность вводить значение с плавающей запятой, принадлежащее к указанному диапазону, следующим образом:

```
(defmethod check-input ((?question STRING)
                        (?value1 NUMBER)
                        (?value2 NUMBER))
   (printout t ?question " (" (float ?value1)
                  "-" (float ?value2) ") ")
   (bind ?answer (read))
   (while (or (not (numberp ?answer))
      (< ?answer ?value1)
      (> ?answer ?value2))
      (printout t ?question " (" (float ?value1)
                     "-" (float ?value2)
                     ") ")
      (bind ?answer (read)))
   (return ?answer))
```

Этот метод является аналогичным описанному ранее методу проверки принадлежности к целочисленному диапазону, за исключением того, что в ограничениях параметров вместо типа INTEGER задан тип NUMBER, при отображении допустимого диапазона осуществляется принудительное преобразование целочисленных значений диапазона в значения с плавающей точкой, а для проверки на наличие недопустимых типов используется предикативная функция numberp, а не integerp. После определения этого дополнительного метода был проведен экспериментальный прогон, который показал, что оба метода проверки принадлежности к диапазону работают должным образом, как показано ниже.

```
CLIPS> (check-input "Pick a number" 1 3.5)↵
Pick a number (1.0-3.5) 6↵
Pick a number (1.0-3.5) 2.7↵
2.7
CLIPS> (check-input "Pick a number" 1 3)↵
Pick a number (1-3) 2.7↵
Pick a number (1-3) 2↵
2
CLIPS>
```

С помощью команды preview-generic можно показать, что для обработки второго вызова универсальной функции check-input из приведенных выше вызовов применимы все три метода:

```
CLIPS>
(preview-generic check-input "Pick a number" 1 3)↵
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #3 (STRING) (NUMBER) (NUMBER)
check-input #1 () $?
CLIPS>
```

Метод с двумя ограничениями INTEGER имеет более высокий приоритет, чем метод с двумя ограничениями NUMBER. Причину этого можно легко выяснить с помощью той же описанной выше процедуры определения порядка приоритетов. Начиная с шага 1, сравним крайние левые неисследованные параметры обоих методов. Ограничением параметра для методов #2 и #3 является (STRING). Оба метода имеют и другие параметры, поэтому перейдем к шагу 2. Поскольку оба параметра являются обычными параметрами, перейдем к шагу 3. Оба параметра имеют ограничения типа, поэтому перейдем к шагу 4. В связи с тем, что оба ограничения типа являются одинаковыми, перейдем к шагу 5. Ни один из параметров не имеет ограничения запроса, следовательно, мы должны возвратиться к шагу 1. Крайним левым неисследованным ограничением параметра для метода #2 является (INTEGER), а для метода #3 крайним левым неисследованным ограничением параметра является (NUMBER). Оба метода имеют параметры, поэтому перейдем к шагу 2. Поскольку оба параметра представляют собой обычные параметры, перейдем к шагу 3. Оба параметра имеют ограничения типа, поэтому перейдем к шагу 4. А в связи с тем, что ограничение типа INTEGER в методе #2 является более конкретным, чем ограничение типа NUMBER в методе #3, метод #2 имеет более высокий приоритет.

Рассмотрим еще один пример определения приоритета. В данном случае определим метод, который по существу является таким же, как и последний определенный метод, за исключением того, что вместо использования NUMBER в качестве ограничения типа определим другие два типа, INTEGER и FLOAT:

```
(defmethod check-input ((?question STRING)
                        (?value1 INTEGER FLOAT)
                        (?value2 INTEGER FLOAT))
   (printout t ?question " (" (float ?value1)
                  "-" (float ?value2) ") ")
   (bind ?answer (read))
   (while (or (not (numberp ?answer))
              (< ?answer ?value1)
              (> ?answer ?value2))
```

```
   (printout t ?question " (" (float ?value1)
                  "-" (float ?value2) ") ")
      (bind ?answer (read)))
   (return ?answer))
```

Результаты вызова команды `preview-generic` показывают, что новый метод `#4` занял свое место между методами `#2` и `#3`:

```
CLIPS>
(preview-generic check-input "Pick a number" 1 3)↵
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #4 (STRING) (INTEGER FLOAT) (INTEGER FLOAT)
check-input #3 (STRING) (NUMBER) (NUMBER)
check-input #1 () $?
CLIPS>
```

Такое расположение методов по приоритетам стало результатом применения действий шага 4 ко второму параметру методов. Сравнение методов `#3` и `#4` показывает, что первое ограничение типа в методе `#4`, `INTEGER`, является более конкретным, чем первое ограничение типа в методе `#3`, `NUMBER`, поэтому метод `#4` имеет более высокий приоритет. А сравнение методов `#2` и `#4` показывает, что первое ограничение типа в методе `#2`, `INTEGER`, является таким же, как и первое ограничение типа в методе `#4`, поэтому переходим к анализу второго ограничения типа в каждом из методов. Поскольку метод `#2` не имеет второго ограничения типа, а метод `#4` имеет, то метод `#2` обладает более высоким приоритетом.

Ограничения запроса

В параметре метода допускается не только задавать ограничения типа, но и определять ограничение запроса. **Ограничение запроса** — это ссылка на конструкцию `defglobal` или вызов функции, выполнение которого позволяет определить применимость метода при вызове универсальной функции. Если обработка ограничения запроса приводит к получению символа `FALSE`, то данный метод является неприменимым. Ограничение запроса для параметра не вычисляется, если не удовлетворяются ограничения типа для этого параметра. Любой метод с несколькими параметрами может иметь несколько ограничений запроса, причем каждое из этих ограничений должно удовлетворяться, для того чтобы метод мог стать применимым.

Рассмотрим пример, в котором может оказаться полезным ограничение запроса. Если будет вызвана универсальная функция `check-input`, но не заданы дополнительные параметры, которые должны следовать за строкой вопроса, то функция будет действовать следующим образом:

```
CLIPS> (check-input "Pick a number")↵
```

```
Pick a number () 3
Pick a number () 2
Pick a number () 5
Pick a number () 0
 .
 .
 .
```

В данной ситуации применим метод #1, в котором используется параметр с подстановочным символом. Но за параметром со строкой вопроса, ?question, не заданы дополнительные параметры, поэтому параметр с подстановочным символом, $?values, связывается с многозначным значением, имеющим длину нуль. Проверка значения member$, применяемая в данном методе, всегда возвращает символ FALSE, поскольку в пустое многозначное значение невозможно включить ни одно значение, вводимое пользователем, поэтому из вызова данного метода так и не выполняется возврат. Эту проблему можно устранить, наложив на параметр с подстановочным символом ограничение запроса:

```
(defmethod check-input
(?question ($?values (= (length$ ?values) 0)))
   (return FALSE))
```

В данном методе количество значений, хранящихся в параметре с подстановочным символом, $?values, определяется путем вызова функции length$, а возвращаемое значение с помощью функции = сравнивается с нулем. Это позволяет обеспечить применимость данного метода только к пустому многозначному значению. Обратите внимание на то, что параметры указанного метода (в данном случае ?value) можно использовать в рамках ограничения запроса. Все, что выполняет этот метод, сводится к тому, что он возвращает символ FALSE, поскольку пользователь не имеет возможности ввести допустимое значение. Но благодаря применению этого определения метода такой же вызов универсальной функции, как и перед этим, не приводит к созданию бесконечного цикла, как показано ниже.

```
CLIPS> (check-input "Pick a number")↵
FALSE
CLIPS>
```

Вызов команды preview-generic применительно к методу, в котором задан лишь параметр со строкой запроса, показывает, что при всех прочих одинаковых параметрах метод #2 получает более высокий приоритет, чем другой применимый метод, #5, благодаря наличию в нем ограничения запроса:

```
CLIPS>
(preview-generic check-input "Pick a number")↵
check-input #5 () ($? <qry>)
```

```
check-input #1 () $?
CLIPS>
```

Рассмотрим еще одну ситуацию, в которой в универсальной функции check-input может возникнуть бесконечный цикл:

```
CLIPS> (check-input "Pick a number" 3 1)↵
Pick a number (3-1) 3
Pick a number (3-1) 2
Pick a number (3-1) 5
Pick a number (3-1) 0
 .
 .
 .
```

В данном случае применимым является метод #2, в котором имеются один параметр со строкой вопроса и два целочисленных параметра. Но такого целого числа, которое было бы больше или равно 3 и меньше или равно 1, не существует, поэтому пользователь так и не сможет передать значение, которое соответствовало бы данной универсальной функции. Эту проблему можно устранить путем создания дополнительного метода с ограничением запроса, позволяющим определить то, что верхний предел меньше нижнего, следующим образом:

```
(defmethod check-input ((?question STRING)
                        (?value1 INTEGER)
                        (?value2 INTEGER
                              (< ?value2 ?value1)))
   (return FALSE))
```

Этот метод после обнаружения того, что пределы перепутаны местами, возвращает символ FALSE. Мы могли бы пройти на шаг дальше и просто переставлять значения в неправильно заданных диапазонах. Одним из способов решения этой задачи могло стать обычное дублирование кода, а также взаимная перестановка местами переменных в соответствующих местах:

```
(defmethod check-input ((?question STRING)
                        (?value1 INTEGER)
                        (?value2 INTEGER
                              (< ?value2 ?value1)))
   (printout t ?question " (" ?value2 "-" ?value1 ") ")
   (bind ?answer (read))
   (while (or (not (integerp ?answer))
              (< ?answer ?value2)
              (> ?answer ?value1))
      (printout t ?question " (" ?value2 "-"
```

```
                        ?value1 ") ")
      (bind ?answer (read)))
   (return ?answer))
```

Этот подход осуществляется успешно, но приводит к дублированию большего объема кода, чем необходимо. Лучший способ состоит в том, чтобы снова вызвать универсальную функцию, но с параметрами, переставленными местами:

```
(defmethod check-input
((?question STRING) (?value1 INTEGER)
   (?value2 INTEGER (< ?value2 ?value1)))
   (check-input ?question ?value2 ?value1))
```

Такая организация работы позволяет избежать возникновения бесконечного цикла из-за того, что параметры находятся в неправильном порядке, как показано ниже.

```
CLIPS> (check-input "Pick a number" 3 1)↵
Pick a number (1-3) 3
3
CLIPS>
```

Для того чтобы обеспечить полный охват всех возможных случаев, необходимо ввести дополнительный метод для осуществления еще одного метода, в котором используется тип NUMBER для задания диапазонов:

```
(defmethod check-input
((?question STRING) (?value1 NUMBER)
   (?value2 NUMBER (< ?value2 ?value1)))
   (check-input ?question ?value2 ?value1))
```

После этого вызов команды preview-generic с пределами диапазона, заданными в обратном порядке, показывает значительное увеличение количества применимых методов:

```
CLIPS>
(preview-generic check-input "Pick a number" 3 1)↵
check-input #6 (STRING) (INTEGER) (INTEGER <qry>)
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #4 (STRING) (INTEGER FLOAT) (INTEGER FLOAT)
check-input #7 (STRING) (NUMBER) (NUMBER <qry>)
check-input #3 (STRING) (NUMBER) (NUMBER)
check-input #1 () $?
CLIPS>
```

Отслеживание работы универсальных функций и методов

При отслеживании работы универсальных функций с помощью команды `watch` каждый раз, когда начинается или заканчивается выполнение универсальной функции, выводится информационное сообщение. А если с помощью команды `watch` отслеживается работа методов, то информационное сообщение отображается каждый раз, когда начинается или заканчивается выполнение метода. В качестве примера рассмотрим следующий вывод:

```
CLIPS> (watch methods)↵
CLIPS> (watch generic-functions)↵
CLIPS> (check-input "Pick a number" 3 1)↵
GNC >> check-input ED:1 ("Pick a number" 3 1)
MTH >> check-input:#6 ED:1 ("Pick a number" 3 1)
GNC >> check-input ED:2 ("Pick a number" 1 3)
MTH >> check-input:#2 ED:2 ("Pick a number" 1 3)
Pick a number (1-3) 2↵
MTH << check-input:#2 ED:2 ("Pick a number" 1 3)
GNC << check-input ED:2 ("Pick a number" 1 3)
MTH << check-input:#6 ED:1 ("Pick a number" 3 1)
GNC << check-input ED:1 ("Pick a number" 3 1)
2
CLIPS>
```

Обозначение `GNC` в начале информационного сообщения показывает, что это сообщение относится к универсальной функции. Символ `>>` показывает, что начинается вызов универсальной функции, а символ `<<` свидетельствует о том, что вызов универсальной функции закончен. Следующим символом является имя универсальной функции; в данном случае рассматривается универсальная функция `check-input`. Символ `ED` является сокращением от “Evaluation Depth”. За этим символом следует двоеточие, а затем показано значение текущей глубины вложенности вызовов в виде целого числа. Глубина вложенности вызовов показывает, как осуществляется вызов вложенных универсальных функций и конструкций `deffunction`. Отсчет этого значения начинается с нуля, а затем увеличивается на единицу после каждой передачи управления в очередную конструкцию `deffunction` или в универсальную функцию. А после каждого выхода из конструкции `deffunction` или из универсальной функции значение глубины вложенности уменьшается на единицу. Наконец, последний фрагмент отображаемой информации показывает фактические параметры вызова универсальной функции. Кроме того, информационные сообщения, относящиеся к методам, обозначенные в начале буквами `MTH`, содержат практически ту же информацию, что и сообщения, относящиеся к универсальным функциям. Основная отличительная особенность сообщения последнего типа состоит в том, что в нем вслед за именем

универсальной функции показан индекс метода. Индекс метода позволяет узнать, какой именно метод вызван на выполнение.

В этом примере видно, что завершено выполнение двух методов. Вначале управление передается в метод #6, поскольку параметр с обозначением конца диапазона, равный 1, меньше, чем параметр с обозначением начала диапазона, равный 3. Затем снова вызывается универсальная функция check-input, но на этот раз с переставленными местами параметрами. Поскольку теперь параметры заданы в правильном порядке, вызывается на выполнение метод #2, пользователь вводит значение 2, находящееся в допустимом диапазоне, и происходит возврат из обоих методов. Обычно гораздо удобнее отслеживать работу методов, а не универсальных функций, поскольку желательно знать, какой конкретный метод вызывается на выполнение, но предусмотрена возможность отслеживать работу и универсальных функций, и методов.

Команды defmethod

Для манипулирования конструкциями defmethod предусмотрено несколько команд. Для отображения текстового представления конструкции defmethod используется **команда ppdefmethod** (сокращение от pretty print defmethod — структурированный вывод конструкции defmethod). Для удаления конструкции defmethod может служить **команда undefmethod**. Для отображения списка конструкций defmethod, определенных в программе CLIPS, применяется **команда list-defmethods**. Наконец, **функция get-defmethod-list** возвращает многозначное значение, содержащее список конструкций defmethod. Эти команды имеют следующий синтаксис:

```
(ppdefmethod <defgeneric-name> <index>)
(undefmethod <defgeneric-name> <index>)
(list-defmethods [<defgeneric-name>])
(get-defmethod-list [<defgeneric-name>])
```

Команды ppdefmethod и undefmethod отличаются от команд, используемых для работы с другими конструкциями, тем, что в них, кроме имени конструкции defgeneric, необходимо указывать индекс. Кроме того, функции list-defmethods и get-defmethod-list не принимают необязательный параметр с именем модуля. Вместо этого необязательным параметром в них служит имя универсальной функции. Если это имя указано, то команда применяется только к методам этой универсальной функции; в ином случае она применяется ко всем методам всех универсальных функций. Ниже приведены примеры, в которых показано, как используются эти функции.

```
CLIPS> (ppdefmethod check-input 5)↵
(defmethod MAIN::check-input
```

```
   (?question ($?values (= (length$ ?values) 0)))
   (return FALSE))
CLIPS> (undefmethod check-input 4)↵
CLIPS> (list-defmethods)↵
check-input #6 (STRING) (INTEGER) (INTEGER <qry>)
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #7 (STRING) (NUMBER) (NUMBER <qry>)
check-input #3 (STRING) (NUMBER) (NUMBER)
check-input #5 () ($? <qry>)
check-input #1 () $?
For a total of 6 methods.
CLIPS> (get-defmethod-list check-input)↵
(check-input 6 check-input 2 check-input 7
 check-input 3 check-input 5 check-input 1)
CLIPS>
```

Обратите внимание на то, что методы, перечисленные в выводе команды `list-defmethods`, показаны в порядке приоритета. Кроме того, возвращаемое значение функции `get-defmethod-list` состоит из пар, которые включают имя универсальной функции и индекс метода.

Команды `defgeneric`

В языке CLIPS предусмотрено несколько команд для манипулирования конструкциями `defgeneric`. Для отображения текстового представления конструкции `defgeneric` используется **команда ppdefgeneric** (сокращение от pretty print defgeneric — структурированный вывод конструкции `defgeneric`). Для удаления конструкции `defgeneric` служит **команда undefgeneric**. **Команда list-defgenerics** применяется для отображения списка конструкций `defgeneric`, определенных в программе CLIPS. А **функция get-defgeneric-list** возвращает многозначное значение, содержащее список конструкций `defgeneric`. Эти команды имеют следующий синтаксис:

```
(ppdefgeneric <defgeneric-name>)
(undefgeneric <defgeneric-name>)
(list-defgenerics [<module-name>])
(get-defgeneric-list [<defgeneric-name>])
```

Как показывает следующий диалог, команда `undefgeneric` удаляет не только указанную конструкцию `defgeneric`, но и все связанные с ней методы:

```
CLIPS> (list-defgenerics)↵
check-input
For a total of 1 defgeneric.
```

```
CLIPS> (undefgeneric check-input)↵
CLIPS> (list-defgenerics)↵
CLIPS> (list-defmethods)↵
CLIPS>
```

Перегруженные функции и команды

Для конструкций deffunction и универсальных функций не могут совместно использоваться общие имена, но универсальные функции позволяют перегружать функции, определяемые пользователем. Предположим, например, что необходимо предусмотреть аналогичные операции для сложения типов данных, отличных от чисел. Вообще говоря, как показано ниже, в системе CLIPS это не допускается.

```
CLIPS> (+ 3 4)↵
7
CLIPS> (+ "red" "blue")↵
[ARGACCES5] Function + expected argument #1 to be
of type integer or float
CLIPS> (+ (create$ a b c) (create$ d e f))↵
[ARGACCES5] Function + expected argument #1 to be
of type integer or float
CLIPS>
```

Попытка сложить две строки или два многозначных значения с использованием функции + приводит к возникновению ошибки. Но если требуется реализовать операцию сложения для строк как конкатенацию этих строк и операцию сложения для многозначных значений как создание комбинации этих значений, то подобные функциональные средства можно создать, определяя методы, в которых для этой цели применяются функции str-cat и create$:

```
(defmethod + ((?x LEXEME) (?y LEXEME))
   (str-cat ?x ?y))
(defmethod + ((?x MULTIFIELD) (?y MULTIFIELD))
   (create$ ?x ?y))
```

После определения этих методов появляется возможность применять операции сложения данных указанных новых типов без возникновения ошибки:

```
CLIPS> (+ "red" "blue")↵
"redblue"
CLIPS> (+ (create$ a b c) (create$ d e f))↵
(a b c d e f)
CLIPS>
```

Для вывода на внешнее устройство информации о двух новых методах, которые были определены наряду с оригинальной функцией +, заданной в языке CLIPS (обозначаемой индексом метода SYS1), можно воспользоваться вызовом команды list-defmethods:

```
CLIPS> (list-defmethods +)↵
+ #SYS1 (NUMBER) (NUMBER) ($? NUMBER)
+ #2 (LEXEME) (LEXEME)
+ #3 (MULTIFIELD) (MULTIFIELD)
For a total of 3 methods.
CLIPS>
```

Обратите внимание на то, что система CLIPS способна при создании сигнатуры метода выявить типы параметров функции +, определенной в системе. В данном случае можно видеть, что функция + принимает два или несколько числовых параметров.

Если в программе требуется применить перегрузку функции, определяемой пользователем, то такую операцию следует выполнить, прежде чем применять какие-либо конструкции, которые ссылаются на эту функцию, определяемую пользователем. Это условие можно соблюсти, прежде чем делать ссылку на соответствующую функцию, либо явно задавая конструкцию defgeneric, либо задавая конструкцию defgeneric неявно, путем определения конструкции defmethod. В ссылках конструкций на определяемую пользователем функцию, сделанных до ее перегрузки, не будет использоваться механизм вызова перегруженного метода, поэтому всегда будет вызываться неперегруженная, определяемая пользователем функция. С другой стороны, в ссылках конструкций на определяемую пользователем функцию, сделанных после ее перегрузки, будет применяться механизм вызова перегруженного метода.

10.6 Процедурные конструкции и конструкции defmodule

В модулях может осуществляться импорт и экспорт не только конструкций deftemplate, но и на основе аналогичных принципов могут импортироваться и экспортироваться процедурные конструкции defglobal, deffunction и defgeneric. Поэтому четыре из возможных операторов export и import, описанных в главе 9 применительно к конструкциям deftemplate, распространяют свое действие и на процедурные конструкции:

```
(export ?ALL)
(export ?NONE)
(import <module-name> ?ALL)
```

```
(import <module-name> ?NONE)
```

Оператор в первой форме позволяет экспортировать из модуля все экспортируемые конструкции; оператор во второй форме показывает, что не экспортируется ни одна конструкция; оператор в третьей форме импортирует все экспортированные конструкции из указанного модуля; а оператор в четвертой форме не импортирует ни одну из конструкций, экспортированных из указанного модуля.

Каждая из конструкций defglobal, deffunction и defgeneric имеет также связанные с ней операторы import и (или) export, которые позволяют указать, следует ли выполнять импорт и (или) экспорт всех, ни одной или указанного множества конструкций, как показано ниже.

```
(export defglobal ?ALL)
(export defglobal ?NONE)
(export defglobal <defglobal-name>+)
(export deffunction ?ALL)
(export deffunction ?NONE)
(export deffunction <deffunction-name>+)
(export defgeneric ?ALL)
(export defgeneric ?NONE)
(export defgeneric <defgeneric-name>+)
(import <module-name> defglobal ?ALL)
(import <module-name> defglobal ?NONE)
(import <module-name> defglobal
                      <defglobal-name>+)
(import <module-name> deffunction ?ALL)
(import <module-name> deffunction ?NONE)
(import <module-name> deffunction
                      <deffunction-name>+)
(import <module-name> defgeneric ?ALL)
(import <module-name> defgeneric ?NONE)
(import <module-name> defgeneric
                      <defgeneric-name>+)
```

При осуществлении импорта и (или) экспорта конкретных конструкций defglobal не нужно задавать знаки * в начале и в конце имени конструкции defglobal. Например, следует указывать имя переменной water-freezing-point-Fahrenheit, а не *water-freezing-point-Fahrenheit*. Модуль, импортирующий конструкцию defgeneric, импортирует также все ее методы. Возможность импортировать или экспортировать только конкретные методы отсутствует. Если в модуле не импортируется какая-либо конструкция defglobal, deffunction или defgeneric из другого модуля, то остается возможность создать собственное определение такой конструкции с использованием того

же имени. Применение ссылки на конструкцию, которая не была экспортирована и импортирована должным образом, приводит к возникновению ошибки, например, как показано ниже.

```
CLIPS> (clear)↵
CLIPS>
(defmodule MAIN
   (export deffunction function-1)
   (export defglobal global-1)
   (export defgeneric generic-1))↵
CLIPS>
(deffunction MAIN::function-1 (?x)
   (+ 1 ?x))↵
CLIPS>
(deffunction MAIN::function-2 (?x)
   (+ 2 ?x))↵
CLIPS>
(defglobal MAIN ?*global-1* = 1
   ?*global-2* = 2)↵
CLIPS>
(defmodule EXAMPLE
   (import MAIN deffunction ?ALL)
   (import MAIN defglobal global-1))↵
CLIPS>
(defglobal EXAMPLE ?*global-2* = 3)↵
CLIPS>
```

В модуле EXAMPLE неявно импортируется конструкция `defglobal` с именем `global-1` из модуля MAIN с использованием ключевого слова ?ALL, а также явно импортируется конструкция `deffunction` с именем `function-1` из модуля MAIN путем указания имени этой конструкции. С результатами выполнения операторов `import` и `export` можно ознакомиться, предприняв попытку получить доступ к каждой из этих конструкций в каждом модуле:

```
CLIPS> (get-current-module)↵
EXAMPLE
CLIPS> (function-1 1)↵
2
CLIPS> (function-2 1)↵
[EXPRNPSR3] Missing function declaration for
function-2.
CLIPS> ?*global-1*↵
1
```

```
CLIPS> ?*global-2*↵
3
CLIPS> (set-current-module MAIN)↵
EXAMPLE
CLIPS> (function-1 1)↵
2
CLIPS> (function-2 1)↵
3
CLIPS>
?*global-1*↵
1
CLIPS>
?*global-2*↵
2
CLIPS>
```

10.7 Полезные команды и функции

Загрузка и сохранение фактов

Быстродействие программы CLIPS можно повысить, уменьшив количество фактов в списке фактов. Один из способов сокращения количества фактов состоит в том, чтобы факты загружались в систему CLIPS только тогда, когда они потребуются. Например, в программе, предназначенной для диагностирования неисправностей в автомобилях, можно вначале запросить информацию об изготовителе и модели автомобиля, а затем загрузить информацию, относящуюся к данному автомобилю. В языке CLIPS предусмотрены функции **load-facts** и **save-facts**, позволяющие загружать факты из файла и сохранять их в файле. Эти две функции имеют следующий синтаксис:

```
(load-facts <file-name>)
(save-facts <file-name>
            [<save-scope> <deftemplate-names>*])
```

В данном определении параметр `<save-scope>` имеет такую форму:

```
visible | local
```

Функция `load-facts` загружает группу фактов, которые хранятся в файле, указанном параметром `<file-name>`. Факты, заданные в файле, должны быть представлены в стандартном формате упорядоченных фактов или фактов, определяемых с помощью конструкции `deftemplate`. Например, если файл "`facts.dat`" содержит следующую информацию:

```
(data 34)
(data 89)
(data 64)
(data 34)
```

то такая команда загрузит все факты, содержащиеся в этом файле:

```
(load-facts "facts.dat")
```

Для сохранения фактов из списка фактов в файле, указанном параметром <file-name>, может использоваться функция save-facts. Факты сохраняются в таком же порядке, какой требуется для функции load-facts. Если параметр <save-scope> не задан или определен как **local**, то в файле сохраняются только факты, соответствующие конструкциям deftemplate, которые определены в текущем модуле. Если же параметр <save-scope> задан как **visible**, то в файле сохраняются все факты, соответствующие конструкциям deftemplate, которые являются видимыми в текущем модуле. Если используется параметр <save-scope>, то предоставляется возможность также задать одно или несколько имен конструкций deftemplate. В таком случае сохраняются только факты, соответствующие указанным конструкциям deftemplate (но имена конструкций deftemplate так или иначе должны соответствовать спецификации local или visible).

И функция load-facts, и функция save-facts возвращают символ TRUE, если файл с фактами был успешно открыт, а затем загружен или сохранен; в противном случае эти функции возвращают символ FALSE. Именно программист обязан обеспечить, чтобы конструкции deftemplate, соответствующие фактам, заданным с помощью конструкции deftemplate и хранящимся в файле фактов, были видимыми в текущем модуле, в котором выполняется команда load-facts.

Команда system

Команда system обеспечивает выполнение команд операционной системы из среды CLIPS. Команда system имеет следующий синтаксис:

```
(system <expression>+)
```

Например, приведенное ниже правило позволяет получить листинг указанного каталога на компьютере, работающем под управлением операционной системы Unix.

```
(defrule list-directory
(list-directory ?directory)
   =>
   (system "ls " ?directory))
```

В данном примере первым параметром команды `system`, "`ls `", является команда Unix, предназначенная для получения списка файлов, записанных в каталоге. Обратите внимание на то, что за буквами `l` и `s` следует пробел. Дело в том, что команда `system` просто соединяет все свои параметры, заданные в виде строк, в одну строку, после чего предоставляет операционной системе возможность выполнить сформированную команду. Поэтому все пробелы, которые должны присутствовать в команде операционной системы, необходимо включить в состав параметров в вызове команды `system`.

Результаты выполнения команды `system` могут зависеть от операционной системы. К тому же не все операционные системы предоставляют функциональные возможности для реализации команды `system`, поэтому не следует рассчитывать на то, что эта команда всегда будет доступна в языке CLIPS. Команда `system` не возвращает значение, поэтому невозможно непосредственно возвратить значение в систему CLIPS после выполнения команды операционной системы.

Команда `batch`

Команда batch позволяет считывать непосредственно из файла команды и ответы, которые в обычных условиях должны быть введены в приглашении верхнего уровня. Команда `batch` имеет следующий синтаксис:

```
(batch <file-name>)
```

Например, предположим, что для прогона программы CLIPS необходимо провести следующий диалог, в котором показаны команды и ответы, предназначенные для ввода пользователем (напомним, что текст, обозначенный полужирным шрифтом, показывает, какие знаки пользователь должен вводить с клавиатуры):

```
CLIPS> (load "rules1.clp")↵
**************
CLIPS> (load "rules2.clp")↵
******************
CLIPS> (load "rules3.clp")↵
***********
CLIPS> (reset)↵
CLIPS> (run)↵
How many iterations? 10↵
Starting value? 1↵
End value? 20↵
Completed
CLIPS>
```

Эти команды и ответы, необходимые для прогона программы, можно сохранить в файле, как показано ниже.

```
(load "rules1.clp")↵
(load "rules2.clp")↵
(load "rules3.clp")↵
(reset)↵
(run)↵
10↵
1↵
20↵
```

Предположим, что файл с командами и ответами получил имя "commands.bat"; в таком случае следующий диалог показывает, как воспользоваться командой batch для его выполнения (еще раз отметим, что полужирным шрифтом отмечены знаки, которые должны быть введены с клавиатуры):

```
CLIPS> (batch "commands.bat")↵
CLIPS> (load "rules1.clp")↵
**************
CLIPS> (load "rules2.clp")↵
******************
CLIPS> (load "rules3.clp")↵
***********
CLIPS> (reset)↵
CLIPS> (run)↵
How many iterations? 10↵
Starting value? 1↵
End value? 20↵
Completed
CLIPS>
```

После считывания всех команд и ответов из пакетного файла режим работы с клавиатурой в приглашении верхнего уровня возвращается к нормальному.

При эксплуатации исполняемой программы CLIPS в операционной системе, которая поддерживает возможность задания параметров командной строки для исполняемых файлов (такой как Unix), система CLIPS может автоматически выполнить команды из пакетного файла при запуске. Предположим, что исполняемый файл CLIPS может быть вызван путем ввода слова “clips”; в таком случае синтаксис команды операционной системы для вызова на выполнение пакетного файла при запуске системы CLIPS становится таковым:

```
clips -f <file-name>
```

Применение опции -f эквивалентно вводу команды (batch <file-name>) непосредственно после запуска системы CLIPS. Вызовы команды batch могут быть вложенными.

Команды dribble-on и dribble-off

Для сохранения в протоколе всех результатов вывода на терминал и ввода с клавиатуры может служить **команда dribble-on**, которая имеет следующий синтаксис:

```
(dribble-on <file-name>)
```

После вызова команды `dribble-on` на выполнение осуществляется эхо-повтор всех результатов вывода на терминал и ввода с клавиатуры не только на терминале, но и в файле, указанном с помощью параметра `<file-name>`.

Действие команды `dribble-on` можно отменить с помощью **команды dribble-off**, синтаксис которой приведен ниже.

```
(dribble-off)
```

Выработка случайных чисел

Для выработки случайных целочисленных значений применяется **функция random**. Она имеет следующий синтаксис:

```
(random [<start-expression> <end-expression])
```

В этом определении параметры `<start-expression>` и `<end-expression>`, если они заданы, должны представлять собой целочисленные значения и показывать диапазон целых чисел, которым ограничиваются возвращаемые случайные целые числа. Например, ниже приведена конструкция `deffunction` с именем `roll-die`, в которой используется функция `random` для моделирования броска шестигранной игральной кости путем выработки целочисленного значения от 1 до 6.

```
(deffunction roll-die ()
   (random 1 6))
```

Многократные вызовы конструкции `deffunction` с именем `roll-die` приводят к получению различных результатов:

```
CLIPS> (roll-die)↵
6
CLIPS> (roll-die)↵
1
CLIPS> (roll-die)↵
4
CLIPS> (roll-die)↵
1
CLIPS>
```

Для задания начального значения генератора случайных чисел может использоваться **функция seed**, которая позволяет в дальнейшем воспроизвести прежнюю

последовательность возвращаемых значений с использованием того же начального значения. Функция seed имеет следующий синтаксис:

```
(seed <integer-expression>)
```

В этом определении параметр <integer-expression> представляет собой начальное значение. Результат применения команды seed для воспроизведения одинаковой последовательности случайных значений можно наблюдать на примере такого ряда команд:

```
CLIPS> (seed 30)↵
CLIPS> (roll-die)↵
4
CLIPS> (roll-die)↵
5
CLIPS> (roll-die)↵
3
CLIPS> (seed 30)↵
CLIPS> (roll-die)↵
4
CLIPS> (roll-die)↵
5
CLIPS> (roll-die)↵
3
CLIPS>
```

Преобразование строки в поле

Для преобразования строкового значения поля в значение поля может использоваться **функция string-to-field**, синтаксис которой приведен ниже.

```
(string-to-field <string-expression>)
```

В этом определении параметр <string-expression> представляет собой строковое поле, которое должно быть подвергнуто синтаксическому анализу и преобразовано в поле, например, таким образом:

```
CLIPS> (string-to-field "7")↵
7
CLIPS> (string-to-field " 3.4 2.1 3")↵
3.4
CLIPS>
```

В первом примере применения вызова функции string-to-field берется строковое поле "7" и преобразуется в целочисленное значение поля 7. Во втором примере вызова функции string-to-field показано, что в функцию переда-

ется несколько полей, содержащихся в строковом параметре. Функция возвращает первое найденное поле, поле с плавающей точкой `3.4`, и отбрасывает все остальные поля, находящиеся в этой строке. Обратите внимание на то, что возвращенное значение не зависит от наличия в строке дополнительных пробелов. По существу вызов функции `string-to-field` идентичен вызову функции `read`, в котором строковый параметр представляет то, что пользователь вводит с клавиатуры.

Поиск символа

Для отображения всех символов, определенных в системе CLIPS, которые содержат указанную подстроку, может использоваться **команда apropos**. Она имеет следующий синтаксис:

```
(apropos <symbol-or-string-expression>)
```

В этом определении параметр `<symbol-or-string-expression>` представляет собой искомую подстроку. Команда `apropos` может принести реальную пользу, если требуется вывести на внешнее устройство список функций или команд, в именах которых имеется общая подстрока, или в тех случаях, когда удается вспомнить часть символа, применяемого в имени функции, команды или конструкции, но не все имя. Например, предположим, что необходимо получить список всех функций и команд, содержащих символ `deftemplate`:

```
CLIPS> (apropos deftemplate)↵
get-deftemplate-list
deftemplate
list-deftemplates
undeftemplate
ppdeftemplate
deftemplate-module
CLIPS>
```

Сортировка списка полей

Для сортировки списка полей может использоваться **функция sort**. Функция `sort` имеет следующий синтаксис:

```
(sort <comparison-function-name> <expression>*)
```

В этом определении параметр `<comparison-function-name>` представляет собой имя функции, конструкции `deffunction` или универсальной функции, которая применяется для определения того, следует ли поменять местами два поля во время сортировки. Оставшимися параметрами, обозначаемыми с помощью терма `<expression>*`, могут быть либо однозначные, либо многозначные значения. Эти значения соединяются в одно многозначное значение, которое затем

сортируется. Возвращаемым значением этой функции является отсортированное многозначное значение. В следующем примере показано, как осуществляется сортировка списка целых чисел:

```
CLIPS> (sort > 4 3 5 7 2 7)↵
(2 3 4 5 7 7)
CLIPS>
```

Область применения функции `sort` не ограничивается сортировкой чисел. Например, рассмотрим следующую конструкцию `deffunction`:

```
(deffunction string> (?a ?b)
   (> (str-compare ?a ?b) 0))
```

Любая функция сравнения, используемая в сочетании с функцией `sort`, должна принимать два параметра и возвращать символ `TRUE`, если первый параметр в отсортированном списке должен стоять после второго параметра; в противном случае функция сравнения должна возвращать символ `FALSE`. В частности, функция `str-compare` возвращает положительное целое число, если первый ее параметр лексикографически больше второго параметра, поэтому, сравнивая возвращаемое значение с 0 с помощью функции >, можно воспользоваться конструкцией `deffunction` с именем `string>` для лексикографической сортировки списка символов или строк, например, как показано ниже.

```
CLIPS> (sort string> ax aa bk mn ft m)↵
(aa ax bk ft m mn)
CLIPS>
```

В функции `str-compare` все прописные буквы трактуются как имеющие меньшее значение по сравнению со всеми строчными буквами, поэтому при сортировке смешанных комбинаций прописных и строчных букв полученный результат может отличаться от ожидаемого, как в следующем примере:

```
CLIPS> (sort string> a C b A B c)↵
(A B C a b c)
CLIPS>
```

Как показывает этот пример, все прописные буквы в результате сортировки оказались расположенными перед всеми строчными буквами. Для того чтобы прописные и строчные буквы трактовались на равных, функцию `string>` можно модифицировать следующим образом:

```
(deffunction string> (?a ?b)
   (bind ?rv (str-compare (lowcase ?a)
   (lowcase ?b)))
   (if (= ?rv 0)
      then
      (> (str-compare ?a ?b) 0)
```

```
      else
      (> ?rv 0)))
```

В данном случае буквы в строках вначале преобразуются в строчные, а затем проводится сравнение строк. Регистр букв учитывается, только если строки равны. Как показано ниже, это приводит к получению более приемлемых результатов.

```
CLIPS> (sort string> a C b A B c)↵
(A a B b C c)
CLIPS>
```

При использовании функции `sort` необходимо учитывать еще одно важное требование, состоящее в том, что функция сравнения должна возвращать символ `TRUE`, только если два сравниваемых значения действительно необходимо поменять местами. Рассмотрим следующий вызов:

```
CLIPS> (sort <> 3 4 5 6)↵
...
```

В этой ситуации в функции `sort` вызывается функция <> для определения того, какие поля должны быть переставлены местами. Но ни одни из этих полей не равны, поэтому функция <> всегда возвращает символ `TRUE` и сортировка никогда не заканчивается. Безусловно, следует избегать возникновения в программе такой ситуации.

10.8 Резюме

В конструкции `deffunction`, универсальной функции или в правой части правила для передачи управления могут использоваться функции `if`, `while`, `switch`, `loop-for-count`, `progn$` и `break`. Но злоупотребление этими функциями в правой части правила считается плохим стилем программирования. Для прекращения выполнения правил может применяться функция `halt`.

Для определения новых функций по такому же принципу, как и в других процедурных языках, может использоваться конструкция `deffunction`; при этом не требуется применять компилятор C для повторной компиляции исходного кода интерпретатора CLIPS. Еще один вариант создания новых функций состоит в использовании механизма, предусмотренного в языке CLIPS, который описан в документе *Advanced Programming Guide*. Этот механизм позволяет интегрировать в систему CLIPS функции, написанные на языке C, но для этого необходимо применять компилятор C, чтобы создать новый исполняемый файл CLIPS.

Конструкция `defglobal` позволяет определять переменные, которые сохраняют свои значения за пределами области действия конструкций, называемые *глобальными переменными*. Универсальные функции, реализованные с помощью конструкций `defgeneric` и `defmethod`, предоставляют бóльшие возможно-

сти и обладают большей гибкостью, чем конструкции deffunction, поскольку позволяют связывать многочисленные процедурные методы с общим именем универсальной функции. При вызове универсальной функции применяется механизм вызова перегруженного метода для исследования типов параметров, передаваемых в функцию, и проверки всех соответствующих ограничений запроса для определения метода, подлежащего вызову.

Кроме того, в данной главе приведено вводное описание нескольких вспомогательных функций. Функции save-facts и load-facts могут использоваться для сохранения фактов в файле и загрузки фактов из файла. Команда system позволяет вызывать на выполнение команды операционной системы из среды CLIPS. Команда batch дает возможность вводить последовательности команд и ответов, хранящихся в файле, вместо обычного их ввода с клавиатуры. Команды dribble-on и dribble-off обеспечивают регистрацию информации, выводимой на терминал, для сохранения ее в файле. Для выработки случайных целочисленных значений может использоваться функция random, а для задания начального значения генератора случайных чисел предназначена функция seed. Функция string-to-field может служить для синтаксического анализа и извлечения поля, содержащегося в строке. Команда apropos применяется для отображения всех символов, содержащих заданную подстроку. Функция sort может использоваться для сортировки списка полей.

Задачи

10.1. Перезапишите следующее правило в виде одного или нескольких правил, в которых не используются функции while и if. Убедитесь в том, что составленные вами правила осуществляют те же действия, сравнив полученные результаты и окончательное состояние списка фактов при использовании составленных вами правил и исходного правила.

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? ")
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no)) do
      (printout t "Continue? ")
      (bind ?answer (read)))
   (if (eq ?answer yes)
      then (assert (phase continue))
      else (assert (phase halt))))
```

10.2. Предположим, что дана шахматная доска размерами N×N, где N — целое число. Напишите программу, которая расставляет N ферзей на шахматной доске таким образом, что ни один ферзь не может напасть на другого. *Подсказка.* Вначале разработайте программу с использованием четырех вертикалей и горизонталей. Это — минимальное число, для которого существует решение (не считая тривиального случая, в котором доска имеет размеры 1×1).

10.3. Модифицируйте программу, разработанную в результате решения задачи 9.12 (см. с. 772), таким образом, чтобы при отсутствии кустарников, соответствующих всем требованиям, создавался список, состоящий из кустарников, соответствующих большинству требований. Дополнительно должно быть выведено количество выполненных требований.

10.4. Модифицируйте программу, разработанную в результате решения задачи 9.17 (см. с. 775), таким образом, чтобы в ней использовались модули. Если не удается найти драгоценный камень, соответствующий указанным значениям цвета, твердости и плотности, программа должна сообщить об этом. После идентификации драгоценного камня программа должна предоставить пользователю возможность идентифицировать еще один драгоценный камень.

10.5. Объедините программы, разработанные в результате решения задач 9.14 и 9.15 (см. с. 773 и 774). Результирующая программа должна иметь интерфейс текстового меню с пунктами меню для запуска приложения, прекращения работы приложения и выхода из программы. После выбора пункта меню, предназначенного для запуска, для пользователя должно быть выведено приглашение, чтобы он мог указать имя приложения и требования к памяти. После выбора пункта меню, предназначенного для завершения работы приложения, для пользователя должно быть выведено приглашение, чтобы он мог указать имя завершаемого приложения.

10.6. Модифицируйте программу, разработанную в результате решения задачи 9.15 (см. с. 774), чтобы в ней поддерживались субменю, например, как показано ниже.

```
CLIPS> (run)
Select one of the following options:
   1 - Option A
   2 - Option B
   3 - Submenu 1
   9 - Quit Program
Your choice: 3
Select one of the following options:
   1 - Option C
```

```
   2 - Option D
   9 - Previous Menu
Your choice: 9
Select one of the following options:
   1 - Option A
   2 - Option B
   3 - Submenu 1
   9 - Quit Program
Your choice: 9
CLIPS>
```

10.7. Модифицируйте программу, разработанную в результате решения задачи 9.18 (см. с. 775), таким образом, чтобы она позволяла составлять расписание занятий по шести предметам для одного учащегося. Программа должна предпринимать попытки максимизировать общую оценку (как описано в условиях задачи 9.18) для всех шести предметов. Входные данные программы должны состоять из шести фактов с указанием предметов, которые должны быть включены в расписание, а выходные данные должны представлять собой список предметов, включенных в расписание в порядке следования времени проведения занятий от одного до шести. Проверьте функционирование разработанной вами программы на примере выбора предметов, показанного в табл. 10.1.

Таблица 10.1. Исходные данные для задачи 10.7

Предмет	Более предпочтительные преподаватели	Более предпочтительное время проведения занятий	Менее предпочтительные преподаватели	Менее предпочтительное время проведения занятий
История Техаса	Хилл	2, 5	–	1, 3
Алгебра	Смит	1, 2	Джоунз	6
Физкультура	–	–	Мак, Кинг	1
Химия	Долби	5, 6	–	–
Литература	–	3, 4	–	1, 6
Немецкий язык	–	–	–	–

10.8. Предположим, что игрок в покер только что обновил свои пять карт. Напишите программу, которая после получения входных фактов, представляющих эти пять карт, выводит на внешнее устройство обозначение типа комбинации карт на руках игрока: флэш ройял (десятка, валет, дама, король и туз одной масти), флэш стрит (комбинация карт, представляющая собой одновременно флэш и стрит), карэ (четыре карты одного ранга), полный дом (три карты одного ранга плюс две карты другого ранга), флэш (все карты одной масти), стрит (комбинация карт, идущих по старшинству непосредственно одна за другой, не обязательно одной масти), трио (три карты одного ранга), две пары (две карты одного ранга плюс две карты другого ранга), одна пара (две карты одного ранга) или ничего.

10.9. Напишите программу, которая упрощает алгебраические уравнения, перемещая все константы в правую часть уравнения, а все переменные — в левую часть уравнения, после чего сокращает общие члены. Например, следующее уравнение:

$$2x + y + 5 + 3y - 2z - 8 = 3z - 4y + 4$$

после упрощения должно принять такой вид:

$$2x + 8y - 5z = 7$$

Поскольку знак = имеет в шаблонах специальное значение, по-видимому, потребуется представить уравнение в таком формате, что знак = не указан явно, например, как показано ниже.

```
(equation (LHS 2 x + y + 5 + 3 y - 2 z - 8)
   (RHS 3 z - 4 y + 4))
```

10.10. Объедините программы, разработанные в результате решения задачи 9.19 (см. с. 776) и задачи 10.6, для создания интерфейса к программе определения конфигурации, действующей под управлением меню. Опции главного меню должны предоставлять возможность пользователю выбирать шасси и приспособления, которые должны быть включены в конфигурацию, исключать приспособления из создаваемой конфигурации и выводить на внешнее устройство информацию о стоимости конфигурации. Субменю должны использоваться для обеспечения выбора шасси, а также для включения или исключения из конфигурации различных приспособлений. После выбора шасси, добавления или удаления какого-то приспособления должны отображаться предупреждающие сообщения, если обнаруживается, что количество приспособлений превышает количество отсеков или потребность приспособлений в электроэнергии превышает возможности шасси. После этого управление должно возвращаться к главному меню. После выбора опции меню, предназначенной для вывода информации о конфигурации,

должен создаваться список выбранных шасси и приспособлений, наряду с указанием стоимости каждого отдельного компонента, а также суммарной стоимости всех выбранных шасси и приспособлений.

10.11. Используя алгоритм, указанный в разделе 2.2, разработайте конструкцию `deffunction`, которая преобразует строку

137179766832525156430015

в строку "`GOLD 438+`".

Подсказка. Для извлечения цифр из строки используйте функции `sub-string` и `string-to-field`, а затем примените флажок `%c` в параметре функции `format` для преобразования числового значения в цифру.

10.12. Напишите конструкции `deffunction`, которые выполняют операции объединения и пересечения множеств применительно к двум многозначным значениям. Обратите внимание на то, что при выполнении операций объединения и пересечения множеств необходимо исключать дублирующиеся поля из значений, возвращаемых функциями.

10.13. Напишите конструкцию `deffunction`, которая вычисляет экспериментальное значение вероятности (как описано в разделе 4.6), полученное при выпадении единицы на шестигранной игральной кости. Параметром этой конструкции `deffunction` должно быть количество бросков, а возвращаемым значением — эмпирически определенная вероятность.

10.14. Напишите конструкцию `deffunction`, которая определяет все простые числа от `1` до указанного целого числа и возвращает эти простые числа в виде многозначного значения.

10.15. Напишите конструкцию `deffunction`, которая определяет количество вхождений одной строки в другой строке.

10.16. Напишите конструкцию `deffunction`, которая построчно сравнивает два файла и выводит информацию об обнаруженных различиях в файл, указанный логическим именем.

10.17. Напишите конструкцию `deffunction`, которая принимает от нуля и больше параметров и возвращает многозначное значение, содержащее значения параметров в обратном порядке.

10.18. Напишите конструкцию `deffunction`, в которой не используется рекурсия для вычисления факториала целого числа `N`.

10.19. Напишите конструкцию `deffunction`, которая преобразовывает двоичную строку, состоящую из нулей и единиц, в десятичное число.

10.20. Без использования функций `if` или `switch` напишите ряд методов, предназначенных для преобразования данных, представленных с помощью таких единиц, как дюймы (`inches`), футы (`feet`) и ярды (`yards`). Например, вызов `(convert 3 feet inches)` должен возвращать `36`. Кроме того, напишите еще один ряд методов, предназначенных для преобразования данных, представленных с помощью таких единиц, как сантиметры (`centimeters`), метры (`meters`) и километры (`kilometers`).

10.21. На основе методов, разработанных в результате решения задачи 10.20, перегрузите функцию + с помощью методов, позволяющих складывать значения длины, представленные в разных единицах измерения. Возвращаемый результат должен быть представлен в таких единицах измерения, в каких задан первый параметр метода +. Например, вызов `(+ 3 feet 12 inches)` должен возвратить `4`. Поддерживать сложение метрических и британских единиц измерения длины пока еще не требуется.

10.22. Перегрузите функцию – с помощью метода, который удаляет поля, содержащиеся в одном многозначном значении, из другого многозначного значения. Например, вызов `(- (create$ a b c d) (create$ b d f))` должен возвратить многозначное значение `(a c)`.

10.23. Напишите по одному методу для каждого из четырех случаев в определении *S*-функции, описанной в разделе 5.5.

10.24. На основе определения степенно́го множества, приведенного в условиях задачи 2.11 (см. с. 190), напишите конструкцию `deffunction`, которая выводит на внешнее устройство каждое из множеств степенно́го множества, полученного на основе множества, которое представлено в виде многозначного значения.

10.25. Пробег автомобиля Джека в милях, `N`, выражается целым числом от 200 000 до 300 000. В десятичном представлении числа `N` имеется точно одна цифра 0. Число `N` — квадрат. Сумма квадратов десятичных цифр числа `N` — также квадрат. Напишите одну или несколько конструкций `deffunction`, которые определяют значение `N`.

Глава 11

Классы, экземпляры и обработчики сообщений

11.1 Введение

В языке CLIPS для представления данных, кроме фактов, используются также экземпляры (или объекты). Экземпляры создаются на основании классов, которые определяются с помощью языка COOL (Object-Oriented Language — объектно-ориентированный язык CLIPS). Так же как структура фактов задается с использованием конструкций `deftemplate`, структура экземпляров задается с применением конструкций `defclass`. Использование экземпляров и классов вместо фактов и конструкций `deftemplate` предоставляет несколько преимуществ. Первым из них является само наследование. Конструкция `defclass` может наследовать информацию от одного или нескольких различных классов. Это позволяет создавать более структурированные, модульные определения данных. Вторым преимуществом является то, что за объектами можно закрепить относящуюся к ним процедурную информацию с помощью обработчиков сообщений. Третьим преимуществом является то, что сопоставление с шаблонами на основе объектов обеспечивает бóльшую гибкость, чем сопоставление с шаблонами на основе фактов. В объектных шаблонах может использоваться наследование, сопоставление с шаблонами может осуществляться с учетом слотов, принадлежащих нескольким классам, существует возможность исключить повторную активизацию шаблона под действием изменений в незаданных слотах, а также может обеспечиваться поддержание истинности на основе значений слотов.

11.2 Конструкция `defclass`

Прежде чем появится возможность создания экземпляров, в систему CLIPS необходимо передать информацию о списке допустимых слотов для данного конкретного класса. Для этой цели применяется **конструкция defclass**. В своей наиболее фундаментальной форме эта конструкция весьма напоминает конструкцию `deftemplate`:

```
(defclass <class-name> [<optional-comment>]
   (is-a <superclass-name>)
   <slot-definition>*)
```

В этом определении терм `<superclass-name>` определяет класс, от которого данный, вновь создаваемый класс должен наследовать информацию. Классом, от которого в конечном итоге наследуют информацию все определяемые пользователем классы, является системный класс `USER`. Определяемый пользователем класс должен наследовать информацию либо от другого определяемого пользователем класса, либо от класса `USER`. Синтаксическое описание `<slot-definition>` определено следующим образом:

```
(slot <slot-name> <slot-attribute>*) |
(multislot <slot-name> <slot-attribute>*)
```

С помощью этого синтаксиса экземпляр `person` может быть описан с использованием такой конструкции `defclass`:

```
(defclass PERSON "Person defclass"
   (is-a USER)
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color))
```

Обратите внимание на то, что в данном случае применялось имя слота `full-name`, а не `name`, в отличие от конструкции `deftemplate` с именем `person`, приведенной в качестве примера в разделе 7.6. Как будет описано ниже, при выполнении операций сопоставления с шаблоном объекта символ `name` используется в качестве зарезервированного символа, который имеет особое значение.

При определении слотов конструкции `defclass` могут также применяться следующие атрибуты слота из конструкции `deftemplate`: **type**, **range**, **cardinality**, **allowed-symbols**, **allowed-strings**, **allowed-lexemes**, **allowed-integers**, **allowed-floats**, **allowed-numbers**, **allowed-values**, **allowed-instance-names**, **default** и **default-dynamic**. Пример применения таких атрибутов показан ниже.

```
(defclass PERSON "Person defclass"
   (is-a USER)
```

```
   (slot full-name
      (type STRING))
   (slot age
      (type INTEGER)
      (range 0 120))
   (slot eye-color
      (type SYMBOL)
      (allowed-values brown blue green)
      (default brown))
   (slot hair-color
      (type SYMBOL)
      (allowed-values black brown red blonde)
      (default brown)))
```

Атрибуты слота для конструкций `defclass` называют также *фасетами слота*.

11.3 Создание экземпляров

Экземпляры создаются с использованием **команды make-instance**. Команда `make-instance` имеет следующий синтаксис:

```
(make-instance [<instance-name-expression>]
               of <class-name-expression>
               <slot-override>*)
```

В этом определении терм `<slot-override>` имеет такую форму:

```
(<slot-name-expression> <expression>)
```

Например, ниже показано, как создать некоторые экземпляры, применяя конструкцию `defclass` с именем `person`.

```
CLIPS>
(make-instance [John] of PERSON
   (full-name "John Q. Public")
   (age 24)
   (eye-color blue)
   (hair-color black))↵
[John]
CLIPS> (make-instance of PERSON)↵
[gen1]
CLIPS> (make-instance Jack of PERSON)↵
[Jack]
CLIPS> (instances)↵
[John] of PERSON
```

```
[gen1] of PERSON
[Jack] of PERSON
For a total of 3 instances.
CLIPS>
```

В этом примере были созданы три экземпляра с именами экземпляров `[John]`, `[gen1]` и `[Jack]`. Если при создании экземпляра не указывается имя экземпляра, то система задает имя автоматически (имеющее примерно такой формат, как `[gen1]`, приведенный в данном примере). Как показывают эти вызовы команды `make-instance`, для данной команды не имеет значения, заключено ли имя экземпляра в квадратные скобки, `[]`, или нет, т.е. имя экземпляра может быть указано и как `[John]`, и как `Jack`. **Команда instances**, показанная в этом примере, аналогична команде `facts`, если не считать того, что при ее использовании отображается список экземпляров. В данном случае вместе с экземплярами значения слотов не отображаются, но в следующем разделе показано, как это можно сделать. Полный синтаксис команды `instances` приведен ниже.

```
(instances [<module-name> [<class-name> [inherit]]])
```

Так же как и факты, и соответствующие им конструкции `deftemplate`, экземпляры принадлежат к модулю, в котором определены соответствующие им конструкции `defclass` (см. раздел 11.16). Если задается необязательный параметр с именем модуля, то с помощью команды `instances` создается только список экземпляров, содержащихся в указанном модуле. Если вместо имени модуля задается символ `*`, то в список включаются все экземпляры. А если также задается необязательное имя класса, то перечисляются только экземпляры, принадлежащие к этому классу. Наконец, если указывается также необязательное ключевое слово `inherit`, то перечисляются и все экземпляры, принадлежащие подклассам класса с заданным именем (см. раздел 11.6).

11.4 Обработчики сообщений, определяемые системой

За классами можно закрепить не только данные, но и процедурную информацию. Процедуры, входящие в состав классов, называются *обработчиками сообщений*. Для каждого класса, кроме обработчиков сообщений, определяемых пользователем, автоматически создается также целый ряд обработчиков сообщений, определяемых системой. Эти обработчики сообщений можно вызывать для работы с некоторым экземпляром с помощью **команды send**. Команда `send` имеет следующий синтаксис:

```
(send <object-expression>
<message-name-expression> <expression>*)
```

Например, **сообщение print** отображает информацию о слотах экземпляра:

```
CLIPS> (send [John] print)↵
[John] of PERSON
(full-name "John Q. Public")
(age 24)
(eye-color blue)
(hair-color black)
CLIPS>
```

Для каждого слота, определяемого в конструкции `defclass`, система CLIPS автоматически определяет обработчики сообщений слота с префиксами **get-** и **put-**, которые используются для выборки и задания значений слота. Действительные имена обработчиков сообщений формируются в результате добавления к этим префиксам имени слота. Поэтому, например, конструкция `defclass` с именем `PERSON`, имеющая слоты `full-name`, `age`, `eye-color` и `hair-color`, автоматически создается для данного класса с восемью обработчиками сообщений, имеющими имена `get-full-name`, `put-full-name`, `get-age`, `put-age`, `get-eye-color`, `put-eye-color`, `get-hair-color` и `put-hair-color`. Обработчики сообщений `get-` не имеют параметров и возвращают значение слота, например, как показано ниже.

```
CLIPS> (send [John] get-full-name)↵
"John Q. Public"
CLIPS> (send [John] get-age)↵
24
CLIPS>
```

Обработчики сообщений `put-` принимают от нуля и больше параметров. Если параметры не задаются, то восстанавливается первоначальное, предусмотренное по умолчанию значение слота, а при передаче одного или большего количества параметров значение слота устанавливается с учетом этих параметров. Попытка поместить больше одного значения в однозначный слот приводит к возникновению ошибки. Обработчик сообщений `put-` возвращает значение, представляющее собой новое значение слота, например, как показано в следующем диалоге:

```
CLIPS> (send [Jack] get-age)↵
nil
CLIPS> (send [Jack] put-age 22)↵
22
CLIPS> (send [Jack] get-age)↵
22
CLIPS> (send [Jack] put-age)↵
nil
```

```
CLIPS> (send [Jack] get-age)↵
nil
CLIPS>
```

Команда `watch` принимает в качестве параметров несколько элементов, подлежащих отслеживанию, которые относятся к данному экземпляру. Одним из таких элементов является **slots** (слоты). Если осуществляется отслеживание слотов, то при каждом изменении значения любого слота экземпляра выводится информационное сообщение. Отслеживание изменений в слотах можно отменить с помощью **команды unwatch**:

```
CLIPS> (watch slots)↵
CLIPS> (send [Jack] put-age 24)↵
::= local slot age in instance Jack <- 24
24
CLIPS> (unwatch slots)↵
CLIPS> (send [Jack] put-age 22)↵
22
CLIPS>
```

Еще одним заранее определенным обработчиком сообщений является **delete**. Как и можно было бы предположить, обработчик сообщений `delete` используется для удаления экземпляра. Он возвращает символ `TRUE`, если экземпляр был успешно удален, в противном случае — символ `FALSE`:

```
CLIPS> (instances)↵
[John] of PERSON
[gen1] of PERSON
[Jack] of PERSON
For a total of 3 instances.
CLIPS> (send [gen1] delete)↵
TRUE
CLIPS> (instances)↵
[John] of PERSON
[Jack] of PERSON
For a total of 2 instances.
CLIPS>
```

Еще одним отслеживаемым элементом является **instances** (экземпляры). Если отслеживаются экземпляры, то система CLIPS автоматически выводит сообщение каждый раз, когда создается или удаляется экземпляр. В отличие от того, какие действия выполняются при модификации значения слота факта, при модификации значения слота экземпляра не создается новый экземпляр с изменившимся значением и не удаляется первоначальный экземпляр, поэтому для наблюдения

за изменениями значений слотов экземпляров необходимо использовать отслеживаемый элемент slots. Применение отслеживаемого элемента instances иллюстрируется в следующем примере диалогового выполнения команд:

```
CLIPS> (watch instances)↵
CLIPS> (make-instance Jill of PERSON)↵
==> instance [Jill] of PERSON
[Jill]
CLIPS> (send [Jill] put-age 22)↵
22
CLIPS> (send [Jill] delete)↵
<== instance [Jill] of PERSON
TRUE
CLIPS> (unwatch instances)↵
CLIPS>
```

Последовательность знаков <== указывает, что экземпляр удаляется, а последовательность знаков ==> свидетельствует о том, что экземпляр создается.

11.5 Конструкция definstances

Конструкцией, эквивалентной deffacts, но применяемой к экземплярам, является **конструкция definstances**. После выдачи команды reset всем экземплярам передается сообщение delete, а затем создаются все экземпляры, обнаруженные в конструкциях definstances. Конструкция definstances имеет следующий общий формат:

```
(definstances <definstances name> [active]
   [<optional comment>]
   <instance-definition>*)
```

В этом определении терм <instance-definition> имеет такую форму:

```
([<instance-name-expression>] of
   <class-name-expression>
   <slot-override>*)
```

Необязательное **ключевое слово active** в конструкции definstances используется для указания на то, что сопоставление с шаблонами должно осуществляться по мере обработки информации о перекрытии слотов в процессе создания экземпляра. По умолчанию для экземпляров конструкции definstances сопоставление с шаблонами не происходит до тех пор, пока не будет обработана вся информация о перекрытии слотов. Пример применения конструкции definstances приведен ниже.

```
(definstances people
   (Jack of PERSON (full-name "Jack Q. Public")
              (age 23))
   (of PERSON (full-name "John Doe")
              (hair-color black)))
```

В языке CLIPS предусмотрено несколько команд для манипулирования конструкциями definstances. Для отображения текущего списка конструкций definstances, сопровождаемых в системе CLIPS, используется **команда list-definstances**. **Команда ppdefinstances** (сокращение от pretty print definstances — структурированный вывод конструкции definstances) служит для отображения текстового представления конструкции definstances. **Команда undefinstances** предназначена для удаления конструкций definstances. **Функция get-definstances-list** возвращает многозначное значение, содержащее список конструкций definstances. Эти команды имеют следующий синтаксис:

```
(list-definstances [<module-name>])
(ppdefinstances <definstances-name>)
(undefrule <defrule-name>)
(get-definstances-list [<module-name>])
```

11.6 Классы и наследование

Одно из преимуществ использования языка COOL состоит в том, что классы могут наследовать информацию от других классов, что позволяет обеспечить совместный доступ к информации. Рассмотрим, какие действия пришлось бы предпринимать при наличии конструкции deftemplate, которая представляет информацию о людях:

```
(deftemplate person "Person deftemplate"
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color))
```

В таком случае, если бы потребовалось представить дополнительную информацию, относящуюся к тому, кто является служащим компании или студентом университета, пришлось бы предпринять определенные усилия. Один из возможных подходов мог бы предусматривать дополнение конструкции deftemplate с именем person для включения другой необходимой информации:

```
(deftemplate person "Person deftemplate"
   (slot full-name)
   (slot age)
```

```
   (slot eye-color)
   (slot hair-color)
   (slot job-position)
   (slot employer)
   (slot salary)
   (slot university)
   (slot major)
   (slot GPA))
```

Но ко всем людям относились бы только четыре слота этой конструкции `deftemplate`: `full-name`, `age`, `eye-color` и `hair-color`. С другой стороны, слоты `job-position`, `employer` и `salary` относились бы только к служащим, а слоты `university`, `major` и `GPA` — только к студентам. По мере добавления информации о людях, занимающихся другой деятельностью, приходилось бы вводить все больше и больше слотов в конструкцию `deftemplate` с именем `person`, причем по большей части эти слоты оказались бы неприменимыми для всех людей.

Еще один подход мог бы состоять в создании отдельных конструкций `deftemplate` для служащих и студентов, как в следующем примере:

```
(deftemplate employee "Employee deftemplate"
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color)
   (slot job-position)
   (slot employer)
   (slot salary))
(deftemplate student "Student deftemplate"
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color)
   (slot university)
   (slot major)
   (slot GPA))
```

При использовании такого подхода каждая конструкция `deftemplate` содержит только необходимую информацию, но приходится дублировать некоторые из слотов. Если бы пришлось модифицировать атрибуты одного из таких дублирующихся слотов, то потребовалось бы вносить изменения во многих местах, чтобы обеспечить единообразие представления информации. Кроме того, если бы нужно было написать правило, позволяющее отыскивать всех людей с синими глазами,

то пришлось бы использовать два шаблона вместо одного (а если потребовалось бы также включить факты person, количество шаблонов стало бы равным трем), как показано ниже.

```
(defrule find-blue-eyes
   (or (employee (full-name ?name)
                 (eye-color blue))
       (student (full-name ?name)
                 (eye-color blue)))
   =>
   (printout t ?full-name "has blue eyes." crlf))
```

Классы позволяют совместно использовать общую информацию, принадлежащую к различным категориям, без дублирования, или включения ненужной информации. Вернемся к первоначально рассматриваемому определению конструкции defclass с именем PERSON:

```
(defclass PERSON "Person defclass"
   (is-a USER)
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color))
```

Чтобы определить новые классы, которые расширяют определение класса PERSON, достаточно указать имя класса PERSON в атрибуте is-a нового класса, как показано ниже.

```
(defclass EMPLOYEE "Employee defclass"
   (is-a PERSON)
   (slot job-position)
   (slot employer)
   (slot salary))
(defclass STUDENT "Student defclass"
   (is-a PERSON)
   (slot university)
   (slot major)
   (slot GPA))
```

Атрибуты класса PERSON наследуются и в классе EMPLOYEE, и в классе STUDENT. Примеры создания экземпляров для каждого из этих трех классов иллюстрирует следующий диалог:

```
CLIPS> (make-instance [John] of PERSON)↵
[John]
CLIPS> (make-instance [Jack] of EMPLOYEE)↵
```

```
[Jack]
CLIPS> (make-instance [Jill] of STUDENT)↵
[Jill]
CLIPS> (send [John] print)↵
[John] of PERSON
(full-name nil)
(age nil)
(eye-color nil)
(hair-color nil)
CLIPS> (send [Jack] print)↵
[Jack] of EMPLOYEE
(full-name nil)
(age nil)
(eye-color nil)
(hair-color nil)
(job-position nil)
(employer nil)
(salary nil)
CLIPS> (send [Jill] print)↵
[Jill] of STUDENT
(full-name nil)
(age nil)
(eye-color nil)
(hair-color nil)
(university nil)
(major nil)
(GPA nil)
CLIPS>
```

Обратите внимание на то, что каждый экземпляр содержит только слоты, относящиеся к его классу. Как показано в следующем подразделе, в любом классе можно переопределить любой слот, который был уже определен в любом из его суперклассов.

Класс, который либо прямо, либо косвенно наследует свойство другого класса, называется **подклассом** того класса, от которого он наследует свойства. Класс, от которого наследуются свойства, называется **суперклассом** наследующего класса. Классы PERSON, EMPLOYEE и STUDENT представляют собой подклассы класса USER. Классы EMPLOYEE и STUDENT являются подклассами класса PERSON. Класс USER — суперкласс классов PERSON, EMPLOYEE и STUDENT, а класс PERSON — суперкласс классов EMPLOYEE и STUDENT. Иерархией классов с единичным наследованием называется такая иерархия, в которой каждый класс имеет

только один суперкласс, связанный с ним прямыми отношениями наследования. Иерархией классов с множественным наследованием называется такая иерархия, в которой любой класс может иметь несколько суперклассов, связанных с ним прямыми отношениями наследования. В языке COOL поддерживается множественное наследование, но до раздела 11.15, в котором множественное наследование рассматривается более подробно, мы будет ограничиваться применением примеров единичного наследования. Ниже приведен пример класса, в котором используется множественное наследование (в нем рассматривается студент, который имеет работу).

```
(defclass WORKING-STUDENT
"Working Student defclass"
(is-a STUDENT EMPLOYEE))
```

Разрешение конфликтов между определениями слотов

По умолчанию, если какой-то слот переопределяется в подклассе, то атрибуты слота из нового определения используются исключительно в экземплярах этого класса. Например, предположим, что определены следующие классы:

```
(defclass A
   (is-a USER)
   (slot x (default 3))
   (slot y)
   (slot z (default 4)))
(defclass B
   (is-a A)
   (slot x)
   (slot y (default 5))
   (slot z (default 6)))
```

В таком случае создание экземпляров классов `A` и `B` приведет к получению следующих результатов:

```
CLIPS> (make-instance [a] of A)↵
[a]
CLIPS> (make-instance [b] of B)↵
[b]
CLIPS> (send [a] print)↵
[a] of A
(x 3)
(y nil)
(z 4)
CLIPS> (send [b] print)↵
```

```
[b] of B
(x nil)
(y 5)
(z 6)
CLIPS>
```

Обратите внимание на то, что слоту `x` экземпляра `b` по умолчанию присвоено значение `nil` вместо `3`. Это связано с тем, что при отсутствии заданного по умолчанию значения для слота `x` класса `B` полностью перекрывается заданное по умолчанию значение `3`, присваиваемое слоту `x` в классе `A`. Чтобы обеспечить возможность наследовать атрибуты слота от суперклассов, можно воспользоваться атрибутом слота **source**. Если этому атрибуту присваивается значение **exclusive**, которое применяется по умолчанию, то атрибуты для слота устанавливаются на основе наиболее конкретного класса, определяющего этот слот. В иерархии единичного наследования как таковой рассматривается класс, имеющий наименьшее количество суперклассов. Если же атрибуту `source` присваивается значение **composite**, то атрибуты, которые не определены явно в наиболее конкретном классе, определяющем слот, берутся из следующего по порядку наиболее конкретного класса, в котором определяется данный атрибут. Например, если описанные ранее конструкции `defclass` с именами `A` и `B` будут объявлены следующим образом:

```
(defclass A
   (is-a USER)
   (slot x (default 3))
   (slot y)
   (slot z (default 4)))
(defclass B
   (is-a A)
   (slot x (source composite))
   (slot y (default 5))
   (slot z (default 6)))
```

то после создания экземпляров классов `A` и `B` будут получены такие результаты:

```
CLIPS> (make-instance [a] of A)↵
[a]
CLIPS> (make-instance [b] of B)↵
[b]
CLIPS> (send [a] print)↵
[a] of A
(x 3)
(y nil)
(z 4)
CLIPS> (send [b] print)↵
```

```
[b] of B
(x 3)
(y 5)
(z 6)
CLIPS>
```

Теперь, после того как слот `x` класса `B` объявлен с атрибутом `source`, которому присвоено значение `composite`, этот слот может наследовать заданный по умолчанию атрибут от класса `A`, и применяемое по умолчанию результирующее значение для слота `x` экземпляра `b` становится равным `3`.

Возможно также запретить наследование значения слота с использованием атрибута слота **propagation**. Если этому атрибуту присваивается значение **inherit**, которое является заданным по умолчанию, то данный слот наследуется подклассами. А если этому атрибуту присваивается значение **no-inherit**, то слот подклассами не наследуется. Например, если классы `A` и `B` будет определены следующим образом:

```
(defclass A
   (is-a USER)
   (slot x (propagation no-inherit))
   (slot y))
(defclass B
   (is-a A)
   (slot z))
```

то после создания экземпляров классов `A` и `B` будут получены такие результаты:

```
CLIPS> (make-instance [a] of A)↵
[a]
CLIPS> (make-instance [b] of B)↵
[b]
CLIPS> (send [a] print)↵
[a] of A
(x
nil)
(y nil)
CLIPS> (send [b] print)↵
[b] of B
(y nil)
(z nil)
CLIPS>
```

Экземпляр b класса B наследует слот y из класса A, но не наследует слот x из класса A, поскольку атрибут propagation последнего имеет значение no-inherit.

Абстрактные и конкретные классы

В языке CLIPS предусмотрена возможность определять классы, используемые только для наследования. Такие классы называются **абстрактными классами**. Создание экземпляров абстрактных классов невозможно. По умолчанию классы являются **конкретными**. Для указания на то, должен ли класс быть абстрактным (abstract) или конкретным (concrete), применяется атрибут класса **role**. Атрибут класса role должен быть указан после атрибута класса is-a, но перед любыми определениями слотов, например, как показано ниже.

```
(defclass ANIMAL
   (is-a USER)
   (role abstract))
(defclass MAMMAL
   (is-a ANIMAL)
   (role abstract))
(defclass CAT
   (is-a MAMMAL)
   (role concrete))
(defclass DOG
   (is-a MAMMAL)
   (role concrete))
```

Классы ANIMAL и MAMMAL являются абстрактными, а классы CAT и DOG — конкретными. Атрибут role наследуется, поэтому, хотя и не требуется объявлять класс MAMMAL как абстрактный, поскольку он наследует этот атрибут от класса ANIMAL, необходимо объявить классы CAT и DOG как конкретные, в связи с тем, что в противном случае они будут рассматриваться как абстрактные. Попытка создать экземпляр абстрактного класса приводит к формированию сообщения об ошибке, как в следующем примере:

```
CLIPS> (make-instance [animal-1] of ANIMAL)↵
[INSMNGR3] Cannot create instances of abstract
class ANIMAL.
CLIPS> (make-instance [cat-1] of CAT)↵
[cat-1]
CLIPS>
```

Настоятельная необходимость объявлять какой-либо класс как абстрактный не возникает, но при использовании такого подхода в соответствующих условиях

код становится более удобным для сопровождения и проще обеспечивает повторное использование. При этом достаточно лишь исключить для пользователя возможность создавать экземпляры с помощью какого-то класса, если класс не предназначен для этой цели. Но если данный класс уже используется таким образом, то в будущих реализациях станет невозможным его исключение, поскольку это приведет к нарушению работы существующего кода.

В рассматриваемом примере ответ на вопрос о том, должны ли классы `ANIMAL` и `MAMMAL` быть абстрактными, не так уж однозначен. Если требуется создать картотеку с информацией о животных, содержащихся в некотором зоопарке, то данные классы, по-видимому, должны быть абстрактными, поскольку в природе не существует животных (в данном случае речь идет о млекопитающих), которые соответствовали бы только этому определению и не относились бы к какому-то более конкретному виду живых существ. Но если бы предпринималась попытка идентификации какого-то животного, то вполне могла бы возникнуть необходимость создавать экземпляры класса `ANIMAL` или `MAMMAL`, например, для включения в них информации о том, что́ мы смогли выяснить в отношении данного животного.

Команды, относящиеся к конструкции `defclass`

Для манипулирования конструкциями `defclass` предусмотрено несколько команд. **Команда list-defclasses** используется для отображения текущего списка конструкций `defclass`, поддерживаемых системой CLIPS. Эта команда имеет следующий синтаксис:

```
(list-defclasses [<module-name>])
```

Целесообразно ознакомиться с таким выводом этой функции:

```
CLIPS> (list-defclasses)↵
FLOAT
INTEGER
SYMBOL
STRING
MULTIFIELD
EXTERNAL-ADDRESS
FACT-ADDRESS
INSTANCE-ADDRESS
INSTANCE-NAME
OBJECT
PRIMITIVE
NUMBER
LEXEME
```

```
ADDRESS
INSTANCE
USER
INITIAL-OBJECT
PERSON
EMPLOYEE
STUDENT
For a total of 20 defclasses.
CLIPS>
```

Как показывают эти результаты, в языке CLIPS имеется целый ряд заранее определенных примитивных классов, в том числе OBJECT, PRIMITIVE, NUMBER, LEXEME, FLOAT, INTEGER, SYMBOL, STRING, MULTIFIELD, ADDRESS, INSTANCE, EXTERNAL-ADDRESS, FACT-ADDRESS, INSTANCE-ADDRESS и INSTANCE-NAME. Эти классы нельзя использовать для создания других классов. Примитивные классы в основном применяются в сочетании с универсальными функциями, которые рассматривались в главе 10. В приведенном выше списке не указаны заранее определенные классы USER и INITIAL-OBJECT. Класс USER является основой для создания новых классов, а класс INITIAL-OBJECT является подклассом класса USER и используется для создания экземпляра initial-object (см. раздел 11.7). Список дополняет созданные нами классы (PERSON, EMPLOYEE и STUDENT). Иерархия заранее определенных классов показана на рис. 11.1.

Для отображения отношений наследования между классом и его подклассами предназначена **команда browse-classes**. Она имеет следующий синтаксис:

```
(browse-classes [<class-name>])
```

Если в вызове команды browse-classes не задается имя класса, то отображаются отношения наследования для корневого класса OBJECT. Например, после вызова следующей команды отображается информация, содержащаяся на рис. 11.1, наряду со сведениями о классе PERSON и его подклассах, которые используются в качестве примеров:

```
CLIPS> (browse-classes)↵
OBJECT
   PRIMITIVE
      NUMBER
         INTEGER
         FLOAT
      LEXEME
         SYMBOL
         STRING
      MULTIFIELD
```

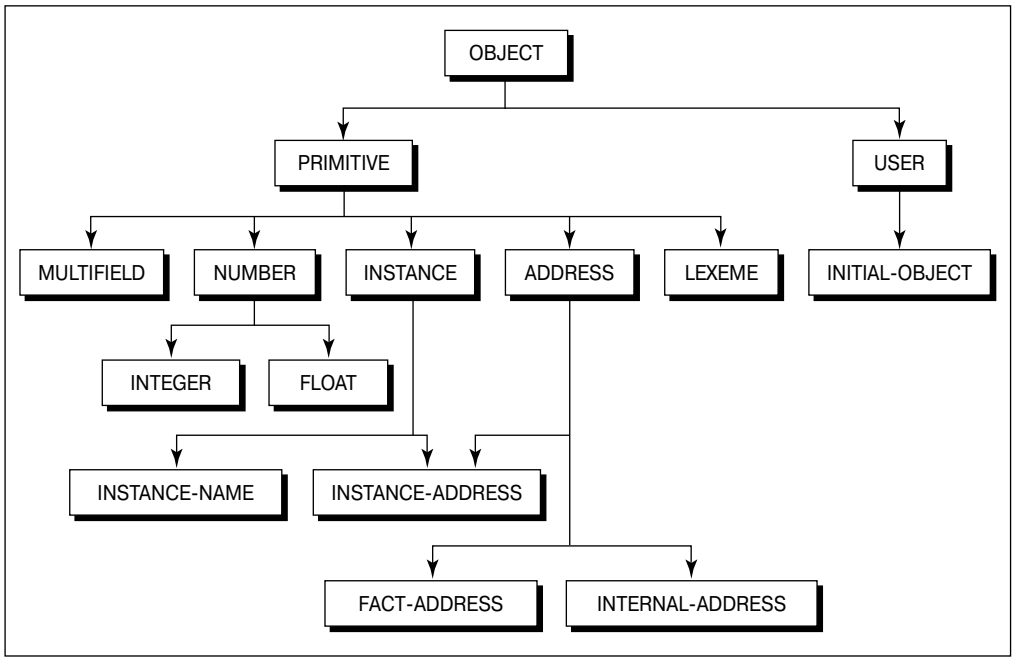


Рис. 11.1. Краткий обзор заранее определенных классов

```
      ADDRESS
         EXTERNAL-ADDRESS
         FACT-ADDRESS
         INSTANCE-ADDRESS *
      INSTANCE
         INSTANCE-ADDRESS *
         INSTANCE-NAME
   USER
      INITIAL-OBJECT
      PERSON
         EMPLOYEE
         STUDENT
CLIPS>
```

Отступы применяются для указания на то, что некоторый класс является подклассом первого предшествующего ему класса, обозначенного меньшим отступом. Например, все классы `NUMBER`, `LEXEME`, `MULTIFIELD`, `ADDRESS` и `INSTANCE` являются подклассами класса `PRIMITIVE`. Звездочка перед именем класса означает, что этот подкласс связан прямыми отношениями наследования

больше чем с одним классом. Например, подкласс INSTANCE-ADDRESS связан прямыми отношениями наследования с классами ADDRESS и INSTANCE.

Если в команде browse-classes указано имя класса, то отображается только информация о наследовании, относящаяся к указанному классу и его подклассам:

```
CLIPS> (browse-classes PERSON)↵
PERSON
   EMPLOYEE
   STUDENT
CLIPS>
```

Для отображения текстового представления конструкции defclass используется **команда ppdefclass** (сокращение от pretty print defclass — структурированный вывод конструкции defclass). А **команда undefclass** служит для удаления конструкции defclass. Эти команды имеют следующий синтаксис:

```
(ppdefclass <defclass-name>)
(undefclass <defclass-name>)
```

Удаление конструкции defclass остается невозможным до тех пор, пока существуют экземпляры класса, определенные с ее помощью. Прежде чем появится возможность удалить класс, необходимо удалить все экземпляры, принадлежащие к этому классу, и все подклассы этого класса, например, как показано ниже.

```
CLIPS> (undefclass STUDENT)↵
[PRNTUTIL4] Unable to delete defclass STUDENT.
CLIPS> (send [Jill] delete)↵
TRUE
CLIPS> (undefclass STUDENT)↵
CLIPS>
```

11.7 Сопоставление с шаблоном объекта

Шаблоны объектов предоставляют некоторые возможности, недоступные при использовании шаблонов конструкций deftemplate или шаблонов упорядоченных фактов. Во-первых, для согласования с экземплярами нескольких классов может использоваться единственный шаблон объекта. Во-вторых, изменения в значениях слотов, не заданных в шаблоне объекта, не приводят к повторной активизации правила, к которому принадлежит данный шаблон. В-третьих, изменения в значениях слотов, не заданные в шаблоне объекта в пределах условного элемента logical, не приводят к удалению логического обоснования, предусмотренного связанным с ним правилом. Шаблон объекта имеет следующую общую форму:

```
(object <attribute-constraint>*)
```

В этом определении терм <attribute-constraint> имеет такой общий формат:

```
(is-a <constraint>) |
(name <constraint>) |
(<slot-name> <constraint>*)
```

а терм <constraint> аналогичен ограничениям слота шаблона, которые использовались в шаблонах конструкции deftemplate. Назначение ключевых слов is-a и name описано ниже. Вначале рассмотрим следующий простой пример сопоставления с шаблоном объекта:

```
CLIPS> (clear)↵
CLIPS>
(defclass 1D-POINT
   (is-a USER)
   (slot x))↵
CLIPS>
(defrule Example-1
   (object (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (make-instance p1 of 1D-POINT (x 3))↵
[p1]
CLIPS> (agenda)↵
0        Example-1: [p1]
For a total of 1 activation.
CLIPS> (run)↵
Value of x slot is 3
CLIPS>
```

Прежде всего определен класс 1D-POINT с единственным атрибутом x. Затем определена конструкция defrule с именем Example-1. Обратите внимание на то, что в этом правиле предусмотрено только согласование с учетом значения слота x и нет никакого упоминания о классе 1D-POINT. При создании этого правила система CLIPS автоматически определяет, что класс 1D-POINT способен к согласованию с этим шаблоном. А после создания экземпляра класса 1D-POINT должным образом активизируется правило Example-1 и во время прогона программы обеспечивается правильный вывод значения слота x данного экземпляра.

Рассмотрим, что произойдет, если будет определен еще один класс, содержащий слот x:

```
CLIPS>
(defclass 2D-POINT
```

```
   (is-a USER)
   (slot x)
   (slot y))↵
CLIPS> (make-instance p2 of 2D-POINT (x 4) (y 2))↵
[p2]
CLIPS> (agenda)↵
CLIPS>
```

Возможно, такое развитие событий покажется неожиданным, но правило `Example-1` не активизируется экземпляром `[p2]`. Это связано с тем, что определение возможности согласования класса с шаблоном объекта происходит во время обработки системой CLIPS объявления правила. Экземпляры классов, определяемые после определения правила, не согласуются с шаблонами объекта, используемыми в правиле. А если правило `Example-1` будет переопределено, то обнаружится поведение, на которое мы рассчитывали первоначально:

```
CLIPS>
(defrule Example-1
   (object (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (agenda)↵
0        Example-1: [p2]
0        Example-1: [p1]
For a total of 2 activations.
CLIPS> (run)↵
Value of x slot is 4
Value of x slot is 3
CLIPS>
```

Обратите внимание на то, что теперь правило `Example-1` активизируется применительно и к экземплярам `[p1]`, и к экземплярам `[p2]`, содержащим слоты `x` из разных классов.

Сопоставление с шаблоном объекта и наследование

Для сопоставления с шаблонами объекта могут также применяться слоты, унаследованные от других классов, как показано ниже.

```
CLIPS> (clear)↵
CLIPS>
(defclass 1D-POINT
   (is-a USER)
   (slot x))↵
```

```
CLIPS>
(defclass 2D-POINT
   (is-a 1D-POINT)
   (slot y))↵
CLIPS>
(defclass 3D-POINT
   (is-a 2D-POINT)
   (slot z))↵
CLIPS>
(defrule Example-2
   (object (y ?y))
   =>
   (printout t "Value of y slot is " ?y crlf))↵
CLIPS> (make-instance p1 of 1D-POINT (x 3))↵
[p1]
CLIPS> (make-instance p2 of 2D-POINT (x 1) (y 2))↵
[p2]
CLIPS> (make-instance p3 of 3D-POINT (x 2) (y 4) (z 3))↵
[p3]
CLIPS> (agenda)↵
0        Example-2: [p3]
0        Example-2: [p2]
For a total of 2 activations.
CLIPS> (run)↵
Value of y slot is 4
Value of y slot is 2
CLIPS>
```

Обратите внимание на то, что правило `Example-2` активизируется под действием экземпляров `[p2]` и `[p3]`, но не `[p1]`. Экземпляр `[p2]` является экземпляром класса `2D-POINT`, в котором определен слот `y`, на который есть ссылка в этом правиле. Экземпляр `[p3]` является элементом класса `3D-POINT`, который наследует слот `y` от класса `2D-POINT`. Экземпляр `[p1]` класса `1D-POINT` включает только слот `x`, поэтому не обеспечивает согласование с шаблоном объекта из правила `Example-2`.

Ключевые слова `is-a` и `name`

Ключевое слово is-a при его использовании в качестве имени слота имеет специальное значение, если оно включено в шаблон объекта. Это ключевое слово ограничивает перечень экземпляров, согласующихся с шаблоном, экземплярами

тех классов, которые удовлетворяют ограничению is-a, например, как показано ниже.

```
CLIPS>
(defrule Example-3
   (object (is-a 2D-POINT) (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (agenda)↵
0        Example-3: [p3]
0        Example-3: [p2]
For a total of 2 activations.
CLIPS> (run)↵
Value of x slot is 2
Value of x slot is 1
CLIPS>
```

Обратите внимание на то, что правило Example-3 активизируют только экземпляры [p2] и [p3], даже несмотря на то, что экземпляр [p1] также содержит атрибут слота x. Это связано с тем, что экземпляр [p1] не является экземпляром класса 2D-POINT или экземпляром класса, который наследует свои свойства от класса 2D-POINT. Но экземпляр [p3] удовлетворяет шаблону объекта, поскольку является элементом класса 2D-POINT в силу наследования. Тем не менее возможность согласования экземпляра [p3] с шаблоном объекта можно исключить, явно запретив использование класса 3D-POINT:

```
CLIPS>
(defrule example-4
   (object (is-a 2D-POINT&~3D-POINT) (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (agenda)↵
0        example-4: [p2]
For a total of 1 activation.
CLIPS> (run)↵
Value of x slot is 1
CLIPS>
```

Любой класс, на который имеется ссылка в шаблоне объекта, должен быть уже определен, в противном случае возникает ошибка:

```
CLIPS>
(defrule example-5
   (object (is-a 4D-POINT) (x ?x))
```

```
   =>
   (printout t "Value of x slot is " ?x crlf))↵
[OBJRTBLD5] Undefined class in object pattern.
ERROR:
(defrule MAIN::example-5
   (object (is-a 4D-POINT)
CLIPS>
```

То же утверждение справедливо по отношению к слотам, на которые имеется ссылка в шаблоне объекта. В программе должен быть представлен по меньшей мере один класс, который содержит все атрибуты слотов, упомянутые в шаблоне объекта, так как в противном случае возникает ошибка:

```
CLIPS>
(defrule example-6
   (object (w ?w))
   =>
   (printout t "Value of w slot is " ?w crlf))↵
[OBJRTBLD2] No objects of existing classes can
satisfy w restriction in object pattern.
ERROR:
(defrule MAIN::example-6
   (object (w
CLIPS>
```

В данном случае ни в одном из существующих классов (`1D-POINT`, `2D-POINT` и `3D-POINT`) нет атрибута слота `w`, поэтому шаблон объекта в правиле `Example-6` не может быть удовлетворен, и вырабатывается сообщение об ошибке.

Для согласования с конкретными экземплярами может применяться **ключевое слово name**, например, следующим образом:

```
CLIPS>
(defrule example-7
   (object (name [p1] | [p3]) (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (agenda)↵
0        example-7: [p3]
0        example-7: [p1]
For a total of 2 activations.
CLIPS> (run)↵
Value of x slot is 2
Value of x slot is 3
CLIPS>
```

В правиле Example-7 используется ключевое слово name для ограничения перечня экземпляров, которые могут быть согласованы с шаблоном объекта, экземплярами [p1] и [p3]. Экземпляр [p2], который содержит значение слота x, не будет согласован с шаблоном, поскольку его имя не соответствует ограничению name. Ключевые слова is-a и name имеют особый смысл в шаблонах объектов, поэтому не могут применяться в качестве имен слотов в определении конструкции defclass.

Активизация шаблонов объектов

Одним очень важным различием между шаблонами объектов и шаблонами фактов является то, что при изменении значения слота экземпляра соответствующее влияние испытывают только те шаблоны объектов, которые явно согласуются с одним из слотов. Для иллюстрации сказанного еще раз рассмотрим первый вариант правила sum-rectangles, приведенный в разделе 8.10:

```
(defrule sum-rectangles
   (rectangle (height ?height) (width ?width))
   ?sum <- (sum ?total)
   =>
   (retract ?sum)
   (assert (sum (+ ?total (* ?height ?width)))))
```

Формулировка этого правила содержит ошибку, связанную с тем, что любая попытка модифицировать факт sum повторно активизирует правило под действием уже обработанного факта rectangle, что приводит к возникновению бесконечного цикла. Этот недостаток не удается устранить путем преобразования факта sum из упорядоченного факта в факт, заданный с помощью конструкции deftemplate, и использования команды modify. В решении этой проблемы, предложенном в разделе 8.4, для вычисления суммы применяется несколько правил. Но лучшее решение состоит в использовании шаблонов объектов соответствующим образом. Ниже приведен результат первой попытки переформулировать правило sum-rectangles из раздела 8.4 в целях применения объектов.

```
(defclass RECTANGLE
   (is-a USER)
   (slot height)
   (slot width))
(defclass SUM
   (is-a USER)
   (slot total))
(definstances initial-information
   (of RECTANGLE (height 10) (width 6))
```

```
   (of RECTANGLE (height 7) (width 5))
   (of RECTANGLE (height 6) (width 8))
   (of RECTANGLE (height 2) (width 5))
   ([sum] of SUM (total 0)))
(defrule sum-rectangles
(object (is-a RECTANGLE)
      (height ?height) (width ?width))
?sum <- (object (is-a SUM) (total ?total))
 =>
(send ?sum put-total
   (+ ?total (* ?height ?width))))
```

В этом новом правиле `sum-rectangles` заслуживает внимания несколько особенностей. Прежде всего, как и при использовании шаблонов фактов, можно связать с переменной экземпляр, согласующийся с шаблоном, применяя оператор связывания с шаблоном `<-`. В данном случае с переменной `?sum` связывается адрес экземпляра, согласующегося с шаблоном объекта `SUM`. Затем эта переменная может использоваться в качестве параметра в функции `send` для передачи сообщений экземплярам. Но в ходе прогона программы этот код демонстрирует возникновение той же проблемы, которая наблюдалась при использовании шаблонов фактов, — бесконечный цикл. Это связано с тем, что изменение значения слота `total` из правой части правила с помощью сообщения `put-total` приводит к повторной активизации шаблона объекта `SUM`, поскольку этот шаблон согласуется со слотом `total`.

Но согласовывать слот `total` в левой части правила не требуется, поскольку ссылка на значение, связанное с переменной `?total`, больше не применяется в каком-либо из шаблонов в левой части. Попытка составить правило, в котором выборка значения осуществляется путем передачи сообщения в правой части правила, приводит к получению следующего правила:

```
(defrule sum-rectangles
   (object (is-a RECTANGLE)
   (height ?height) (width ?width))
   ?sum <- (object (is-a SUM))
   =>
   (bind ?total (send ?sum get-total))
   (send ?sum put-total
   (+ ?total (* ?height ?width))))
```

Эта версия правила, в отличие от первоначальной версии, не приводит к возникновению бесконечного цикла. А если бы было известно, что в системе будет всегда присутствовать только один экземпляр класса `SUM`, то можно было бы

пройти еще на шаг дальше по пути упрощения данного правила, как показано ниже.

```
(defrule sum-rectangles
   (object (is-a RECTANGLE)
   (height ?height) (width ?width))
   =>
   (bind ?total (send [sum] get-total))
   (send [sum] put-total
   (+ ?total (* ?height ?width))))
```

Дело в том, что можно просто сослаться на экземпляр по имени в правой части правила, а не применять сопоставление с шаблонами для выборки адреса экземпляра `[sum]` и присваивания переменной `?sum`.

Атрибут сопоставления с шаблонами

Атрибут pattern-match позволяет указать, что либо слот, либо класс не может участвовать в сопоставлении с шаблонами. Если этому атрибуту присваивается значение **reactive**, которое предусмотрено по умолчанию, то указанный слот или класс обладает способностью активизировать сопоставление с шаблонами в левой части правила. Если же этому атрибуту присваивается значение **non-reactive**, то указанный слот или класс не активизирует сопоставление с шаблонами в левой части правила. Например, если бы класс `SUM` был переопределен в целях использования атрибута `pattern-match` и применялась первоначальная версия правила `sum-rectangles` следующим образом:

```
(defclass SUM
   (is-a USER)
   (slot total (pattern-match non-reactive)))
(defrule sum-rectangles
   (object (is-a RECTANGLE)
   (height ?height) (width ?width))
   ?sum <- (object (is-a SUM) (total ?total))
   =>
   (send ?sum put-total
   (+ ?total (* ?height ?width))))
```

то при обработке в системе определения правила `sum-rectangles` возникло бы такое сообщение об ошибке:

```
[OBJRTBLD2] No objects of existing classes can
satisfy total restriction in object pattern.
```

По существу, возможность провести сопоставление с этим шаблоном отсутствует, поскольку условие `is-a` во втором шаблоне ограничивает перечень воз-

можных классов классом SUM, а этот класс не имеет слота total, доступного для сопоставления с шаблонами, поэтому при обработке данного правила вырабатывается ошибка. В программе может быть предусмотрено применение нескольких классов со слотами, имеющими одно и то же имя, причем часть из них может иметь атрибут reactive, а часть — non-reactive. При обработке объявления правила система определяет, какие классы в принципе могут с ним согласовываться, и из рассмотрения исключаются классы, имеющие слоты с атрибутами non-reactive, на которые имеется ссылка в соответствующем шаблоне.

Атрибут pattern-match можно также использовать на уровне класса, например, как показано ниже.

```
(defclass SUM
   (is-a USER)
   (pattern-match non-reactive)
   (slot total))
```

Атрибут класса pattern-match должен быть задан после атрибутов класса is-a и role, но до каких-либо определений слотов. Если класс объявлен с атрибутом non-reactive, то экземпляры этого класса не будут согласовываться с какими-либо шаблонами (независимо от того, какими являются значения атрибутов pattern-match отдельных слотов), но экземпляры подклассов этого класса могут согласовываться с шаблонами, если атрибут класса pattern-match переопределен в них как reactive. Кроме того, атрибут класса pattern-match не влияет явно на атрибуты слотов pattern-match, поэтому класс может наследовать слоты с атрибутом reactive от класса с атрибутом non-reactive.

Шаблоны объектов и условный элемент logical

Шаблоны объектов и средства создания экземпляров могут использоваться с условным элементом logical, так же, как могут использоваться факты и шаблоны фактов. Но если на слот нет ссылки в шаблоне объекта в пределах условного элемента logical, то изменения в слотах экземпляра не влияют на логическое обоснование для факта или экземпляра. Например, рассмотрим следующие конструкции, созданные с использованием шаблонов конструкций deftemplate:

```
(deftemplate emergency
   (slot type)
   (slot location))
(deftemplate response-team
   (slot name)
   (slot location))
(defrule create-response-team
   (logical (emergency (location ?location)))
```

```
   =>
   (assert (response-team (name first-response)
                          (location ?location))))
```

Если после загрузки этих конструкций в программу и внесения факта `emergency` в список фактов начнется прогон программы, то правило `create-response-team` создаст факт `response-team` следующим образом:

```
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS>
(assert (emergency (type unknown)
                   (location building-1))↵
==> f-1     (emergency (type unknown)
                   (location building-1))
<Fact-1>
CLIPS> (run)↵
==> f-2     (response-team (name first-response)
                   (location building-1))
CLIPS>
```

Модификация факта `emergency` вызывает извлечение факта `response-team`, даже несмотря на то, что в правиле `create-response-team` не происходит доступ к слоту `type`:

```
CLIPS> (modify 1 (type fire))↵
<== f-1     (emergency (type unknown)
                       (location building-1))
<== f-2     (response-team (name first-response)
                           (location building-1))
==> f-3     (emergency (type fire)
                       (location building-1))
<Fact-3>
CLIPS>
```

Для воссоздания факта `response-team` необходимо снова запустить правило `create-response-team`:

```
CLIPS> (run)↵
==> f-4     (response-team (name first-response)
                           (location building-1))
CLIPS>
```

Ниже приведена та же программа, в которой используются конструкции `defclass` и объекты. Обратите внимание на то, что в этом варианте удален слот `name`, применяемый в конструкции `deftemplate` с именем `response-team`,

поскольку он имеет особый смысл в шаблонах объектов. Просто в данном случае экземпляру RESPONSE-TEAM присваивается имя, которое должно быть помещено в слот name конструкции deftemplate с именем response-team.

```
(defclass EMERGENCY
   (is-a USER)
   (slot type)
   (slot location))
(defclass RESPONSE-TEAM
   (is-a USER)
   (slot location))
(defrule create-response-team
   (logical (object (is-a EMERGENCY)
               (location ?location)))
   =>
   (make-instance first-response of RESPONSE-TEAM
      (location ?location)))
```

Проводя работу с этими новыми конструкциями, можно убедиться в том, что их первоначальное поведение аналогично тому, которое возникает при использовании фактов:

```
CLIPS> (reset)↵
CLIPS> (watch instances)↵
CLIPS>
(make-instance e1 of EMERGENCY
(type unknown) (location building-1))↵
==> instance [e1] of EMERGENCY
[e1]
CLIPS> (send [e1] print)↵
[e1] of EMERGENCY
(type unknown)
(location building-1)
CLIPS> (run)↵
==> instance [first-response] of RESPONSE-TEAM
CLIPS>
```

В результате прогона программы экземпляр [first-response] создается после экземпляра [e1]. Эти экземпляры аналогичны конструкциям deftemplate с именами emergency и response-team, создаваемыми в примере факта.

Но замена типа экземпляра [e1] с именем unknown типом fire не вызывает удаление и создание другого экземпляра [first-response], как показано ниже.

```
CLIPS> (send [e1] put-type fire)↵
fire
CLIPS> (send [e1] print)↵
[e1] of EMERGENCY
(type fire)
(location building-1)
CLIPS>
```

Безусловно, в данном случае экземпляр [first-response] не удаляется, а для его воссоздания снова выполняется правило create-response-team, поэтому в этом примере применение экземпляров и шаблонов объектов оказывается более эффективным, чем фактов и шаблонов фактов.

Слот location явно согласуется с правилом create-response-team шаблона объекта, поэтому при изменении значения этого слота возникает поведение, аналогичное наблюдаемому при использовании фактов, как в следующем примере:

```
CLIPS> (send [e1] put-location building-2)↵
<== instance [first-response] of RESPONSE-TEAM
building-2
CLIPS> (run)↵
==> instance [first-response] of RESPONSE-TEAM
CLIPS>
```

Шаблон initial-object

Как было описано в разделах 7.11 и 8.15, в некоторых обстоятельствах система CLIPS вводит в левую часть правила шаблон initial-fact. Но при определенных условиях, если в каком-то правиле используется шаблон объекта, система CLIPS вместо шаблона initial-fact применяет шаблон initial-object. Объектными эквивалентами конструкции deftemplate с именем initial-fact и конструкции deffacts с именем initial-fact (см. раздел 7.10) являются конструкции defclass с именем INITIAL-OBJECT и конструкции definstances с именем initial-object:

```
(defclass INITIAL-OBJECT
   (is-a USER))
(definstances initial-object
   (initial-object of INITIAL-OBJECT))
```

Если в некоторой позиции в правило необходимо ввести шаблон initial-fact и (или) initial-object, то в случае, если шаблоном, предшествующим позиции вставки, является шаблон факта, добавляется шаблон initial-fact, в противном случае, если шаблоном, предшествующим точке вставки, являет-

ся шаблон объекта, то вставляется шаблон `initial-object`. Если же точке вставки не предшествует ни один шаблон, то для определения типа добавляемого шаблона используется шаблон, который следует за точкой вставки. Для шаблона `initial-object` применяется следующий формат:

```
(object (is-a INITIAL-OBJECT)
   (name [initial-object]))
```

С учетом того, что для добавления шаблона используются эти новые правила, следующую конструкцию `defrule`:

```
(defrule no-emergencies
   (not (object (is-a EMERGENCY)))
   =>
   (printout t "No emergencies" crlf))
```

можно преобразовать в такую конструкцию:

```
(defrule no-emergencies
   (object (is-a INITIAL-OBJECT)
   (name [initial-object]))
   (not (object (is-a EMERGENCY)))
   =>
   (printout t "No emergencies" crlf))
```

поскольку отсутствует шаблон, предшествующий условному элементу `not`, а шаблон, следующий за условным элементом `not`, является шаблоном объекта.

11.8 Обработчики сообщений, определяемые пользователем

Как уже было сказано, согласно определению языка COOL, к объявлению каждого класса автоматически добавляются создаваемые системой обработчики сообщений `print`, `delete`, `put-` и `get-`. Но пользователь имеет также возможность определить собственные обработчики сообщений. Для этой цели используется **конструкция defmessage-handler**, которая имеет следующий общий синтаксис:

```
(defmessage-handler <class-name> <message-name>
                    [<handler-type>]
                    [<optional-comment>]
   (<regular-parameter>*
[<wildcard-parameter>])
   <expression>*)
```

В данном определении терм <regular-parameter> представляет собой однозначную переменную, терм <wildcard-parameter> — это многозначная переменная, а терм <handler-type> — один из символов around, before, primary или after. По умолчанию любой обработчик сообщений является обработчиком сообщений типа primary. Назначение обработчиков типа around, before и after рассматривается в разделе 11.10. Каждый класс имеет свой собственный набор обработчиков сообщений, поэтому не требуется, чтобы имя <message-name> отличалось от имен обработчиков сообщений, используемых в других классах. Объявления <regular-parameter> и <wildcard-parameter> относятся к параметрам, которые могут передаваться в обработчик сообщений при его вызове. Эти параметры действуют по такому же принципу, как и параметры, применяемые в конструкциях deffunction. Кроме того, и тело обработчика сообщений, представленное термом <expression>*, ведет себя так же, как тело конструкции deffunction. Для связывания локальных переменных может использоваться функция bind, а возвращаемым значением для обработчика сообщений становится значение последнего выражения, вычисленного в его теле.

Рассмотрим следующий пример:

```
(defclass RECTANGLE
   (is-a USER)
   (slot height)
   (slot width))
(defclass CIRCLE
   (is-a USER)
   (slot radius))
(defmessage-handler RECTANGLE compute-area
   ()
   (* (send ?self get-height)
   (send ?self get-width)))
(defmessage-handler CIRCLE compute-area
   ()
   (* (pi)
   (send ?self get-radius)
   (send ?self get-radius)))
(definstances figures
   (rectangle-1 of RECTANGLE (height 2) (width 4))
   (circle-1 of CIRCLE (radius 3)))
```

В данном примере определяются два класса, RECTANGLE и CIRCLE, с соответствующими слотами. За каждым классом закреплен обработчик сообщений compute-area. Этот обработчик сообщений предназначен для вычисления пло-

щади каждого объекта. Для класса RECTANGLE площадь экземпляра RECTANGLE вычисляется путем умножения высоты, заданной в экземпляре, на ширину. А для класса CIRCLE площадь экземпляра CIRCLE представляет собой значение числа π, возвращаемое функцией pi, которое умножается на значение радиуса, заданное в экземпляре, возведенное в квадрат. Обратите внимание на то, что в обоих обработчиках сообщений используется переменная ?self. Это — специальная переменная, автоматически определяемая для каждого обработчика сообщений. При вызове обработчика сообщений переменной ?self присваивается значение адреса того экземпляра, которому передается сообщение. Эта переменная может применяться для передачи в экземпляр сообщений, как и было сделано в рассматриваемом примере для выборки значений слотов height, width и radius.

После определения обработчиков сообщений появляется возможность отправлять сообщения compute-area в экземпляры классов RECTANGLE и CIRCLE для получения информации о площади фигуры, заданной этим экземпляром, как в следующем примере:

```
CLIPS> (reset)↵
CLIPS> (send [circle-1] compute-area)↵
28.2743338823081
CLIPS>
(send [rectangle-1] compute-area)↵
8
CLIPS>
```

Здесь заслуживает внимания то, что каждому экземпляру передается одно и то же сообщение, но, вычисляя площадь фигуры, заданной с его помощью, каждый экземпляр отвечает по-разному. Такая способность различных экземпляров отвечать на одно и то же сообщение в характерной для него форме называется **полиморфизмом**.

Сокращенные ссылки на слоты

Если бы всегда приходилось передавать экземпляру сообщения для выборки или установки значения любого слота, то при написании многих программ возникали бы значительные неудобства, поэтому предусмотрен сокращенный механизм, позволяющий получить доступ к любым слотам экземпляра, связанного с переменной ?self. Таким образом, вместо применения следующего выражения:

```
(send ?self get-<slot-name>)
```

для выборки значения слота можно использовать выражение

```
?self:<slot-name>
```

Например, описанные выше обработчики сообщений compute-area можно переопределить следующим образом:

```
(defmessage-handler RECTANGLE compute-area
   ()
   (* ?self:height ?self:width))
(defmessage-handler CIRCLE compute-area
   ()
   (* (pi) ?self:radius ?self:radius))
```

Аналогичный механизм может применяться для задания значения слота. Вместо такого выражения:

```
(send ?self put-<slot-name> <expression>*)
```

может использоваться выражение

```
(bind ?self:slot-name <expression>*)
```

Оба эти альтернативных механизма позволяют обойти этап передачи сообщений и непосредственно манипулировать слотами. Необходимость в этом еще больше возрастает, если на основе обработчиков сообщений get- и put- для создаваемых классов программист определяет обработчики сообщений after, before или around (эта тема рассматривается в разделе 11.10).

По умолчанию в обработчиках сообщений для класса сокращенные ссылки могут применяться только по отношению к слотам, которые непосредственно определены в классе (т.е. к слотам, которые не являются унаследованными). Например, предположим, что заданы две следующие конструкции defclass:

```
(defclass A
   (is-a USER)
   (slot x))
(defclass B
   (is-a A)
   (slot y))
```

В таком случае приведенный ниже диалог показывает, что сокращенную ссылку можно использовать лишь для того, чтобы сослаться на слот y, а не на слот x в обработчике сообщений для класса B.

```
CLIPS>
(defmessage-handler B bmh1 ()
   (* 2 ?self:x))↵
[MSGFUN6] Private slot x of class A cannot be
accessed directly by handlers attached to class B
[PRCCODE3] Undefined variable self:x referenced in
message-handler.
ERROR:
```

```
(defmessage-handler MAIN::B bmh1
   ()
   (* 2 ?self:x)
   )
CLIPS>
(defmessage-handler B bmh2 ()
   (* 2 ?self:y))↵
CLIPS>
```

В обработчике сообщений bmh1 создается ссылка на переменную ?self:x, а это недопустимо, поскольку слот x унаследован от класса A. В обработчике сообщений bmh2 разрешается сделать ссылку на переменную ?self:y, поскольку слот y определен в классе B. В языке COOL поддерживается **инкапсуляция** объектов, а это означает, что детали реализации класса скрываются от постороннего взгляда и доступ к классу ограничивается вполне определенным интерфейсом — обработчиками сообщений, определенными для данного класса. Поскольку интерфейсом для выборки значений слота x из экземпляров класса A является обработчик сообщений get-x, то в обработчике сообщений bmh1 и должен использоваться этот интерфейс. С учетом такой поправки обработчик сообщений bmh1 может быть безошибочно определен следующим образом:

```
CLIPS>
(defmessage-handler B bmh1 ()
   (* 2 (send ?self get-x)))↵
CLIPS>
```

Такой принцип действия слота, обеспечивающий инкапсуляцию, можно отменить с использованием атрибута слота **visibility**. Если этому атрибуту присваивается значение **private**, предусмотренное по умолчанию, то непосредственный доступ к слоту может осуществляться только в обработчиках сообщений класса, определяющего этот слот. А если данному атрибуту присвоено значение **public**, то к слоту может быть получен непосредственный доступ в подклассах и суперклассах любого определяющего его класса. В частности, в предыдущем примере применение следующего определения класса A:

```
(defclass A
   (is-a USER)
   (slot x (visibility public)))
```

позволяет определить обработчик сообщений bmh1 таким образом:

```
(defmessage-handler B bmh1 ()
   (* 2 ?self:x))
```

Отслеживание процесса передачи сообщений и функционирования обработчиков сообщений

При отслеживании сообщений или обработчиков сообщений с использование команды `watch` с параметром `messages` или `message-handlers` каждый раз, после того как начинается и заканчивается выполнение кода обработчика сообщений или начинается и заканчивается передача сообщений, выводится информационное сообщение:

```
CLIPS> (watch messages)↵
CLIPS> (watch message-handlers)↵
CLIPS> (send [circle-1] compute-area)↵
MSG >> compute-area ED:1 (<Instance-circle-1>)
HND >> compute-area primary in class CIRCLE
       ED:1 (<Instance-circle-1>)
HND << compute-area primary in class CIRCLE
       ED:1 (<Instance-circle-1>)
MSG << compute-area ED:1 (<Instance-circle-1>)
28.2743338823081
CLIPS> (send [rectangle-1] compute-area)↵
MSG >> compute-area ED:1 (<Instance-rectangle-1>)
HND >> compute-area primary in class RECTANGLE
       ED:1 (<Instance-rectangle-1>)
HND << compute-area primary in class RECTANGLE
       ED:1 (<Instance-rectangle-1>)
MSG << compute-area ED:1 (<Instance-rectangle-1>)
8
CLIPS>
```

Если сообщения отслеживаются с помощью параметра `messages`, то отладочная информация отображается после передачи в экземпляр сообщения и после полного завершения обработки этого сообщения. Если же с помощью параметра `message-handlers` отслеживается функционирование обработчиков сообщений, то отладочная информация выводится после того, как начинается и заканчивается выполнение кода каждого конкретного обработчика сообщений. При отслеживании функционирования обработчиков сообщений предоставляется вся та информация, которая может быть получена при отслеживании сообщений, а также некоторая дополнительная информация. В приведенном выше диалоге информационные сообщения, выводимые в результате отслеживания сообщений, обозначались префиксом `MSG`, а информация, полученная при отслеживании функционирования обработчиков сообщений, отмечалась префиксом `HND`. Символ >> свидетельствует об инициализации отслеживания определенного со-

общения или обработчика сообщений, а символ << — о завершении отслеживания конкретного сообщения или обработчика сообщений. За символом >> или << следует имя сообщения или обработчика сообщений. В современной версии языка вслед за именем обработчика сообщений предусматривается вывод некоторой дополнительной информации: обозначение типа обработчика сообщений (`before`, `after`, `primary` или `around`), за которым следует имя класса, связанного с данным конкретным функционирующим обработчиком сообщений. Одно-единственное сообщение способно вызвать на выполнение многочисленные обработчики сообщений различных типов или классов, поэтому данная информация позволяет определить, какой из них фактически функционирует. В последнюю очередь отображается фрагмент информации, показывающий глубину вложенности вызовов, обозначаемый символом `ED`, за которым следуют параметры, передаваемые обработчику сообщений. Как и при использовании конструкций `deffunction`, глубина вложенности вызовов позволяет определить, насколько глубоко вложены вызовы конструкций `deffunction` и вызовы обработчиков сообщений. Отсчет глубины вложенности вызовов начинается с нуля, и после каждой передачи управления очередной конструкции `deffunction` или обработчику сообщений глубина увеличивается на единицу. После выхода из каждой конструкции `deffunction` или из обработчика сообщений значение глубины вложенности уменьшается на единицу. В списке параметров в первую очередь всегда указывается параметр, представляющий собой значение переменной `?self`. После этого перечисляются все явно заданные параметры обработчика сообщений.

Возможно также обеспечить отслеживание функционирования сразу всех обработчиков сообщений конкретного класса, указав имя этого класса; для отслеживания функционирования конкретного обработчика сообщений необходимо задать только имя класса и обработчика сообщений; для отслеживания функционирования конкретного обработчика сообщений определенного типа необходимо указать класс, имя обработчика сообщений, а после этого тип (`before`, `after`, `primary` или `around`):

```
CLIPS> (unwatch all)↵
CLIPS> (watch message-handlers CIRCLE)↵
CLIPS> (watch message-handlers RECTANGLE
         compute-area)↵
CLIPS> (watch message-handlers RECTANGLE
         get-height primary)↵
CLIPS>
```

Первая команда `watch` обеспечивает отслеживание функционирования всех обработчиков сообщений, определяемых пользователем и системой для класса `CIRCLE`. Вторая команда `watch` обеспечивает отслеживание функционирования обработчика сообщений `compute-area` типа `primary` для класса `RECTANGLE`

(и обеспечивала бы отслеживание функционирования обработчиков сообщений типа `after`, `before` и `around`, если бы они были определены). Третья команда `watch` отслеживает функционирование только определяемого системой обработчика `get-height` типа `primary` для класса `RECTANGLE`.

Команды `defmessage-handler`

Для манипулирования конструкциями `defmessage-handler` предусмотрено несколько команд. Для отображения текущего списка конструкций `defmessage-handlers`, поддерживаемых системой CLIPS, служит **команда list-defmessage-handlers**. Эта команда имеет следующий синтаксис:

```
(list-defmessage-handlers
   [<defclass-name> [inherit]])
```

В данном определении ключевое слово **inherit** указывает, что в список должны быть включены унаследованные обработчики сообщений. Пример применения команды `list-defmessage-handlers` приведен ниже.

```
CLIPS> (list-defmessage-handlers RECTANGLE)↵
get-height primary in class RECTANGLE
put-height primary in class RECTANGLE
get-width primary in class RECTANGLE
put-width primary in class RECTANGLE
compute-area primary in class RECTANGLE
For a total of 5 message-handlers.
CLIPS>
```

Для отображения текстового представления конструкции `defmessage-handler` используется **команда ppdefmessage-handler** (сокращение от pretty print defmessage-handler — структурированный вывод конструкции `defmessage-handler`). **Команда undefmessage-handler** предназначена для удаления конструкции `defmessage-handler`. **Функция get-defmesage-handler-list** возвращает многозначное значение, содержащее список конструкций `defmessage-handlers`, которые относятся к указанному классу. Эти команды имеют следующий синтаксис:

```
(ppdefmessage-handler <defclass-name>
                      <handler-name>
                      [<handler-type>])
(undefmessage-handler <defclass-name>
                      <handler-name>
                      [<handler-type>])
(get-defmessage-handler-list <defclass-name>
                      [inherit])
```

В данном определении терм <handler-type> обозначает один из символов around, before, primary или after. Примеры применения указанных команд приведены ниже.

```
CLIPS>
(ppdefmessage-handler RECTANGLE compute-area)↵
(defmessage-handler MAIN::RECTANGLE compute-area
   ()
   (* (send ?self get-height) (send ?self
                               get-width)))
CLIPS>
(undefmessage-handler RECTANGLE get-height)↵
[MSGPSR3] System message-handlers may not
be modified.
CLIPS> (undefmessage-handler
RECTANGLE compute-area before)↵
[MSGFUN8] Unable to delete message-handler(s) from
   class RECTANGLE.
CLIPS> (undefmessage-handler RECTANGLE
         compute-area primary)↵
CLIPS> (get-defmessage-handler-list CIRCLE)↵
(CIRCLE get-radius primary
CIRCLE put-radius primary
CIRCLE compute-area primary)
CLIPS>
```

Следует отметить, что обработчики сообщений, определяемые системой, не подлежат удалению. Класс RECTANGLE не имеет обработчика сообщений типа before для экземпляра compute-area, поэтому не может быть удален. В этом классе имеется обработчик сообщений compute-area типа primary, удаление которого возможно, но в команде undefmessage-handler не требуется задавать тип primary, поскольку по умолчанию удаляются именно те обработчики сообщений, которые имеют этот тип. Функция get-defmessage-handler-list возвращает три значения для каждого обработчика сообщений: класс, за которым закреплен этот обработчик сообщений (это имя класса будет отличаться от имени класса, переданного в функцию, только если задано ключевое слово inherit), имя обработчика сообщений и тип обработчика сообщений.

11.9 Доступ к слоту и создание обработчика

Управление доступом к слоту может быть обеспечено с помощью атрибутов слота **access** и **create-accessor**. Атрибут access непосредственно ограничивает

тип доступа, который допускается применять к слоту. Если этому атрибуту присвоено значение **read-write**, предусмотренное по умолчанию, то к этому слоту может быть непосредственно получен доступ для чтения или записи с помощью обработчиков класса, с использованием сокращенного обозначения слота: `?self:<slot-name>`. Если этому атрибуту присвоено значение **read-only**, то может осуществляться выборка значения слота с использованием сокращенного обозначения, но применение функции `bind` для изменения значения не допускается. Единственный способ обеспечить передачу значения для сохранения в слоте — использовать атрибут, заданный по умолчанию (`read-write`). Задание значения атрибута доступа, равного **initialize-only**, аналогично применению атрибута `read-only`, за исключением того, что допускается возможность задавать значение при создании экземпляра (например, с помощью функции `make-instance`).

Для управления автоматическим созданием обработчиков `get-` и `put-` для слотов класса используется атрибут `create-accessor`. Если этому атрибуту присвоено значение **read-write**, которое предусмотрено по умолчанию, то создаются оба обработчика, `get-` и `put-`. Аналогичным образом, если этому атрибуту присвоено значение **read**, то создаются только обработчики `get-`, а если присвоено значение **write** — только обработчики `put-`. Наконец, если этому атрибуту присвоено значение **?NONE**, то не создаются ни обработчики `get-`, ни обработчики `put-`.

Некоторые комбинации атрибутов `access` и `create-accessor` явно вызывают ошибки; например, это происходит, если атрибуту `access` для слота присваивается значение `read-only`, а атрибуту `create-accessor` — значение `read-write`. Очевидно, что не существует возможности выполнять запись в слот, если разрешено только чтение из этого слота.

Рассмотрим пример, в котором демонстрируется применение этих атрибутов. Предположим, что от заказчика получен заказ и необходимо подсчитать его общую стоимость. Каждый заказ обозначается уникальным идентификатором, причем после присваивания этого идентификатора он не должен изменяться. Общая стоимость заказа рассчитывается как стоимость отдельных позиций в заказе, складывающаяся с налогом с оборота. Кроме того, желательно, чтобы общая стоимость вычислялась или задавалась в коде обработчика создаваемого класса. Код, который будет использоваться для разрабатываемого класса, предназначенного для вычисления общей стоимости заказа, приведен ниже.

```
(defclass ORDER
   (is-a USER)
   (slot ID (access initialize-only)
         (default-dynamic (gensym)))
(slot total-price (create-accessor read)
      (default 0.0))
```

```
   (slot order-price (default 0.0))
   (slot sales-tax (default 0.0)))
(defmessage-handler ORDER compute-total-price ()
   (bind ?self:total-price
      (* ?self:order-price
         (+ 1 ?self:sales-tax))))
```

Значение слота ID может быть задано только при создании экземпляра. Если же для этого слота значение останется незаданным, то необходимое значение будет вырабатываться динамически путем вызова функции gensym. Атрибуту create-accessor слота total-price присваивается значение read. Таким образом, обработчик get-total-price будет создаваться, а обработчик put-total-price — нет. Наконец, определяется обработчик compute-total-price, в котором вычисляется общая стоимость заказа путем сложения стоимости заказа с соответствующей суммой налога, начисляемой по указанной ставке налога с оборота. Например, если значение order-price равно 10.00, а значение sales-tax составляет 0.05, то значение total-price будет равно 10.50. Приведенный ниже диалог показывает, какие ограничения распространяются на значения слотов.

```
CLIPS>
(make-instance order1 of ORDER
 (ID #001)
 (order-price 10.00)
 (sales-tax 0.05))↵
[order1]
CLIPS> (send [order1] put-ID #002)↵
[MSGFUN3] ID slot in [order1] of ORDER: write
access denied.
[PRCCODE4] Execution halted during the actions of
message-handler put-ID primary in class ORDER
FALSE
CLIPS>
```

Обратите внимание на то, что значение слоту ID может быть присвоено во время создания экземпляра order1, но в дальнейшем присвоить это значение с помощью обработчика put-ID невозможно. Аналогичным образом, как показывает следующий диалог, может быть осуществлена выборка значения total-price с использованием обработчика get-total-price, а присвоить это значение с помощью обработчика put-total-price невозможно:

```
CLIPS> (send [order1] get-total-price)↵
0.0
CLIPS> (send [order1] put-total-price 10.50)↵
```

```
[MSGFUN1] No applicable primary message-handlers
found for put-total-price.
FALSE
CLIPS>
```

Для вычисления правильного значения стоимости заказа должен быть вызван обработчик compute-total-price:

```
CLIPS> (send [order1] compute-total-price)↵
10.5
CLIPS> (send [order1] get-total-price)↵
10.5
CLIPS>
```

Можно также ввести код обработчика get-total-price вручную, так, чтобы не приходилось явно вызывать обработчик compute-total-price для вычисления правильного значения общей стоимости. Если для определения обработчика get- класса используется атрибут create-accessor, то обработчик сообщений создается в следующей форме:

```
(defmessage-handler <class> get-<slot-name>
            primary ()
?self:<slot-name>)
```

Аналогичным образом, для обработчиков put- применяется такая форма:

```
(defmessage-handler <class> put-<slot-name>
            primary (?value)
(bind ?self:<slot-name> ?value))
```

В обоих случаях терм <class> обозначает имя конструкции defclass, а терм <slot-name> — имя слота в этом классе. Если в классе ORDER значение атрибута create-accessor слота total-price будет изменено на ?NONE, то появится возможность определить обработчик get-total-price, позволяющий вычислять стоимость при каждом его вызове:

```
(defclass ORDER
   (is-a USER)
   (slot ID (access initialize-only)
            (default-dynamic (gensym)))
   (slot total-price (create-accessor ?NONE)
                     (default 0.0))
   (slot order-price (default 0.0))
   (slot sales-tax (default 0.0)))
(defmessage-handler ORDER get-total-price ()
   (send ?self compute-total-price))
```

```
(defmessage-handler ORDER get-total-price ()
   (send ?self compute-total-price))
```

После этого, создав экземпляр ORDER, а затем запрашивая значение общей стоимости, можно каждый раз получать наиболее актуальное значение:

```
CLIPS>
(make-instance order2 of ORDER
   (ID #002)
   (order-price 20.00)
   (sales-tax 0.05))↵
[order2]
CLIPS> (send [order2] get-total-price)↵
21.0
CLIPS>
```

11.10 Обработчики сообщений before, after и around

Действия, выполняемые классом, могут не соответствовать предъявляемым к нему требованиям, притом что возможность модифицировать код класса в соответствии с этими требованиями будет отсутствовать. Причиной этого часто бывает то, что от такого именно функционирования класса зависит работа другого кода. Еще одной причиной может стать недостаточное знакомство разработчика с кодом класса, что не позволяет модифицировать должным образом работу этого класса. В подобных случаях можно определить новый класс, наследующий от родительского класса все свойства, которые должны остаться неизменными, а затем предусмотреть в новом классе другие необходимые свойства, которые должны измениться. Еще раз рассмотрим код примера с определением класса ORDER и попытаемся создать новый класс, который будет гарантировать, что информация об общей стоимости заказа всегда остается актуальной:

```
(defclass ORDER
   (is-a USER)
   (slot ID (access initialize-only))
   (slot total-price (create-accessor read)
                     (default 0.0))
   (slot order-price (default 0.0))
   (slot sales-tax (default 0.0)))
(defmessage-handler ORDER compute-total-price ()
   (bind ?self:total-price
```

```
    (* ?self:order-price
       (+ 1 ?self:sales-tax))))
```

Затем необходимо определить новый класс, в котором требуется реализовать возможность выполнения особых действий. Назовем этот класс MY-ORDER:

```
(defclass MY-ORDER
   (is-a ORDER))
```

Один из подходов, который может быть принят при решении данной задачи, состоит в том, чтобы полностью переопределить обработчик сообщений get-total-price, предусмотрев в нем и новые, и прежние возможности:

```
(defmessage-handler MY-ORDER get-total-price ()
   (send ?self compute-total-price)
   ?self:total-price)
```

В данном случае перед возвратом значения слота total-price вызывается обработчик сообщений compute-total-price. Но при использовании такого подхода возникает целый ряд проблем. Во-первых, этот замысел просто неосуществим. Слот total-price имеет атрибут private, поэтому к нему невозможно получить доступ с помощью ссылки ?self из-за пределов класса ORDER. Кроме того, ссылку ?self:total-price невозможно заменить вызовом (send ?self get-total-price), поскольку это будет вызов обработчика MY-ORDER, а не ORDER. Во-вторых, еще один недостаток этого подхода связан с дублированием кода. В данном случае дублируется очень небольшой и несложный фрагмент кода, ?self:total-price, но предположим, что в действительности этот фрагмент весьма велик. Неважно даже то, что для дублирования кода в программе требуется дополнительное пространство. Гораздо важнее то, что после внесения изменений в первоначальный код обработчика ORDER новые свойства этого обработчика не будут унаследованы обработчиком MY-ORDER. Даже по одной этой причине не следует пытаться переопределять обработчик таким образом.

Возможно также немного переопределить обработчик для получения требуемых свойств следующим образом:

```
(defmessage-handler MY-ORDER get-total-price ()
   (send ?self compute-total-price)
   (call-next-handler))
```

Функция call-next-handler вызывает следующий обработчик сообщений, который был перекрыт текущим обработчиком сообщений. При вызове этой функции не требуется задавать какие-либо параметры. Следующему обработчику передаются параметры, с которыми был вызван текущий обработчик. Отслеживание работы обработчиков сообщений во время передачи сообщения get-total-

price экземпляру MY-ORDER позволяет убедиться в том, что вызываются обработчики сообщений get-total-price обоих классов, MY-ORDER и ORDER:

```
CLIPS>
(make-instance order3 of MY-ORDER
   (ID #003)
   (order-price 10.00)
   (sales-tax 0.05))↵
CLIPS> (watch message-handlers)↵
CLIPS> (send [order3] get-total-price)↵
HND >> get-total-price primary in class MY-ORDER
       ED:1 (<Instance-order3>)
HND >> compute-total-price primary in class ORDER
       ED:2 (<Instance-order3>)
HND << compute-total-price primary in class ORDER
       ED:2 (<Instance-order3>)
HND >> get-total-price primary in class ORDER
       ED:1 (<Instance-order3>)
HND << get-total-price primary in class ORDER
       ED:1 (<Instance-order3>)
HND << get-total-price primary in class MY-ORDER
       ED:1 (<Instance-order3>)
10.5
CLIPS>
```

Обработчики before и after

В языке CLIPS, кроме обработчиков сообщений типа primary, которые применяются по умолчанию, если тип обработчика не указан, предусмотрена также возможность использовать обработчики типов before, after и around. Тип обработчика задается после имени обработчика, причем каждый класс может иметь обработчики каждого типа. Не удивительно, что тип обработчика сообщений before определяет обработчик сообщений, который должен быть вызван на выполнение перед обработчиком сообщений типа primary. Ниже приведен пример применения обработчика сообщений типа before в классе MY-ORDER.

```
(defmessage-handler MY-ORDER get-total-price
            before ()
(send ?self compute-total-price))
```

В отличие от рассматриваемого перед этим обработчика сообщений get-total-price типа primary, в данном обработчике передается только сообщение compute-total-price, а функция call-next-handler не вызывается.

Вызов этой функции не требуется, поскольку так или иначе в первую очередь вызывается обработчик типа before класса MY-ORDER, а затем — обработчик типа primary класса ORDER. Удалив ранее определенный обработчик типа primary класса MY-ORDER, в этом можно убедиться, отслеживая работу обработчиков сообщений следующим образом:

```
CLIPS>
(make-instance order4 of MY-ORDER
   (ID #004)
   (order-price 10.00)
   (sales-tax 0.05))↵
[order4]
CLIPS> (watch message-handlers)↵
CLIPS> (send [order4] get-total-price)↵
HND >> get-total-price before in class MY-ORDER
       ED:1 (<Instance-order4>)
HND >> compute-total-price primary in class ORDER
       ED:2 (<Instance-order4>)
HND << compute-total-price primary in class ORDER
       ED:2 (<Instance-order4>)
HND << get-total-price before in class MY-ORDER
       ED:1 (<Instance-order4>)
HND >> get-total-price primary in class ORDER
       ED:1 (<Instance-order4>)
HND << get-total-price primary in class ORDER
       ED:1 (<Instance-order4>)
10.5
CLIPS>
```

Вначале вызывается обработчик сообщений get-total-price типа before класса MY-ORDER, который вызывает обработчик сообщений compute-total-price типа primary класса ORDER, а затем осуществляется выход из обоих обработчиков. Наконец вызывается обработчик сообщений get-total-price типа primary класса ORDER, который возвращает обновленное значение слота total-price.

Вместо вычисления правильного значения total-price перед вызовом обработчика get-total-price можно также вычислить значение total-price после внесения изменений либо в слот sales-tax, либо в order-price. Эту задачу можно осуществить, определив следующие обработчики сообщений типа after:

```
(defmessage-handler MY-ORDER put-sales-tax
                    after (?value)
```

```
   (if (numberp (send ?self get-order-price))
      then
      (send ?self compute-total-price)))
(defmessage-handler MY-ORDER put-order-price
                    after (?value)
   (if (numberp (send ?self get-sales-tax))
      then
      (send ?self compute-total-price)))
```

В указанных обработчиках сообщений обработчик `compute-total-price` вызывается после вызова обработчиков `put-sales-tax` и `put-order-price` типа `primary`. В каждом из этих обработчиков необходимо проверить, определено ли должным образом (как числовое) значение другого слота, используемое в обработчике сообщений `compute-total-price`, поскольку в течение всего процесса создания экземпляра таким слотам не присваивается предусмотренное по умолчанию значение, равное `0.0`. В этом можно убедиться, отслеживая работу обработчиков сообщений после удаления ранее заданного обработчика `get-total-price` типа `before`:

```
CLIPS>
(make-instance order5 of MY-ORDER
   (ID #005)
   (order-price 10.00)
   (sales-tax 0.05))↵
[order5]
CLIPS> (send [order5] get-total-price)↵
10.5
CLIPS> (watch message-handlers)↵
CLIPS> (send [order5] put-order-price 20.00)↵
HND >> put-order-price primary in class ORDER
       ED:1 (<Instance-order5> 20.0)
HND << put-order-price primary in class ORDER
       ED:1 (<Instance-order5> 20.0)
HND >> put-order-price after in class MY-ORDER
       ED:1 (<Instance-order5> 20.0)
HND >> get-sales-tax primary in class ORDER
       ED:2 (<Instance-order5>)
HND << get-sales-tax primary in class ORDER
       ED:2 (<Instance-order5>)
HND >> compute-total-price primary in class ORDER
       ED:2 (<Instance-order5>)
HND << compute-total-price primary in class ORDER
```

```
       ED:2 (<Instance-order5>)
HND << put-order-price after in class MY-ORDER
       ED:1 (<Instance-order5> 20.0)
20.0
CLIPS> (send [order5] get-total-price)↵
HND >> get-total-price primary in class ORDER
       ED:1 (<Instance-order5>)
HND << get-total-price primary in class ORDER
       ED:1 (<Instance-order5>)
21.0
CLIPS>
```

После создания экземпляра можно обнаружить, что значение слота total-price является актуальным. В этот момент работа обработчиков сообщений не отслеживалась, поскольку для нас не представляют интереса несколько сообщений, отправленных во время создания экземпляра. Изменив значение слота order-price путем вызова обработчика put-order-price, можно обнаружить, что вначале вызывается обработчик сообщений put-order-price типа primary класса ORDER, а затем осуществляется выход из данного обработчика. После этого вызывается обработчик сообщений put-order-price типа after класса MY-ORDER, который затем вызывает обработчик сообщений get-sales-tax типа primary класса ORDER для определения того, присвоено ли слоту sales-tax соответствующее значение. Поскольку слот sales-tax имеет допустимое значение, вызывается обработчик compute-total-price типа primary класса ORDER для вычисления правильного значения стоимости, после чего происходит выход из обоих обработчиков — compute-total-price типа primary и put-order-price типа after. В последующем для возврата правильного значения передается сообщение get-total-price.

Обработчики around

В связи с тем, что приходится проверять наличие числовых значений в слотах order-price и sales-tax с помощью обработчиков сообщений типа after, программа становится громоздкой, поэтому рассмотрим другой способ решения этой задачи. Вначале удалим код проверки значений из обработчиков сообщений типа after:

```
(defmessage-handler MY-ORDER put-sales-tax
                    after (?value)
   (send ?self compute-total-price))
(defmessage-handler MY-ORDER put-order-price
                    after (?value)
   (send ?self compute-total-price))
```

Четвертым и последним типом обработчиков сообщений, предусмотренных в языке COOL, является обработчик сообщений типа `around`. Обработчики сообщений такого типа называют также *оболочками*, поскольку они охватывают своим кодом код обработчиков сообщений других типов, выполняемый до и после них. Обработчик сообщений типа `around` представляет собой своего рода комбинацию обработчиков типа `before` и `after`, которая позволяет также аварийно прервать обработку сообщения в процессе передачи сообщения. Ниже приведен обработчик типа `around` для сообщения `compute-total-price`.

```
(defmessage-handler MY-ORDER compute-total-price
                    around ()
   (if (or (not (numberp
                (send ?self get-order-price)))
           (not (numberp
                (send ?self get-sales-tax))))
           then
           (return))
   (call-next-handler))
```

Прежде всего в этом обработчике типа `around` проверяется, являются ли значения слотов `order-price` и `sales-tax` числовыми. В случае отрицательного результата этой проверки в данном обработчике происходит возврат и не выполняются какие-либо иные действия. В этот момент не вызывается ни один из обработчиков типа `before`, `primary` или `after` для сообщения `compute-total-price`. Если значения обоих слотов, `order-price` и `sales-tax`, являются числовыми, то вызывается функция `call-next-handler` и вызываются другие обработчики сообщений. В данном случае должен быть вызван лишь обработчик сообщений `compute-total-price` типа `primary` класса `ORDER`. Это демонстрируется в приведенном ниже диалоге. Поскольку вывод становится довольно объемным, части трассировки действий, выполняемых обработчиком сообщений, чередуются с комментариями.

```
CLIPS> (watch message-handlers)↵
CLIPS>
(make-instance order5 of MY-ORDER
   (ID #005)
   (order-price 10.00)
   (sales-tax 0.05))↵
HND >> create primary in class USER
       ED:1 (<Instance-order5>)
HND << create primary in class USER
       ED:1 (<Instance-order5>)
HND >> put-ID primary in class ORDER
```

```
       ED:1 (<Instance-order5> #005)
HND << put-ID primary in class ORDER
       ED:1 (<Instance-order5> #005)
```

С помощью вызова функции make-instance создается экземпляр order5. Сразу после создания экземпляра вызывается обработчик сообщений create — еще один обработчик сообщений, определяемый системой. Вслед за созданием исходного экземпляра в слоты этого экземпляра помещаются значения, заданные в вызове функции make-instance. Вначале, как показано ниже, вызывается обработчик сообщений put-ID для сохранения значения #005 в слоте ID.

```
HND >> put-order-price primary in class ORDER
       ED:1 (<Instance-order5> 10.0)
HND << put-order-price primary in class ORDER
       ED:1 (<Instance-order5> 10.0)
HND >> put-order-price after in class MY-ORDER
       ED:1 (<Instance-order5> 10.0)
HND >> compute-total-price around in class MY-ORDER
       ED:2 (<Instance-order5>)
HND >> get-order-price primary in class ORDER
       ED:3 (<Instance-order5>)
HND << get-order-price primary in class ORDER
       ED:3 (<Instance-order5>)
HND >> get-sales-tax primary in class ORDER
       ED:3 (<Instance-order5>)
HND << get-sales-tax primary in class ORDER
       ED:3 (<Instance-order5>)
HND >> get-order-price primary in class ORDER
       ED:3 (<Instance-order5>)
HND << get-order-price primary in class ORDER
       ED:3 (<Instance-order5>)
HND << compute-total-price around in class MY-ORDER
       ED:2 (<Instance-order5>)
HND << put-order-price after in class MY-ORDER
       ED:1 (<Instance-order5> 10.0)
```

Затем вызывается обработчик сообщений put-order-price типа primary класса ORDER для сохранения значения 10.0 в слоте order-price. После того как этот обработчик завершит свою работу, вызывается обработчик put-order-price типа after класса MY-ORDER. Данный обработчик передает сообщение compute-total-price, в результате чего вызывается обработчик compute-total-price типа around класса MY-ORDER. В обработчике типа around вызывается обработчик get-order-price типа primary класса ORDER для

определения того, является ли значение слота order-price числовым (оно таковым и является, поскольку только что установлено равным 10.0). Затем вызывается обработчик get-sales-tax типа primary класса ORDER для определения того, является ли значение слота sales-tax числовым. Если эта проверка оканчивается неудачей, в связи с тем, что рассматриваемому слоту еще не было присвоено значение с помощью функции make-instance, то обработчик типа around снова вызывает обработчик get-order-price типа primary класса ORDER и возвращает требуемое значение. В таком случае обработчик computer-order-price типа around заканчивает свою работу, не вызывая обработчик типа primary. Наконец осуществляется выход из обработчика put-order-price типа after класса MY-ORDER, как показано ниже.

```
HND >> put-sales-tax primary in class ORDER
       ED:1 (<Instance-order5> 0.05)
HND << put-sales-tax primary in class ORDER
       ED:1 (<Instance-order5> 0.05)
HND >> put-sales-tax after in class MY-ORDER
       ED:1 (<Instance-order5> 0.05)
HND >> compute-total-price around in class MY-ORDER
       ED:2 (<Instance-order5>)
HND >> get-order-price primary in class ORDER
       ED:3 (<Instance-order5>)
HND << get-order-price primary in class ORDER
       ED:3 (<Instance-order5>)
HND >> get-sales-tax primary in class ORDER
       ED:3 (<Instance-order5>)
HND << get-sales-tax primary in class ORDER
       ED:3 (<Instance-order5>)
HND >> compute-total-price primary in class ORDER
       ED:2 (<Instance-order5>)
HND << compute-total-price primary in class ORDER
       ED:2 (<Instance-order5>)
HND << compute-total-price around in class MY-ORDER
       ED:2 (<Instance-order5>)
HND << put-sales-tax after in class MY-ORDER
       ED:1 (<Instance-order5> 0.05)
```

После этого вызывается обработчик сообщений put-sales-tax типа primary класса ORDER для сохранения значения 0.05 в слоте sales-tax. Непосредственно после завершения работы этого обработчика вызывается обработчик put-sales-tax типа after класса MY-ORDER. Данный обработчик передает сообщение compute-total-price, в результате чего вызывается обработчик

compute-total-price типа around класса MY-ORDER. В этом обработчике типа around вызывается обработчик get-order-price типа primary класса ORDER для определения того, является ли значение слота order-price числовым. Оно таковым и является, поэтому затем вызывается обработчик get-sales-tax типа primary класса ORDER для определения того, является ли числовым значение слота sales-tax (и оно является числовым, поскольку этому слоту было только что присвоено значение 0.05). Теперь в обработчике типа around вызывается функция call-next-handler, которая, в свою очередь, вызывает обработчик compute-total-price типа primary класса ORDER, вычисляющий обновленное значение слота total-price. Наконец обработчик compute-total-price типа primary класса ORDER завершает свою работу, после чего происходит выход из обработчика сообщений compute-total-price типа around класса MY-ORDER, а затем — выход из обработчика put-sales-tax типа after класса MY-ORDER:

```
HND >> init primary in class USER
       ED:1 (<Instance-order5>)
HND << init primary in class USER
       ED:1 (<Instance-order5>)
[order5]
CLIPS>
```

Затем во вновь созданный экземпляр передается сообщение init, после чего происходит возврат имени экземпляра. Обработчик сообщений init — это еще один обработчик сообщений, определяемый системой. Он вызывается в конце процесса создания экземпляра, после инициализации значений слотов.

Перекрытие параметров обработчика сообщений

В системе CLIPS предусмотрена возможность перекрывать параметры, передаваемые обработчику сообщений, с помощью **команды override-next-handler**. Эта команда имеет следующий синтаксис:

```
(override-next-handler <expression>*)
```

В данном определении каждое выражение <expression> представляет собой параметр, передаваемый в качестве замены существующего параметра в следующий обработчик сообщений. Для ознакомления с примером использования этой команды рассмотрим, как можно было бы обрабатывать заказы на товары в иностранной валюте. Снова предположим, что по различным причинам нежелательно непосредственно модифицировать класс ORDER. Если значение order-price в экземпляре ORDER задано в долларах США, то можно предусмотреть специальную обработку информации с учетом стоимости, представлен-

ной в долларах. Например, за доставку заказов со стоимостью больше 100 долларов может не взиматься оплата за доставку.

Один из подходов к обработке данных о стоимости, представленных в другой валюте, мог бы состоять в том, чтобы определить класс, который наследует свои свойства от класса ORDER, но обеспечивает автоматическое преобразование стоимостей, заданных в иностранной валюте и в долларах США. Такой подход реализован в следующей конструкции defclass с именем FOREIGN-ORDER и в связанных с ней обработчиках:

```
(defclass FOREIGN-ORDER
   (is-a ORDER)
   (slot exchange-rate (default 1.0)))
(defmessage-handler FOREIGN-ORDER get-order-price
                    around ()
   (* ?self:exchange-rate (call-next-handler)))
(defmessage-handler FOREIGN-ORDER put-order-price
                    around (?value)
(override-next-handler
   (/ ?value ?self:exchange-rate)))
```

Класс FOREIGN-ORDER имеет слот exchange-rate, который используется для представления валютного курса, связывающего иностранную валюту и доллары США. Например, если валютный курс равен 2, то 10 долларов США эквивалентны 20 единицам иностранной валюты. Для использования данного подхода на практике, по-видимому, потребовалось бы предусмотреть еще один слот, который указывал бы название единицы иностранной валюты (например, евро), но для данного примера в этом нет необходимости.

В обработчике get-order-price типа around класса FOREIGN-ORDER вызывается обработчик get-order-price типа primary класса ORDER с помощью команды call-next-handler. Затем возвращаемое значение умножается на значение слота exchange-rate и происходит возврат полученного значения. В обработчике put-order-price типа around класса FOREIGN-ORDER вызывается обработчик put-order-price типа primary класса ORDER с помощью команды override-next-handler, но вместо передачи значения, соответствующего стоимости в иностранной валюте, это значение делится на валютный курс и вместо него передается полученное значение, представленное в долларах США.

Мы можем понаблюдать за действиями, выполняемыми в классе FOREIGN-CURRENCY, отслеживая работу обработчиков сообщений. Вначале создадим экземпляр класса:

```
CLIPS> (unwatch all)↵
CLIPS>
```

```
(make-instance order6 of FOREIGN-ORDER
   (ID #006)
   (exchange-rate 2)
   (sales-tax .10))↵
[order6]
CLIPS>
```

Затем изменим значение слота order-price, введя вместо него значение 20.00 в иностранной валюте:

```
CLIPS> (watch message-handlers)↵
CLIPS>
(send [order6] put-order-price 20.00)↵
HND >> put-order-price around in class FOREIGN-ORDER
       ED:1 (<Instance-order6> 20.0)
HND >> put-order-price primary in class ORDER
       ED:1 (<Instance-order6> 10.0)
HND << put-order-price primary in class ORDER
       ED:1 (<Instance-order6> 10.0)
HND << put-order-price around in class FOREIGN-ORDER
       ED:1 (<Instance-order6> 20.0)
10.0
CLIPS>
```

Прежде всего вызывается обработчик put-order-price типа around класса FOREIGN-ORDER со значением 20. После этого вызывается обработчик put-order-price типа primary класса ORDER со значением 10, полученным в результате вызова команды override-next-handler. Затем указанное значение в долларах США передается назад в качестве возвращаемого значения с помощью команды send.

Операция выборки значения слота order-price осуществляется аналогичным образом:

```
CLIPS> (send [order6] get-order-price)↵
HND >> get-order-price around in class FOREIGN-ORDER
       ED:1 (<Instance-order6>)
HND >> get-order-price primary in class ORDER
       ED:1 (<Instance-order6>)
HND << get-order-price primary in class ORDER
       ED:1 (<Instance-order6>)
HND << get-order-price around in class FOREIGN-ORDER
       ED:1 (<Instance-order6>)
20.0
CLIPS>
```

Вначале вызывается обработчик get-order-price типа around класса FOREIGN-ORDER, который вызывает обработчик get-order-price типа primary класса ORDER с помощью команды call-next-handler. Этот обработчик типа primary возвращает значение 10 в долларах США, которое затем умножается на валютный курс в обработчике типа around для получения окончательного возвращаемого значения в единицах иностранной валюты, равного 20.

Передача сообщения print экземпляру FOREIGN-CURRENCY показывает, что значение стоимости в иностранной валюте фактически хранится в виде суммы в долларах США, а не в виде суммы в иностранной валюте:

```
CLIPS> (unwatch all)↵
CLIPS> (send [order6] print)↵
[order6] of FOREIGN-ORDER
(ID #006)
(total-price 0.0)
(order-price 10.0)
(sales-tax 0.1)
(exchange-rate 2)
CLIPS>
```

Порядок вызова обработчиков на выполнение

Выше были описаны четыре метода модификации действий, выполняемых классом ORDER путем закрепления обработчиков за классом MY-ORDER: перекрытие обработчика primary, определение обработчика before, определение обработчика after, а также определение обработчиков after и around. Теперь необходимо найти ответ на очевидный вопрос: какой подход является наилучшим? В данном случае, по-видимому, наилучшим решением является использование обработчика before. В этом подходе применяется наименьший объем кода, к тому же он предоставляет одно существенное преимущество по сравнению с тем, когда осуществляется перекрытие обработчика primary (последний характеризуется использованием количества кода, находящегося на втором месте после наименьшего): в нем вызываются все унаследованные обработчики сообщений типа before и after, принадлежащие к классу или его суперклассам, если из-за ошибки, возникшей в обработчике around, не прекращается обработка сообщения. Это означает, что в классе можно лишь слегка модифицировать действия, выполняемые суперклассом, с использованием обработчиков типа before и after, не перекрывая обработчик типа primary, а в подклассах невозможно предотвратить вызов на выполнение обработчика типа before или типа after, если только в них не прекращена обработка сообщения, что влечет за собой прекращение выполнения всех обработчиков типа before, after

и primary. Если действия, выполняемые существующим классом, модифицированы путем переопределения их в новом классе, а затем перекрыт обработчик типа primary, то новый обработчик типа primary также будет подвержен перекрытию в подклассе. Если в перекрывающем классе не вызывается функция call-next-handler, то не будет вызвана на выполнение и новая версия обработчика типа primary. Тем самым мы не утверждаем, что не следует никогда перекрывать обработчик типа primary, но если вы задумали какие-то специальные действия, возможность перекрытия которых в классах, наследующих свойства, нежелательна, то определенно следует подумать о возможности размещения кода реализации этих действий в обработчике типа before или after.

Предположим, что принято решение ввести код реализации каких-либо специализированных действий в несколько обработчиков типа before, after и around, унаследованных классом. В таком случае важно знать, в каком порядке вызываются различные обработчики сообщений. После передачи экземпляру какого-либо сообщения с помощью команды send определяются все применимые обработчики сообщений и выполняются описанные ниже шаги.

1. Если есть какие-либо не вызванные обработчики типа around, то вызвать наиболее конкретный из них; в противном случае перейти к шагу 2. Если в вызванном обработчике типа around вызывается функция call-next-handler или override-next-handler, то повторить данный шаг; в противном случае перейти к шагу 6.
2. Если есть какие-либо не вызванные обработчики типа before, то вызвать наиболее конкретный обработчик, дождаться завершения его работы, а затем повторить данный шаг. В противном случае перейти к шагу 3. Возвращаемые значения обработчиков before игнорируются. Не предусмотрено никакого способа непосредственно возвратить эти значения в другой обработчик.
3. Если есть какие-нибудь не вызванные обработчики primary, то вызвать наиболее конкретный обработчик; в противном случае перейти к шагу 4. Если вызванный обработчик primary вызывает обработчик call-next-handler или override-next-handler, то повторить данный шаг; в противном случае перейти к шагу 4. Экземпляру не может быть передано сообщение, если отсутствует по меньшей мере один применимый обработчик primary.
4. Предоставить возможность каждому обработчику primary завершить свою работу и выполнить возврат, а затем удалить его из списка вызванных обработчиков primary. Если обработчик primary снова вызывает обработчик call-next-handler или override-next-handler, возвратиться к шагу 3. После завершения работы всех обработчиков primary записать

в память значение, возвращенное последним выполненным обработчиком primary, и перейти к шагу 5.

5. Если есть какие-либо не вызванные обработчики after, то вызвать наименее конкретный обработчик, дождаться завершения его работы, а затем повторить данный шаг. В противном случае перейти к шагу 6. Возвращаемые значения обработчиков after игнорируются. Не предусмотрено никакого способа непосредственно возвратить эти значения в другой обработчик.
6. Если не вызванные обработчики around отсутствуют, перейти к шагу 7. В противном случае предоставить возможность каждому обработчику around завершить свою работу и выполнить возврат, а затем удалить его из списка вызванных обработчиков around. Если какой-либо обработчик around снова вызывает обработчик call-next-handler или override-next-handler, вернуться к шагу 1. После завершения работы всех обработчиков around записать в память значение, возвращенное последним выполненным обработчиком around и перейти к шагу 7.
7. Окончательным возвращаемым значением команды send становится возвращаемое значение самого конкретного обработчика around, полученное на шаге 6. Если обработчики around отсутствуют, то используется возвращаемое значение самого конкретного обработчика primary, полученное на шаге 4. Получение возвращаемого значения функции call-next-handler или override-next-handler — это последнее действие, выполняемое в следующем вызванном обработчике around или primary. Обработчик может либо игнорировать это значение, либо использовать его в качестве возвращаемого значения.

На рис. 11.2 приведена блок-схема, позволяющая получить такие же результаты, как и при выполнении шагов 1–7. Шаг 1 начинается с прямоугольника в верхнем левом углу этого рисунка. А на рис. 11.3 показана блок-схема, позволяющая получить результаты, эквивалентные результатам шагов 3 и 4, касающиеся выполнения применимых обработчиков типа primary, которые связаны с сообщением, переданным экземпляру. В тех случаях, когда отсутствуют применимые обработчики типа around, before или after (т.е. имеются только обработчики типа primary), блок-схема, приведенная на рис. 11.2, сокращается и становится эквивалентной блок-схеме на рис. 11.3.

Для демонстрации порядка вызова обработчиков на выполнение будут использоваться следующие конструкции:

```
(defclass A
   (is-a USER))
(defmessage-handler A msg1 primary ()
   (return msg1-A))
(defmessage-handler A msg1 before ())
```

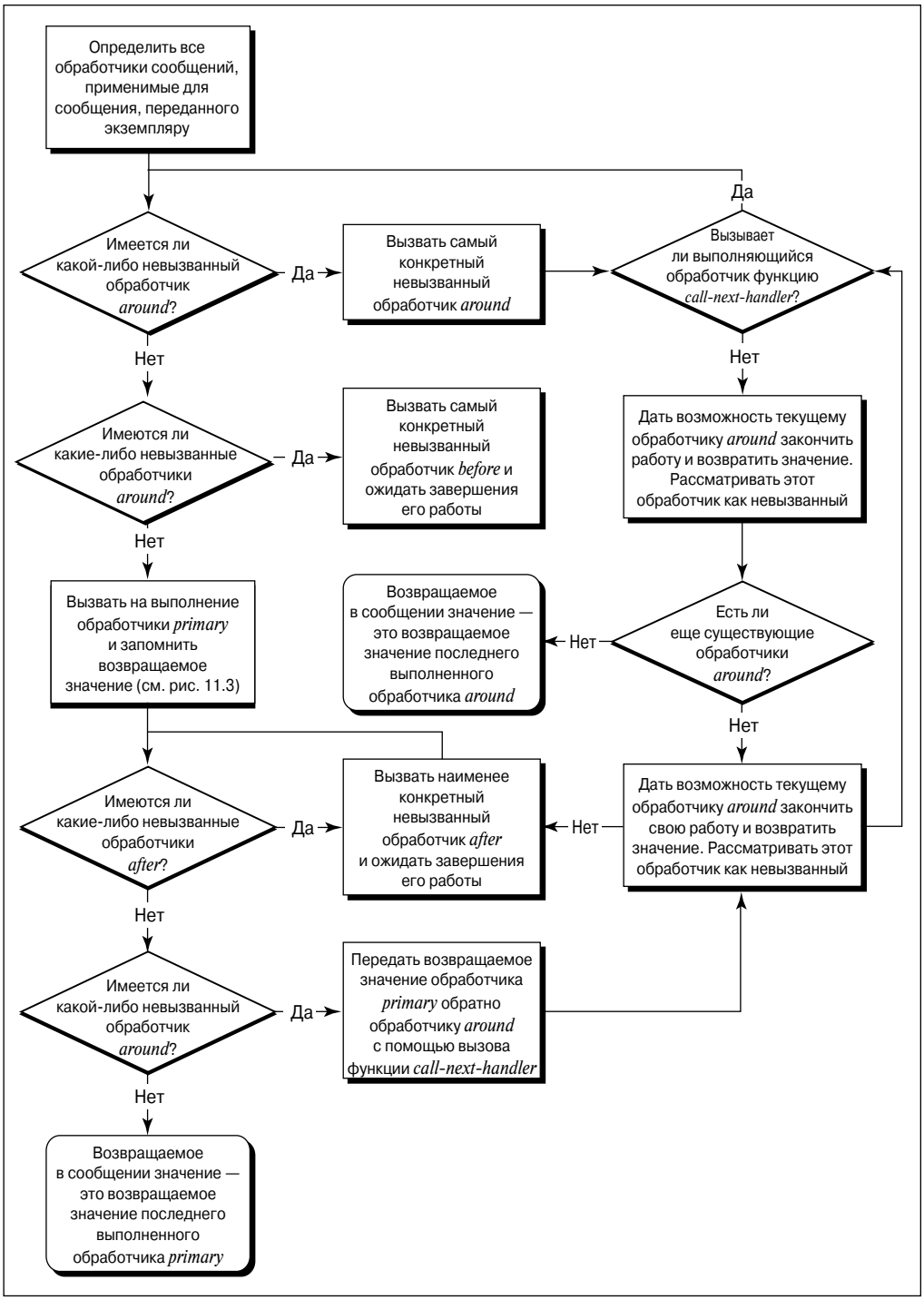


Рис. 11.2. Определение порядка вызова обработчиков на выполнение

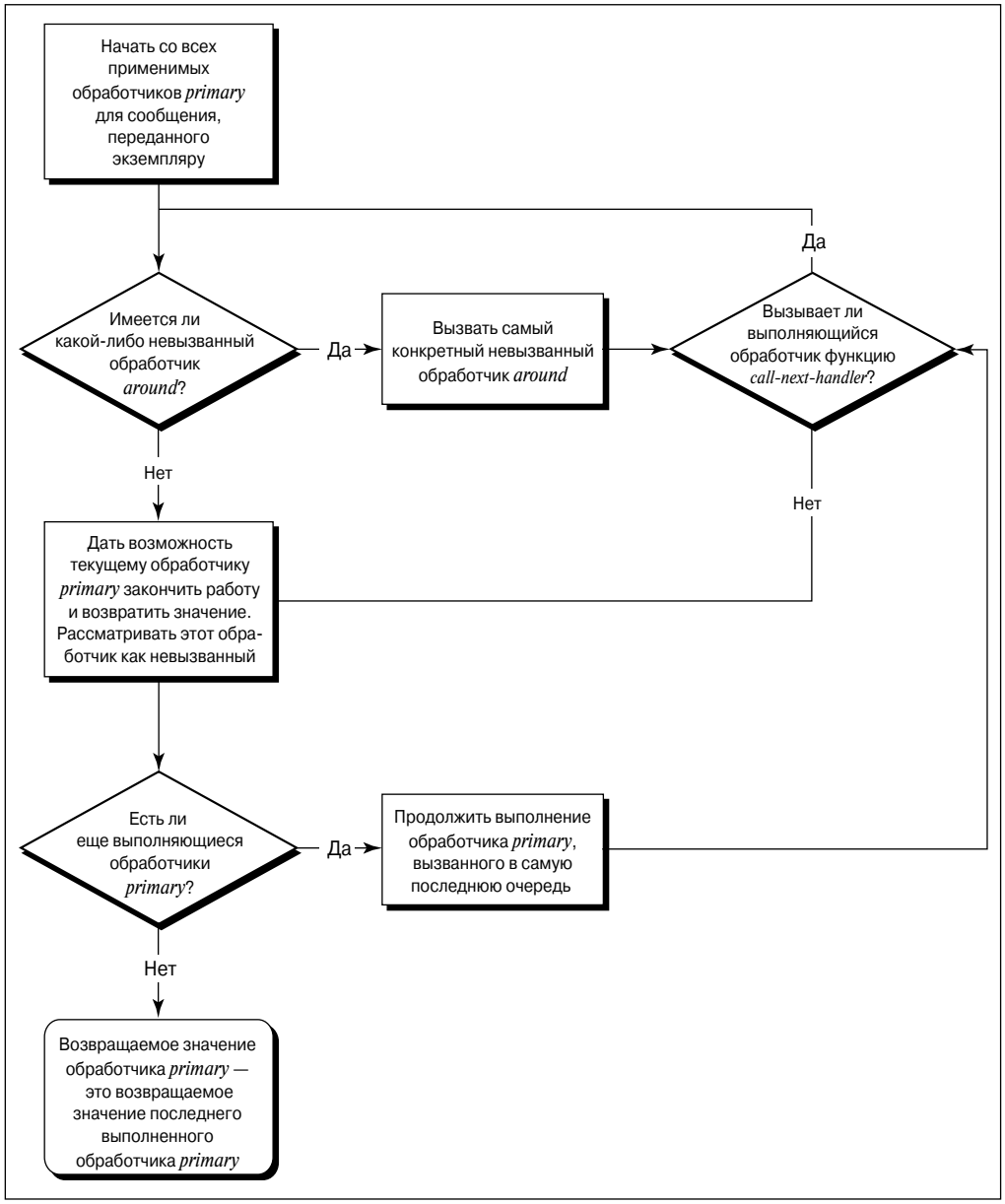


Рис. 11.3. Определение порядка вызова на выполнение обработчиков типа `primary`

```
(defmessage-handler A msg1 after ())
(defmessage-handler A msg1 around ()
   (call-next-handler))
(defclass B
```

```
   (is-a A))
(defmessage-handler B msg1 primary ()
   (return msg1-B))
(defmessage-handler B msg1 before ())
(defmessage-handler B msg1 after ())
(defmessage-handler B msg1 around ()
   (call-next-handler))
```

Класс B является подклассом класса A. Каждый класс имеет свои собственные обработчики primary, before, after и around. Вначале создадим экземпляры обоих классов:

```
CLIPS> (make-instance [a1] of A)↵
[a1]
CLIPS> (make-instance [b1] of B)↵
[b1]
CLIPS>
```

После этого организуем отслеживание работы обработчиков сообщений командой watch с помощью ключевого слова message-handlers и передадим экземпляру a1 сообщение msg1:

```
CLIPS> (watch message-handlers)↵
CLIPS> (send [a1] msg1)↵
HND >> msg1 around in class A
       ED:1 (<Instance-a1>)
HND >> msg1 before in class A
       ED:1 (<Instance-a1>)
HND << msg1 before in class A
       ED:1 (<Instance-a1>)
HND >> msg1 primary in class A
       ED:1 (<Instance-a1>)
HND << msg1 primary in class A
       ED:1 (<Instance-a1>)
HND >> msg1 after in class A
       ED:1 (<Instance-a1>)
HND << msg1 after in class A
       ED:1 (<Instance-a1>)
HND << msg1 around in class A
       ED:1 (<Instance-a1>)
msg1-A
CLIPS>
```

Для этого сообщения есть четыре применимых обработчика сообщений: обработчики msg1 типа primary, before, after и around класса A. Начиная с шага 1 вызывается на выполнение обработчик msg1 типа around класса A. В этом обработчике вызывается функция call-next-handler, поэтому шаг 1 повторяется. Поскольку нет других обработчиков типа around, оставшихся не вызванными, переходим к шагу 2. Обработчику msg1 типа before класса A разрешается выполнить свой код и возвратить управление. Другие не вызванные обработчики типа before отсутствуют, поэтому переходим к шагу 3. На выполнение вызывается обработчик сообщений msg1 типа primary класса A. В нем не вызывается функция call-next-handler, поэтому переходим к шагу 4. Обработчику сообщений msg1 типа primary класса A разрешается завершить свое выполнение, после чего он возвращает символ msg1-A. На этом выполнение всех обработчиков типа primary завершается, поэтому переходим к шагу 5. Теперь разрешается выполнить свой код и возвратить управление обработчику msg1 типа after класса A. Какие-либо иные не вызванные обработчики типа after отсутствуют, поэтому переходим к шагу 6. Обработчику msg1 типа around класса A разрешается закончить свою работу. Последним действием, выполняемым этим обработчиком, является вызов функции call-next-handler, поэтому возвращаемым значением становится возвращаемое значение следующего вызванного обработчика типа around или primary. В данном случае возвращаемое значение msg1-A должен был иметь обработчик msg1 типа primary класса A, поэтому возвращаемое значение обработчика типа around остается тем же самым. На этом работа всех обработчиков типа around заканчивается, поэтому переходим к шагу 7. Окончательным возвращаемым значением команды send становится значение msg1-A, возвращенное обработчиком msg1 типа around класса A.

Теперь передадим сообщение msg1 экземпляру b1 следующим образом:

```
CLIPS> (send [b1] msg1)↵
HND >> msg1 around in class B
    ED:1 (<Instance-b1>)
HND >> msg1 around in class A
    ED:1 (<Instance-b1>)
HND >> msg1 before in class B
    ED:1 (<Instance-b1>)
HND << msg1 before in class B
    ED:1 (<Instance-b1>)
HND >> msg1 before in class A
    ED:1 (<Instance-b1>)
HND << msg1 before in class A
    ED:1 (<Instance-b1>)
```

```
HND >> msg1 primary in class B
    ED:1 (<Instance-b1>)
HND << msg1 primary in class B
    ED:1 (<Instance-b1>)
HND >> msg1 after in class A
    ED:1 (<Instance-b1>)
HND << msg1 after in class A
    ED:1 (<Instance-b1>)
HND >> msg1 after in class B
    ED:1 (<Instance-b1>)
HND << msg1 after in class B
    ED:1 (<Instance-b1>)
HND << msg1 around in class A
    ED:1 (<Instance-b1>)
HND << msg1 around in class B
    ED:1 (<Instance-b1>)
msg1-B
CLIPS>
```

Для данного сообщения имеются восемь применимых обработчиков сообщений: обработчики msg1 типа primary, before, after и around класса A и обработчики msg1 типа primary before, after и around класса B. Начиная с шага 1 выполняется обработчик msg1 типа around класса B, поскольку он является наиболее конкретным обработчиком around. В этом обработчике вызывается функция call-next-handler, поэтому шаг 1 повторяется. На следующем месте среди наиболее конкретных обработчиков around находится обработчик msg1 типа around класса A. В этом обработчике также вызывается функция call-next-handler, поэтому шаг 1 повторяется еще один раз. Какие-либо оставшиеся не вызванными обработчики around отсутствуют, поэтому переходим к шагу 2. Наиболее конкретным обработчиком типа before является обработчик msg1 типа before класса B, поэтому он вызывается на выполнение в первую очередь, после чего разрешается завершить его работу. На следующем месте среди наиболее конкретных обработчиков этого типа находится обработчик msg1 типа before класса A, поэтому разрешается его выполнение и возврат управления. Какие-либо оставшиеся не вызванными обработчики типа before отсутствуют, поэтому переходим к шагу 3. На выполнение вызывается обработчик сообщений msg1 типа primary класса B, так как он является наиболее конкретным обработчиком типа primary. В нем не вызывается функция call-next-handler, поэтому обработчик msg1 типа primary класса A не выполняется и осуществляется переход к шагу 4. Разрешается завершение работы обработчика сообщений msg1 типа primary класса B, и он возвращает

символ msg1-B. На этом выполнение всех обработчиков типа primary заканчивается, поэтому переходим к шагу 5. В порядке, противоположном запуску обработчиков типа before, вначале разрешается выполнение обработчика msg1 типа after класса A и выполняется его возврат, поскольку он является наименее конкретным обработчиком типа after. Затем разрешается выполнение и возврат обработчика msg1 типа after класса B. Поскольку не остается больше каких-либо других не вызванных обработчиков типа after, переходим к шагу 6. Обработчику msg1 типа around класса A разрешается завершить свою работу. Последним действием, выполняемым этим обработчиком, становится вызов функции call-next-handler, поэтому возвращаемым значением становится значение следующего обработчика around или primary, который был вызван. В данном случае это был обработчик msg1 типа primary класса B, который имел возвращаемое значение msg1-B, поэтому возвращаемое значение данного обработчика around становится тем же самым. Теперь разрешается завершить свою работу обработчику msg1 типа around класса B. Его последним действием также был вызов функции call-next-handler, поэтому его возвращаемым значением становится возвращаемое значение обработчика msg1 типа around класса A, которое представляет собой msg1-B. Поскольку все обработчики типа around завершили свою работу, переходим к шагу 7. Окончательным возвращаемым значением команды send становится значение msg1-B, возвращенное обработчиком msg1 типа around класса B.

Для вывода на внешнее устройство списка обработчиков, применимых для обработки определенного сообщения, переданного экземпляру указанного класса, может использоваться **команда preview-send**. Она имеет следующий синтаксис:

```
(preview-send <defclass-name> <message-name>)
```

Например, вместо отслеживания работы обработчиков сообщений с помощью ключевого слова message-handlers и передачи сообщения msg1 в экземпляр b1 можно было бы воспользоваться следующей командой:

```
CLIPS> (preview-send B msg1)↵
>> msg1 around in class B
| >> msg1 around in class A
| | >> msg1 before in class B
| | << msg1 before in class B
| | >> msg1 before in class A
| | << msg1 before in class A
| | >> msg1 primary in class B
| | | >> msg1 primary in class A
| | | << msg1 primary in class A
| | << msg1 primary in class B
| | >> msg1 after in class A
```

```
| | << msg1 after in class A
| | >> msg1 after in class B
| | << msg1 after in class B
| << msg1 around in class A
<< msg1 around in class B
CLIPS>
```

Приведенные выше результаты показывают, что все применимые обработчики перечислены в том порядке, в каком они были бы вызваны, если бы в каждом обработчике типа `around` и `primary` вызывалась функция `call-next-handler`. Уровень отступа обозначает вложенность вызовов обработчиков. По аналогии с тем вариантом, когда осуществляется отслеживание работы обработчиков сообщений с помощью ключевого слова `message-handlers`, символы >> и << обозначают начало и конец выполнения кода обработчика. Вертикальные ряды знаков | связывают начальную и конечную части выполнения обработчиков, которые должны вызывать функцию `call-next-handler` для обеспечения выполнения обработчиков суперклассов. Обычно удобнее вызывать команду `preview-send`, чем использовать команду `watch message-handlers` и действительно передавать экземпляру сообщение. Первый способ позволяет проще ознакомиться со списком применимых обработчиков для данного сообщения и класса экземпляра. Тем не менее первый способ показывает, что́ могло быть вызвано, а второй способ позволяет ознакомиться с тем, что́ действительно было вызвано.

11.11 Создание экземпляра, инициализация и удаление обработчиков сообщений

Как уже было сказано выше в данной главе, в языке CLIPS предусмотрено несколько заранее определенных обработчиков сообщений, которые могут быть унаследованы от класса `USER`, в том числе обработчики сообщений **create**, **init** и **delete**. Обработчик сообщений `create` вызывается после создания экземпляра, но перед применением каких-либо предусмотренных по умолчанию значений или перекрытых значений слотов. Обработчик сообщений `init` вызывается после выполнения операций перекрытия значений слотов для задания всех оставшихся значений слотов, которые не были перекрыты своими значениями, заданными по умолчанию. А обработчик сообщений `delete` либо вызывается явно для удаления экземпляра, либо вызывается автоматически при вызове функции `make-instance` и указании в качестве имени экземпляра имени уже существующего экземпляра. Вообще говоря, при определении собственных классов не следует перекрывать обработчик типа `primary`, входящий в число указанных обработчиков сообщений, определенных в классе `USER`, но может оказаться весь-

ма полезной возможность определить собственные обработчики типа before и after, которые отвечали бы на существующие сообщения.

Вначале рассмотрим ситуацию, в которой может потребоваться определить обработчик init типа after. Предположим, что имеется следующая конструкция defclass:

```
(defclass PERSON
   (is-a USER)
   (slot first-name (type STRING)
   (access initialize-only))
   (slot middle-name (type STRING)
   (access initialize-only))
   (slot last-name (type STRING)
   (access initialize-only))
   (slot full-name (type STRING)
   (access initialize-only)))
```

При определении этой конструкции defclass с именем PERSON предусмотрено такое ограничение, что имя лица задается при создании экземпляра PERSON и в дальнейшем не изменяется. Слот full-name предназначен для хранения результатов конкатенации значений слотов first-name, middle-name и last-name с включением пробела между каждым компонентом полного имени. Итак, слот full-name может быть задан только во время инициализации, поэтому необходимо предоставлять для него значение наряду со значениями других компонентов полного имени, чтобы этот слот был заполнен правильно, как в следующем примере:

```
CLIPS>
(make-instance [p1] of PERSON
   (first-name "John")
   (middle-name "Quincy")
   (last-name "Public")
   (full-name "John Quincy Public"))↵
[p1]
CLIPS>
```

Тем не менее можно избавиться от необходимости задавать значение слота full-name, предусмотрев использование обработчика сообщений init типа after, который автоматически создает значение полного имени из значений других слотов, как показано ниже.

```
(defmessage-handler PERSON init after ()
   (bind ?self:full-name
   (str-cat ?self:first-name " "
```

```
   ?self:middle-name " "
   ?self:last-name)))
```

Отслеживая работу соответствующих обработчиков сообщений `init`, можно видеть, что обработчик сообщений `init` типа `after` с именем `PERSON` вызывается после обработчика сообщений `init` с именем `USER` и значение, сохраняемое в слоте `full-name`, принимает требуемый вид:

```
CLIPS> (watch message-handlers USER init)↵
CLIPS> (watch message-handlers PERSON init)↵
CLIPS>
(make-instance [p2] of PERSON
   (first-name "Jane")
   (middle-name "Paula")
   (last-name "Public"))↵
HND >> init primary in class USER
       ED:1 (<Instance-p2>)
HND << init primary in class USER
       ED:1 (<Instance-p2>)
HND >> init after in class PERSON
       ED:1 (<Instance-p2>)
HND << init after in class PERSON
       ED:1 (<Instance-p2>)
[p2]
CLIPS> (send [p2] get-full-name)↵
"Jane Paula Public"
CLIPS>
```

Следует отметить, что обычно не имеет особого смысла определять обработчик `init` типа `before`, поскольку ко времени его вызова все значения слотов экземпляра все еще остаются неинициализированными.

Атрибут `storage`

При изучении примера применения обработчика `create` типа `after` и обработчика `delete` типа `before` воспользуемся атрибутом слота **storage**. Если этому атрибуту присвоено значение **local**, которое является значением, предусмотренным по умолчанию, то каждый создаваемый экземпляр получает свою собственную область памяти, предназначенную для хранения значения слота. Если атрибуту `storage` присваивается значение **shared**, то все экземпляры данного класса совместно используют область памяти, предусмотренную для значения слота. Если значение слота изменяется для одного экземпляра, то данное изменение применяется во всех экземплярах. Например, ниже приведено объявление

класса, в котором содержится слот count с атрибутом storage, имеющим значение shared.

```
(defclass INSTANCE-COUNTER
   (is-a USER)
   (slot count (storage shared) (default 1)))
```

Если будут созданы два экземпляра, а затем изменено значение слота count в одном экземпляре, то, как показывает следующая последовательность команд, значение слота count изменится и в другом экземпляре:

```
CLIPS> (make-instance i1 of INSTANCE-COUNTER)↵
[i1]
CLIPS> (make-instance i2 of INSTANCE-COUNTER)↵
[i2]
CLIPS> (send [i1] get-count)↵
1
CLIPS> (send [i2] get-count)↵
1
CLIPS> (send [i1] put-count 2)↵
2
CLIPS> (send [i1] get-count)↵
2
CLIPS> (send [i1] get-count)↵
2
CLIPS>
```

С помощью такого средства можно создать обработчики сообщений create типа after и delete типа before, которые будут увеличивать значение слота count после создания каждого экземпляра данного класса и уменьшать значение слота count после уничтожения каждого экземпляра этого класса:

```
(defmessage-handler INSTANCE-COUNTER create
                        after ()
   (if (integerp ?self:count)
      then
      (send ?self put-count (+ ?self:count 1))))
(defmessage-handler INSTANCE-COUNTER delete
                        before ()
   (bind ?self:count (- ?self:count 1)))
```

Функционирование обработчика сообщений create типа after требует определенных пояснений. Сообщение create передается экземпляру до применения каких-либо заданных по умолчанию значений. В случае создания первого экземпляра INSTANCE-COUNTER слоту count в качестве значения присваивает-

ся символ nil во время вызова обработчика сообщений create. Попытка сложить единицу с этим значением привела бы к возникновению ошибки, поскольку символ nil — не числовой. Проверка наличия этой ситуации осуществляется с помощью функции if в обработчике сообщений create типа after, что позволяет предотвратить выполнение указанной операции сложения. А поскольку заданное по умолчанию значение совместно используемого слота применяется только один раз при создании первого экземпляра, то заданное по умолчанию значение слота count устанавливается равным 1, чтобы при создании первого экземпляра слот count автоматически приобретал правильное значение. При последующих вызовах обработчика create типа after будет обнаруживаться, что значение слота count является целочисленным, поэтому вызов предикативной функции во время проверки с помощью функции if завершится успешно и значение слота count увеличится на 1.

Ниже приведена последовательность команд, которая показывает, что в результате создания двух новых экземпляров класса INSTANCE-COUNTER значение слота count увеличится должным образом до 4, а после удаления одного экземпляра соответственно уменьшится до 3.

```
CLIPS> (make-instance i3 of INSTANCE-COUNTER)↵
[i3]
CLIPS> (make-instance i4 of INSTANCE-COUNTER)↵
[i4]
CLIPS> (send [i1] get-count)↵
4
CLIPS> (send [i4] delete)↵
TRUE
CLIPS> (send [i1] get-count)↵
3
CLIPS>
```

11.12 Модификация и дублирование экземпляров

В языке CLIPS предусмотрено несколько команд для работы с экземплярами, которые предоставляют функциональные возможности, аналогичные возможностям команд модификации и дублирования, предусмотренным для фактов. Эти команды имеют следующий синтаксис:

```
(modify-instance
   <instance-expression> <slot-overrides>*)
(message-modify-instance
```

```
   <instance-expression> <slot-overrides>*)
(active-modify-instance
   <instance-expression> <slot-overrides>*)
(active-message-modify-instance
   <instance-expression> <slot-overrides>*)
(duplicate-instance <instance-expression>
                    [to <instance-name-expression>]
                    <slot-overrides>*)
(message-duplicate-instance <instance-expression>
                    [to <instance-name-expression>]
                    <slot-overrides>*)
(active-duplicate-instance <instance-expression>
                    [to <instance-name-expression>]
                    <slot-overrides>*)
(active-message-duplicate-instance
                    <instance-expression>
                    [to <instance-name-expression>]
                    <slot-overrides>*)
```

В этих определениях терм <instance-expression> обозначает экземпляр, который должен быть модифицирован или дублирован, терм <slot-overrides> содержит список модификаций слотов, а что касается команд дублирования, то терм <instance-name-expression> представляет собой необязательное новое имя для дублируемого экземпляра.

Наиболее важной командой модификации экземпляра является **команда modify-instance**. Ниже приведен пример ее использования.

```
CLIPS> (unwatch all)↵
CLIPS> (clear)↵
CLIPS>
(defclass PERSON
   (is-a USER)
   (slot first-name)
   (slot last-name))
CLIPS>
(make-instance [p1] of PERSON
   (first-name "Jeff")
   (last-name "Public"))↵
[p1]
CLIPS> (watch messages)↵
CLIPS> (watch slots)↵
CLIPS>
```

```
(modify-instance [p1] (first-name "Jack")
                                (last-name "Private"))↵
MSG >> direct-modify ED:1 (<Instance-p1>
                           <Pointer-0x0062b200>)
::= local slot first-name in instance p1 <- "Jack"
::= local slot last-name in instance p1 <- "Private"
MSG << direct-modify ED:1 (<Instance-p1>
                           <Pointer-0x0062b200>)
TRUE
CLIPS> (unwatch all)↵
CLIPS> (send [p1] print)↵
[p1] of PERSON
(first-name "Jack")
(last-name "Private")
CLIPS>
```

Обратите внимание на то, что сообщение **direct-modify** передается экземпляру `[p1]`, а сообщения `put-first-name` и `put-last-name` не передаются. От имени программиста автоматически создается еще один определяемый системой обработчик сообщений — обработчик сообщений `direct-modify`. Если используется команда `modify-instance`, то значения слотов изменяются непосредственно с помощью обработчика сообщений `direct-modify`, поэтому средства передачи сообщений не вызываются. Следствием указанного положения дел является то, что не вызываются обработчики `put-` типов `primary`, `after`, `before` и `around`, связанные со слотом.

Команда message-modify-instance имеет такой же синтаксис параметров, как и команда `modify-instance`, но при ее использовании изменение значений слотов осуществляется с помощью средств передачи сообщений, например, как показано ниже.

```
CLIPS> (watch messages)↵
CLIPS>
(message-modify-instance [p1]
   (first-name "Jeff")
   (last-name "Public"))↵
MSG >> message-modify ED:1 (<Instance-p1>
                            <Pointer-0x0062b1c0>)
MSG >> put-first-name ED:2 (<Instance-p1> "Jeff")
MSG << put-first-name ED:2 (<Instance-p1> "Jeff")
MSG >> put-last-name ED:2 (<Instance-p1> "Public")
MSG << put-last-name ED:2 (<Instance-p1> "Public")
MSG << message-modify ED:1 (<Instance-p1>
```

```
                                <Pointer-0x0062b1c0>)
TRUE
CLIPS>
```

В данном случае применялась команда `message-modify-instance`, поэтому для изменения значений слотов использовались обработчики сообщений `put-first-name` и `put-last-name`. А для модификации экземпляра вместо `direct-modify` применялся еще один определяемый системой обработчик сообщений — **message-modify**.

Наконец, предусмотрены две дополнительные команды модификации экземпляра, обеспечивающие более широкий контроль над сопоставлением с шаблоном объекта, — **active-modify-instance** и **active-message-modify-instance**. И в этом случае синтаксис параметров является таким же, как и в других командах модификации. При использовании команд `modify-instance` и `message-modify-instance` сопоставление с шаблоном объекта не происходит до тех пор, пока не будет завершена обработка всех изменений в слоте. Поэтому при использовании команд `active-modify-instance` и `active-message-modify-instance` сопоставление с шаблоном объекта осуществляется только после того, как будут внесены изменения в каждый слот. Как правило, действительно требуется, чтобы все модификации слотов в экземпляре были выполнены до сопоставления с шаблоном объекта, поскольку обычно это позволяет повысить производительность, но на тот случай, если потребуется дополнительная обработка, предусмотрены указанные команды активизированной модификации.

Кроме того, в языке CLIPS имеются четыре аналогичные команды, предназначенные для дублирования экземпляров: **duplicate-instance**, **message-duplicate-instance**, **active-duplicate-instance** и **active-message-modify-instance**. Каждая из этих команд имеет такой же принцип действия и состав параметров, как и команды модификации, имеющие подобные имена. Но с помощью этих команд вместо модификации экземпляра, переданного в качестве параметра, создается дубликат экземпляра, к которому применяются операции перекрытия значений слотов. Кроме того, в качестве необязательного параметра в командах дублирования может быть задано имя дублируемого экземпляра; в противном случае такое имя создается автоматически. Возвращаемым значением этих команд является дублированный экземпляр. Ниже приведены примеры, в которых показано, как используются две из указанных команд дублирования.

```
CLIPS> (duplicate-instance [p1]
   (first-name "Jack"))↵
MSG >> direct-duplicate ED:1 (<Instance-p1> [gen2]
                              <Pointer-0x006b8840>)
MSG << direct-duplicate ED:1 (<Instance-p1> [gen2]
                              <Pointer-0x006b8840>)
```

```
[gen2]
CLIPS> (message-duplicate-instance [p1] to [p3]
   (first-name "Jill"))↵
MSG >> message-duplicate ED:1 (<Instance-p1> [p3]
                               <Pointer-0x006b8840>)
MSG >> create ED:2 (<Instance-p3>)
MSG << create ED:2 (<Instance-p3>)
MSG >> put-first-name ED:2 (<Instance-p3> "Jill")
MSG << put-first-name ED:2 (<Instance-p3> "Jill")
MSG >> put-last-name ED:2 (<Instance-p3> "Public")
MSG << put-last-name ED:2 (<Instance-p3> "Public")
MSG >> init ED:2 (<Instance-p3>)
MSG << init ED:2 (<Instance-p3>)
MSG << message-duplicate ED:1 (<Instance-p1> [p3]
                               <Pointer-0x006b8840>)
[p3]
CLIPS> (instances)↵
[p1] of PERSON
[gen2] of PERSON
[p3] of PERSON
For a total of 3 instances.
CLIPS>
```

В примере использования команды `duplicate-instance` имя экземпляра не задано, поэтому применяется имя, сформированное системой, `[gen2]`. А в примере использования команды `message-duplicate` для дублированного экземпляра служит указанное имя, `[p3]`. Двум другим определяемым системой обработчикам сообщений, **direct-duplicate** и **message-duplicate**, передаются сообщения, зависящие от того, какая команда дублирования экземпляра была вызвана. В случае вызова команд `message-duplicate` и `active-message-duplicate` передаются дополнительные сообщения `create`, `init` и `put-` для копирования и перекрытия значений слотов во вновь созданном экземпляре.

Наконец, предусмотрена возможность определить обработчики сообщений типа `around`, `before` и `after` для определяемых системой обработчиков сообщений `direct-modify`, `message-modify`, `direct-duplicate` и `message-duplicate`. Тем не менее, поскольку значения перекрытий слотов передаются как тип `EXTERNAL-ADDRESS`, а такие параметры могут быть декодированы только с помощью внешнего кода, то программист, определяя свои собственные обработчики, вряд ли сможет добиться каких-либо продуктивных результатов.

11.13 Классы и универсальные функции

Ограничения параметров для универсальных функций не сводятся к использованию только заранее определенных типов, предусмотренных в языке CLIPS. В качестве ограничений параметров могут также использоваться определяемые пользователем классы. Например, следующий класс может служить для представления комплексного числа:

```
(defclass COMPLEX
   (is-a USER)
   (slot real)
   (slot imaginary))
```

Класс COMPLEX представляет комплексные числа в таком формате:

```
a + bi
```

В этом определении a и b — вещественные числа; i — квадратный корень от -1. В классе COMPLEX значение a хранится в слоте real, а значение b — в слоте imaginary. Операция сложения комплексных чисел выполняется просто:

```
(a + bi) + (c + di) = (a + c) + (b + d)i
```

Ниже приведено определение метода, который перегружает функцию + в целях реализации операции сложения двух комплексных чисел.

```
(defmethod + ((?c1 COMPLEX) (?c2 COMPLEX))
   (make-instance of COMPLEX
   (real (+ (send ?c1 get-real)
   (send ?c2 get-real)))
   (imaginary (+ (send ?c1 get-imaginary)
   (send ?c2 get-imaginary)))))
```

Все действия, выполняемые этим методом, сводятся к тому, что складываются части real комплексных чисел, а затем части imaginary комплексных чисел, после чего они сохраняются во вновь созданном экземпляре класса COMPLEX. Приведенные ниже примеры создания экземпляров класса COMPLEX и сложения представленных ими комплексных чисел показывают, что данный метод работает правильно.

```
CLIPS> (make-instance c1 of COMPLEX
       (real 3) (imaginary 4))↵
[c1]
CLIPS> (make-instance c2 of COMPLEX
       (real 5) (imaginary 6))↵
[c2]
CLIPS> (+ [c1] [c2])↵
[gen1]
```

```
CLIPS> (send [gen321] print)↵
[gen1] of COMPLEX
(real 8)
(imaginary 10)
CLIPS>
```

11.14 Функции запроса множества экземпляров

В языке CLIPS не только предусмотрены возможности применять операции сопоставления с шаблонами к объектам, но допускается непосредственная передача запросов системе COOL, касающихся множеств экземпляров, которые удовлетворяют заданному множеству условий. Для иллюстрации применения этих функций будут служить следующие конструкции, представляющие небольшое генеалогическое дерево:

```
(defclass PERSON
   (is-a USER)
   (slot full-name)
   (slot gender)
   (multislot children))
(defclass FEMALE
   (is-a PERSON)
   (slot gender (access read-only)
   (storage shared)
   (default female)))
(defclass MALE
   (is-a PERSON)
   (slot gender (access read-only)
   (storage shared)
   (default male)))
(definstances people
   ([p1] of MALE (full-name "John Smith")
            (children [p5]))
   ([p2] of FEMALE (full-name "Jan Smith")
            (children [p5]))
   ([p3] of MALE (full-name "Bob Jones")
            (children [p6]))
   ([p4] of FEMALE (full-name "Pam Jones")
            (children [p6]))
```

```
   ([p5] of MALE (full-name "Frank Smith")
            (children [p7]))
   ([p6] of FEMALE (full-name "Sue Jones")
            (children [p7]))
   ([p7] of MALE (full-name "Dave Smith")))
```

Определение того, успешно ли выполнен запрос

Простейшей из функций запроса является **функция any-instancep**, которая имеет следующий синтаксис:

```
(any-instancep <instance-set-template> <query>)
```

В данном определении терм <instance-set-template> представляет собой спецификацию классов, в которых должен осуществляться поиск, а терм <query> — это булево выражение, которому должны соответствовать согласующиеся с запросом экземпляры этих классов. Если обнаруживается непустое множество экземпляров, удовлетворяющих запросу, то функция any-instancep возвращает символ TRUE; в противном случае она возвращает символ FALSE.

Терм <instance-set-template> имеет следующий синтаксис:

```
(<instance-set-member-template>+)
```

В этом определении терм <instance-set-member-template> имеет такую форму:

```
(<single-field-variable> <class-name-expression>+)
```

Терм <single-field-variable> представляет собой переменную, с которой должен быть связан экземпляр, а одно или несколько вхождений выражения <class-name-expression> — это классы, содержащие экземпляры, которые по отдельности должны быть связаны с переменной <single-field-variable>.

Наиболее простая проверка, которая может быть выполнена с помощью функции any-instancep, состоит в определении того, существует ли какой-либо экземпляр некоторого класса:

```
CLIPS> (any-instancep ((?p PERSON)) TRUE)↵
FALSE
CLIPS> (reset)↵
CLIPS>
(any-instancep ((?p PERSON)) TRUE)↵
TRUE
CLIPS>
```

В данном примере какие-либо экземпляры класса PERSON не обнаруживаются до тех пор, пока не выполняется команда reset и не создаются экземпляры

в конструкциях definstances с именем people. В качестве запросе используется символ TRUE, а это означает, что запросу удовлетворяет любой экземпляр, присвоенный переменной ?p.

Если бы создавались только экземпляры MALE и FEMALE, то можно было бы определить, есть ли какие-либо экземпляры класса PERSON, просто указав эти два класса:

```
CLIPS> (any-instancep ((?p MALE FEMALE)) TRUE)↵
TRUE
CLIPS>
```

С другой стороны, наличие экземпляров класса MALE можно определить с помощью любого из следующих трех запросов:

```
CLIPS> (any-instancep ((?p MALE)) TRUE)↵
TRUE
CLIPS>
(any-instancep ((?p PERSON))
   (eq (send ?p get-gender) male))↵
TRUE
CLIPS>
(any-instancep ((?p PERSON))
   (eq ?p:gender male))↵
TRUE
CLIPS>
```

В первом случае проводится поиск экземпляров класса MALE, в отношении которых известно, что слот gender содержит значение male. Во втором случае экземпляру PERSON, присвоенному переменной ?p, передается сообщение get-gender и используется функция eq для определения того, равно ли полученное значение символу male. В третьем случае показано, как можно использовать сокращенное обозначение слота с переменными, связанными в шаблоне множества экземпляров.

В шаблоне может быть также задано несколько элементов множества экземпляров примерно таким образом:

```
CLIPS> (any-instancep ((?f FEMALE) (?p PERSON))
   (member$ ?p ?f:children))↵
TRUE
CLIPS>
```

В данном примере проверяется, являются ли какие-либо представительницы класса FEMALE (Женщина) матерями. К каждой комбинации экземпляров класса FEMALE с экземплярами класса PERSON применяется запрос для определения то-

го, является ли экземпляр PERSON ?p одним из детей (children) лица женского пола, данные о котором представлены с помощью экземпляра FEMALE ?f.

Определение экземпляров, удовлетворяющих запросу

Функции запроса **find-instance** и **find-all-instances** имеют синтаксис, подобный синтаксису функции any-instancep:

```
(find-instance <instance-set-template> <query>)
(find-all-instances <instance-set-template> <query>)
```

Но функция запроса find-instance возвращает не символ TRUE или FALSE, а многозначное значение, содержащее первое множество экземпляров, удовлетворяющее запросу. Если не обнаруживается множество экземпляров, удовлетворяющее запросу, то возвращаемое многозначное значение остается пустым. Аналогичным образом, функция find-all-instances возвращает многозначное значение, содержащее все множества экземпляров, удовлетворяющих запросу, например, как показано ниже.

```
CLIPS> (find-instance ((?p MALE)) TRUE)↵
([p1])
CLIPS> (find-all-instances ((?p MALE)) TRUE)↵
([p1] [p3] [p5] [p7])
CLIPS>
(find-all-instances ((?p MALE))
                    (eq ?p:gender female))↵
()
CLIPS>
```

В том случае, если в шаблоне задано несколько элементов множества, необходимо обеспечить группирование элементов в возвращаемом значении программным путем, например, как в следующем диалоге:

```
CLIPS>
(find-all-instances ((?f FEMALE) (?p PERSON))
                    (member$ ?p ?f:children))↵
([p2] [p5] [p4] [p6] [p6] [p7])
CLIPS>
```

Экземпляры [p2] и [p5] принадлежат к первому множеству экземпляров, экземпляры [p4] и [p6] — ко второму множеству экземпляров, а экземпляры [p6] и [p7] — к третьему множеству экземпляров. Ниже приведена конструкция deffunction, которая показывает, как можно сгруппировать элементы множеств экземпляров программным путем.

```
(deffunction print-mother-message (?query-result)
```

```
   (bind ?iterations
   (div (length$ ?query-result) 2))
(loop-for-count (?i ?iterations) do
   (bind ?mother (nth$
      (- (* 2 ?i) 1) ?query-result))
   (bind ?child (nth$ (* 2 ?i) ?query-result))
   (printout t (send ?mother get-full-name)
         " is the mother of "
         (send ?child get-full-name)
         "." crlf)))
```

Длина возвращаемого значения запроса делится на количество экземпляров в расчете на каждое множество элементов для определения количества множества элементов, подлежащего обработке в цикле. Затем используется функция `loop-for-count` в сочетании с функцией `nth$` для извлечения значения для каждого экземпляра в множестве элементов. С помощью сообщения `print-mother-message` с возвращаемым значением, равным результату предыдущего вызова `find-all-instances`, вырабатываются следующие результаты:

```
CLIPS>
(print-mother-message
   (find-all-instances ((?f FEMALE) (?p PERSON))
   (member$ ?p ?f:children))).↵
Jan Smith is the mother of Frank Smith.
Pam Jones is the mother of Sue Jones.
Sue Jones is the mother of Dave Smith.
FALSE
CLIPS>
```

Выполнение действий над экземплярами, удовлетворяющими запросу

Функции запроса **do-for-instance**, **do-for-all-instances** и **delayed-do-for-all-instances** позволяют выполнять необходимые действия над множествами экземпляров, удовлетворяющих запросу. Эти функции имеют такой синтаксис:

```
(do-for-instance
   <instance-set-template> <query> <expression>*)
(do-for-all-instances
   <instance-set-template> <query> <expression>*)
(delayed-do-for-all-instances
   <instance-set-template> <query> <expression>*)
```

Функция do-for-instance применяет указанные действия к первому множеству экземпляров, удовлетворяющих запросу, например, таким образом:

```
CLIPS>
(do-for-instance ((?f FEMALE) (?p PERSON))
            (member$ ?p ?f:children)
   (printout t ?f:full-name " is the mother of "
               ?p:full-name "." crlf))↵
Jan Smith is the mother of Frank Smith.
CLIPS>
```

Функция do-for-all-instances выполняет указанные действия над всеми множествами экземпляров, удовлетворяющих запросу, например, следующим образом:

```
CLIPS>
(do-for-all-instances ((?f FEMALE) (?p PERSON))
            (member$ ?p ?f:children)
   (printout t ?f:full-name " is the mother of "
               ?p:full-name "." crlf))↵
Jan Smith is the mother of Frank Smith.
Pam Jones is the mother of Sue Jones.
Sue Jones is the mother of Dave Smith.
CLIPS>
```

Следует отметить, что с помощью функции do-for-all-instances может быть создан более простой механизм, позволяющий добиться тех же функциональных возможностей, что и в предыдущем примере, в котором применялась функция find-all-instances с конструкцией deffunction с именем print-mother-message.

Функция delayed-do-for-all-instances аналогична функции do-for-all-instances, за исключением того, что в ней вначале отыскиваются все множества экземпляров, удовлетворяющие запросу, а затем выполняются указанные действия над всеми множествами экземпляров. В отличие от этого, при использовании функции do-for-all-instances запрос применяется к каждому множеству экземпляров, и в случае успешного выполнения проверки, предусмотренной в запросе, указанные действия применяются к полученному множеству экземпляров. Единственная ситуация, в которой эти функции вырабатывают разные результаты, возникает тогда, когда выполняемые действия изменяют результаты запроса в множестве экземпляров, которое еще не было обработано. Например, разные результаты применения этих двух функций могут быть вызваны изменением значения слота экземпляра, на который распространяется данный запрос.

Если ни одно множество экземпляров не удовлетворяет запросу, то возвращаемым значением всех функций запроса, допускающих применение действий, становится FALSE. В противном случае возвращаемым значением являются результаты последнего действия, выполненного над последним множеством экземпляров, удовлетворяющим запросу. Кроме того, функции запроса, допускающие применение действий, могут закончить свою работу преждевременно, если в них используются функция `break` или `return`.

11.15 Множественное наследование

Во всех примерах применения конструкции `defclass`, которые рассматривались до сих пор в данной главе, в атрибуте `is-a` указан единственный класс. Все эти примеры характеризуются применением **единичного наследования**. Но в качестве значения атрибута `is-a` допускается задавать больше одного класса. В таком случае происходит так называемое **множественное наследование**. Но в тех случаях, когда указанные классы не имеют общих слотов или обработчиков сообщений, множественное наследование по существу эквивалентно единичному наследованию, в котором используются суперклассы определяемого класса. Например, в следующем примере слоты `x`, `y` и `z` наследуются непосредственно от класса `D` с применением множественного наследования:

```
CLIPS> (clear)↵
CLIPS>
(defclass A
   (is-a USER)
   (slot x))↵
CLIPS>
(defclass B
   (is-a USER)
   (slot y))↵
CLIPS>
(defclass C
   (is-a USER)
   (slot z))↵
CLIPS>
(defclass D
   (is-a C B A))↵
CLIPS> (make-instance [d1] of D (x 3) (y 4) (z 5))↵
[d1]
CLIPS> (send [d1] print)↵
[d1] of D
```

```
(x 3)
(y 4)
(z 5)
CLIPS>
```

В следующем примере классы B и C используются в промежуточных классах, чтобы обеспечить возможность косвенного наследования классом D слотов x и y по принципу единичного наследования:

```
CLIPS> (clear)↵
CLIPS>
(defclass A
   (is-a USER)
   (slot x))↵
CLIPS>
(defclass B
   (is-a A)
   (slot y))↵
CLIPS>
(defclass C
   (is-a B)
   (slot z))↵
CLIPS>
(defclass D
   (is-a C))↵
CLIPS> (make-instance [d1] of D (x 3) (y 4) (z 5))↵
[d1]
CLIPS> (send [d1] print)↵
[d1] of D
(x 3)
(y 4)
(z 5)
CLIPS>
```

Важно отметить, что в обоих случаях класс D наследует одни и те же слоты, но к классам B и C это не относится. Во втором примере существовала возможность переопределить классы B и C, поэтому мог быть достигнут тот же окончательный результат по созданию класса D с использованием единичного наследования вместо множественного наследования. Тем не менее множественное наследование применяется в тех ситуациях, когда имеются заранее заданные классы, которые невозможно модифицировать.

Конфликты, связанные с множественным наследованием

Большинство практических примеров использования множественного наследования относится к такой категории, что суперклассы, от которых производный класс наследует свои свойства, не имеют общих слотов или обработчиков сообщений (отличных от тех, которые унаследованы от класса USER). В подобных ситуациях между суперклассами не возникают конфликты, которые приходилось бы разрешать, а определяемые слоты и обработчики сообщений, которые конфликтуют со слотами и обработчиками сообщений суперкласса, перекрывают определения суперкласса так же, как и при единичном наследовании.

Рассмотрим краткий пример, в котором используется множественное наследование и обнаруживаются конфликтующие определения слотов и обработчиков сообщений в суперклассах:

```
CLIPS> (clear)↵
CLIPS>
(defclass A
   (is-a USER)
   (slot x (default 3))
   (slot y (default 4)))↵
CLIPS>
(defmessage-handler A compute ()
   (* ?self:y 10))↵
CLIPS>
(defclass B
   (is-a USER)
   (slot y (default 1))
   (slot z (default 5)))↵
CLIPS>
(defmessage-handler B compute ()
   (+ ?self:y 3))↵
CLIPS> (defclass C (is-a A B))↵
CLIPS> (defclass D (is-a B A))↵
CLIPS>
```

В этом примере классы C и D непосредственно наследуют свои свойства от классов A и B. Но в классах A и B имеются конфликтующие определения слота y и обработчика сообщений compute, поэтому рассмотрим, как разрешается этот конфликт:

```
CLIPS> (make-instance [c1] of C)↵
[c1]
CLIPS> (make-instance [d1] of D)↵
```

```
[d1]
CLIPS> (send [c1] print)↵
[c1] of C
(z 5)
(x 3)
(y 4)
CLIPS> (send [d1] print)↵
[d1] of D
(x 3)
(y 1)
(z 5)
CLIPS>
```

Анализ показанных результирующих значений слотов, относящихся к экземплярам `[c1]` и `[c2]`, позволяет сделать два замечания. Первое из них касается порядка, в котором происходит вывод значений слотов, а второе касается значения слота `y`. Порядок, в котором выводятся значения слотов, не играет особой роли и просто показывает, что на результирующее определение класса оказывает некоторое влияние та последовательность, в которой имена суперклассов перечислены в атрибуте `is-a`. Более существенное различие между двумя экземплярами состоит в том, что заданное по умолчанию значение слота `y` для экземпляра `[c1]` равно `4`, тогда как заданное по умолчанию значение слота `y` для экземпляра `[d1]` равно `1`. Поскольку класс `C` вначале наследует свои свойства от класса `A`, то в нем используется значение слота `y`, заданное по умолчанию в классе `A`, а не значение, заданное по умолчанию в классе `B`. Аналогичным образом, класс `D` в первую очередь наследует свои свойства от класса `B`, поэтому в нем применяется значение слота `y`, заданное по умолчанию в классе `B`, а не значение, заданное по умолчанию в классе `A`. Аналогичные действия осуществляются при определении обработчика сообщений `compute`:

```
CLIPS> (send [c1] put-y 3)↵
3
CLIPS> (send [d1] put-y 3)↵
3
CLIPS> (send [c1] compute)↵
30
CLIPS> (send [d1] compute)↵
6
CLIPS>
```

Даже несмотря на то, что слоту `y` в обоих экземплярах присвоено значение `3`, результат обработки сообщения `compute` для экземпляра `[c1]` равен `30`, т.е. $(3 * 10)$, а результат для экземпляра `[d1]` равен `6`, т.е. $(3 + 3)$. Еще раз от-

метим, что причина этого состоит в том, что в экземпляре [c1] используется обработчик сообщений compute для класса A, а в экземпляре [d1] — обработчик сообщений compute для класса B.

В тех простых случаях, в которых классы, заданные в атрибуте is-a, не имеют общих определяемых пользователем суперклассов, приоритет, используемый при наличии нескольких определений одного и того же слота или обработчика сообщений, определяется последовательностью задания классов в указанном атрибуте. В данном примере определения класса A имеют более высокий приоритет, чем определения класса B во время обработки объявления класса C, поскольку A указан перед B. A в классе D определения класса B имеют более высокий приоритет над определениями класса A, поскольку B был задан перед A.

Задачу изучения более сложных случаев применения множественного наследования оставляем читателю. Полное описание проблематики множественного наследования наряду с примерами того, как происходит разрешение конфликтов, связанных с множественным наследованием во всех ситуациях, приведено в документе *Basic Programming Guide*, который находится на компакт-диске, прилагаемом к данной книге. Но в качестве общего принципа следует указать, что если в программе создаются классы с использованием множественного наследования и от того порядка, в котором заданы классы в атрибуте is-a, зависит поведение определяемого класса, то можно предположить, что в программе применяется более сложное решение рассматриваемой проблемы, чем могло бы быть.

Сохранение и восстановление значений слотов

Рассмотрим практический пример применения множественного наследования — определение класса RESTORABLE, который может использоваться с другими классами для сохранения и восстановления значений слотов экземпляра. Для реализации этого класса необходимо определить еще один класс, который будет служить для хранения значений слотов:

```
(defclass SAVED-SLOT
   (is-a USER)
   (slot slot-name)
   (multislot slot-value))
```

Класс SAVED-SLOT имеет слоты slot-name и slot-value. Слот slot-name используется для хранения имени сохраненного слота, а слот slot-value применяется для хранения значения указанного слота. Он определен как многозначный слот, multislot, поскольку необходимо предусмотреть возможность хранения значений слотов, полученных как из однозначных, так и из многозначных слотов.

Класс RESTORABLE определен следующим образом:

```
(defclass RESTORABLE
   (is-a USER)
   (multislot saved-slots))
```

Многозначный слот `saved-slots` предназначен для хранения от нуля и больше ссылок на экземпляры `SAVED-SLOT`, представляющие значения сохраненных слотов экземпляра.

Для сохранения значений слотов экземпляра `RESTORABLE` используется следующий обработчик сообщений `save`:

```
(defmessage-handler RESTORABLE save ()
   ; Удалить существующие сохраненные слоты
   (progn$ (?si ?self:saved-slots)
      (send ?si delete))
   (bind ?self:saved-slots)
   ; Составить список слотов
   (bind ?class (class ?self))
   (bind ?slots
      (delete-member$ (class-slots ?class inherit)
                      saved-slots))
   ; Создать пустой список
   (bind ?list (create$))
   ; Обработать в цикле каждый слот
   (progn$ (?slot ?slots)
      (bind ?value (send ?self
                     (sym-cat get- ?slot)))
      (bind ?ins (make-instance of SAVED-SLOT
                     (slot-name ?slot)
                     (slot-value ?value)))
      (bind ?list (create$ ?list ?ins)))
   ; Записать сохраненные слоты в память
   (bind ?self:saved-slots ?list))
```

Первое действие, выполняемое обработчиком сообщений `save`, состоит в том, что он удаляет все экземпляры `SAVED-SLOT`, хранимые в слоте `saved-slots`, на которые имеются ссылки. Затем слот `saved-slots` связывается с пустым многозначным значением.

После этого формируется список слотов, подлежащих сохранению. Для определения имени класса данного экземпляра вызывается **функция class**. Затем имя класса передается в **функцию class-slots** наряду с ключевым словом `inherit` для получения многозначного списка, содержащего все имена слотов, связанные с данным классом. Наконец, имя слота `saved-slots` удаляется из этого списка с помощью вызова **функции delete-member$**. Этот слот не должен применять-

ся для сохранения рассматриваемых значений, поскольку он предназначен для использования в качестве области памяти для значений всех других слотов.

После этого создается пустой список, который должен содержать экземпляры SAVED-SLOT. Для обработки в цикле всех слотов сохраняемого экземпляра используется функция progn$. Прежде всего осуществляется выборка значения слота. Имя слота добавляется к символу get- для создания соответствующего сообщения, передаваемого экземпляру в целях выборки значения слота. Вслед за выборкой значения слота создается экземпляр SAVED-SLOT, содержащий имя и значение слота. Ссылка на этот экземпляр добавляется к списку сохраненных слотов. А после обработки всех слотов список ссылок на экземпляры SAVED-SLOT сохраняется в слоте saved-slots.

Теперь, после того как определен обработчик сообщений save, можно воспользоваться следующим обработчиком сообщений restore для выборки значений слотов экземпляра:

```
(defmessage-handler RESTORABLE restore ()
   (progn$ (?si ?self:saved-slots)
   (bind ?name (send ?si get-slot-name))
   (bind ?value (send ?si get-slot-value))
   (send ?self (sym-cat put- ?name) ?value)))
```

Обработчик сообщений restore обрабатывает в цикле все экземпляры SAVED-SLOT, хранимые в слоте saved-slots. Из экземпляра SAVED-SLOT вначале осуществляется выборка имени и значения каждого сохраненного слота. Имя слота добавляется к символу put- для создания соответствующего сообщения, предназначенного для передачи экземпляру в целях присваивания значения слота. Затем это сообщение передается экземпляру для восстановления сохраненного значения в качестве значения слота.

Теперь, после определения класса RESTORABLE, рассмотрим, как можно было бы его использовать вместе с существующим классом для создания нового класса, обладающего функциональными возможностями сохранения и восстановления. Для этого воспользуемся классом PERSON, аналогичным рассматриваемому в предыдущих примерах:

```
(defclass PERSON
   (is-a USER)
   (slot full-name)
   (slot gender)
   (multislot children))
```

Определим следующий класс RESTORABLE-PERSON, который наследует свои свойства от обоих классов, RESTORABLE и PERSON:

```
(defclass RESTORABLE-PERSON
   (is-a RESTORABLE PERSON))
```

Для ознакомления с тем, как реализуются функциональные возможности сохранения и восстановления, вначале необходимо создать экземпляр класса RESTORABLE-INSTANCE:

```
CLIPS> (reset)↵
CLIPS>
(make-instance [p1] of RESTORABLE-PERSON
   (full-name "Sue Jones")
   (gender female)
   (children Bob Jan))↵
[p1]
CLIPS> (send [p1] print)↵
[p1] of RESTORABLE-PERSON
(full-name "Sue Jones")
(gender female)
(children Bob Jan)
(saved-slots)
CLIPS>
```

После передачи экземпляру [p1] сообщения save сохраняются текущие значения его слотов:

```
CLIPS> (send [p1] save)↵
([gen1] [gen2] [gen3])
CLIPS> (send [p1] print)↵
[p1] of RESTORABLE-PERSON
(full-name "Sue Jones")
(gender female)
(children Bob Jan)
(saved-slots [gen1] [gen2] [gen3])
CLIPS>
```

Экземпляры [gen1], [gen2] и [gen3] представляют собой экземпляры класса SAVED-SLOT, созданные для сохранения значений слотов экземпляра [p1]. Рассмотрим содержимое этих экземпляров, чтобы ознакомиться с тем, как хранятся отдельные значения слотов:

```
CLIPS> (send [gen1] print)↵
[gen1] of SAVED-SLOT
(slot-name full-name)
(slot-value "Sue Jones")
CLIPS> (send [gen2] print)↵
```

```
[gen2] of SAVED-SLOT
(slot-name gender)
(slot-value female)
CLIPS> (send [gen3] print)↵
[gen3] of SAVED-SLOT
(slot-name children)
(slot-value Bob Jan)
CLIPS>
```

Затем изменим некоторые значения слотов экземпляра [p1] следующим образом:

```
CLIPS> (send [p1] put-full-name "Sue Smith")↵
"Sue Smith"
CLIPS> (send [p1] put-children Bob Jan Paul)↵
(Bob Jan Paul)
CLIPS> (send [p1]
print)↵
[p1] of RESTORABLE-PERSON
(full-name "Sue Smith")
(gender female)
(children Bob Jan Paul)
(saved-slots [gen1] [gen2] [gen3])
CLIPS>
```

После передачи экземпляру сообщения restore восстанавливаются первоначальные значения слотов:

```
CLIPS> (send [p1] restore)↵
(Bob Jan)
CLIPS> (send [p1] print)↵
[p1] of RESTORABLE-PERSON
(full-name "Sue Jones")
(gender female)
(children Bob Jan)
(saved-slots [gen1] [gen2] [gen3])
CLIPS>
```

11.16 Конструкции defclass и defmodule

Конструкции defclass могут импортироваться и экспортироваться модулями по такому же принципу, как и другие конструкции. Дело в том, что к конструкциям defclass также могут применяться описанные выше в данной книге

операторы export и import, которые экспортируют или импортируют все конструкции. Кроме того, предусмотрена возможность явно указывать, какие конструкции defclass должны быть экспортированы или импортированы, с использованием одного из следующих операторов:

```
(export defclass ?ALL)
(export defclass ?NONE)
(export defclass <deffunction-name>+)
(import <module-name> defclass ?ALL)
(import <module-name> defclass ?NONE)
(import <module-name> defclass <defclass-name>+)
```

При импортировании или экспортировании класса импортируются или экспортируются также все связанные с ним конструкции defmessage-handler. Но импорт или экспорт только определенных обработчиков сообщений невозможен. Если в модуле не импортируется какая-то конструкция defclass из другого модуля, то в нем существует возможность создать собственное определение этого класса с помощью того же имени класса. Исключением из этого правила являются классы, заранее определенные в системе, которые являются видимыми во всех модулях и не могут быть переопределены, такие как класс USER.

Безусловно, факты связаны с тем модулем, в котором определены относящиеся к ним конструкции deftemplate, а экземпляры связаны с тем модулем, в котором определены относящиеся к ним конструкции defclass. Тем не менее каждый модуль имеет свое собственное "пространство имен", позволяющее обеспечить уникальность имен экземпляров. Это означает, что в одном модуле не могут существовать два экземпляра, совместно использующие одно и то же имя экземпляра, а в двух разных модулях могут присутствовать экземпляры, имеющие одинаковое имя экземпляра:

```
CLIPS> (defmodule A)↵
CLIPS>
(defclass A::ACLASS (is-a USER)
   (export defclass ?ALL))↵
CLIPS> (make-instance [same] of ACLASS)↵
[same]
CLIPS> (send [same] print)↵
[same] of A::ACLASS
CLIPS> (defmodule B)↵
CLIPS>
(defclass B::BCLASS (is-a USER)
   (export defclass ?ALL))↵
CLIPS> (make-instance [same] of BCLASS)↵
[same]
```

```
CLIPS> (send [same] print)↵
[same] of B::BCLASS
CLIPS> (set-current-module A)↵
B
CLIPS> (send [same] print)↵
[same] of A::ACLASS
CLIPS>
```

Обратите внимание на то, что определение экземпляра [same] в модуле B не вызовет удаления экземпляра [same] в модуле A, как это обычно происходит, если в модуле A создается экземпляр с тем же именем.

Как показывает следующий пример, пространство имен экземпляров экспортирующего модуля не становится непосредственно видимым для импортирующего модуля:

```
CLIPS> (defmodule C (import A defclass ?ALL)
   (import B defclass ?ALL))↵
CLIPS> (send [same] print)↵
[MSGPASS2] No such instance same in function send.
FALSE
CLIPS>
```

В данном случае экземпляр [same] имеется и в модуле A, и в модуле B, но несмотря на это, в модуле C поиск указанного экземпляра осуществляется только в собственном пространстве имен экземпляров, поэтому данный экземпляр не обнаруживается. Один из способов, позволяющих сослаться на имя экземпляра в другом модуле, состоит в том, чтобы включить в состав имени экземпляра имя модуля и отделитель имени модуля, например, как показано ниже.

```
CLIPS> (send [A::same] print)↵
[same] of A::ACLASS
CLIPS> (send [B::same] print)↵
[same] of B::BCLASS
CLIPS>
```

В данном случае в составе имени экземпляра было указано имя модуля, поэтому система CLIPS имеет информацию о том, какое пространство имен должно использоваться для поиска экземпляра, указанного по имени. Для поиска экземпляра, на который указывает имя экземпляра, может также применяться сам отделитель имени модуля, отдельно взятый:

```
CLIPS> (send [::same] print)↵
[same] of B::BCLASS
CLIPS>
```

В рассматриваемом примере система CLIPS вначале осуществляет поиск указанного экземпляра в текущем модуле, а затем проверяет наличие экземпляра в каждом модуле, из которого были импортированы конструкции `defclass`. Вообще говоря, если известно, что экземпляр определен либо в текущем модуле, либо в одном из импортированных модулей, следует использовать только спецификатор модуля. Дело в том, что в данном примере в первую очередь был найден экземпляр `[B::same]`, но если бы порядок операторов импорта был изменен на противоположный, то первым был бы найден экземпляр `[A::same]`.

Соглашения, применяемые для формирования ссылок на экземпляры в других модулях, могут показаться сложными, но на практике достаточно лишь иметь общее представление о том, как они действуют. Как правило, в программах ссылки на экземпляры создаются с применением адресов экземпляров, а не имен экземпляров. При использовании адресов экземпляров не возникает неоднозначность в отношении того, на какой экземпляр осуществляется ссылка, в отличие от тех ситуаций, когда применяются имена экземпляров. Дело в том, что имена экземпляров обычно используются при передаче сообщения экземпляру из приглашения к вводу команд и такие имена, вообще говоря, являются уникальными, поэтому если в текущем модуле импортируется конструкция `defclass`, связанная с рассматриваемым экземпляром, то можно применять лишь отделитель имени модуля.

11.17 Загрузка и сохранение экземпляров

В языке CLIPS предусмотрено несколько команд, предназначенных для сохранения экземпляров в файлах и загрузки экземпляров из файлов по такому же принципу, как осуществляется сохранение и загрузка фактов. Этими командами являются **save-instances**, **bsave-instances**, **load-instances**, **restore-instances** и **bload-instances**. Указанные команды имеют следующий синтаксис:

```
(save-instances <file-name>
   [<save-scope> [[inherit] <class-names>+])
(bsave-instances <file-name>
   [<save-scope> [[inherit] <class-names>+])
(load-instances <file-name>)
(restore-instances <file-name>)
(bload-instances <file-name>)
```

В этих определениях терм `<save-scope>` имеет следующую форму:

```
visible | local
```

Команда `load-instances` загружает группу экземпляров, хранящихся в файле, который указан параметром `<file-name>`. Экземпляры в этом файле должны быть представлены в стандартном формате, который используется кон-

струкцией definstances для определения экземпляров. Например, если файл "instances.dat" содержит информацию

```
(Jack of PERSON (full-name "Jack Q. Public")
                (age 23))
(of PERSON (full-name "John Doe")
                (hair-color black))
```

то загрузку экземпляров, хранящихся в этом файле, обеспечит такая команда:

```
(load-instances "instances.dat")
```

Вызов команды load-instances эквивалентен выполнению ряда вызовов команды make-instance. Команда restore-instances аналогична команде load-instances, но в ней не используются средства передачи сообщений при удалении, инициализации или определении значений слотов тех экземпляров, которые загружаются с ее помощью. Возвращаемым значением обеих этих команд является количество загруженных экземпляров или -1, если при выполнении команды не удалось получить доступ к файлу экземпляров.

Команда save-instances может использоваться для сохранения экземпляров в файле, указанном параметром <file-name>. Экземпляры хранятся в формате, требуемом для команд load-instances и restore-instances. Если параметр <save-scope> не задан или определен как **local**, то в файле сохраняются только те экземпляры, которые соответствуют конструкциям defclass, определенным в текущем модуле. Если в качестве значения параметра <save-scope> задано **visible**, то в файле сохраняются все экземпляры, соответствующие конструкциям defclass, видимым в текущем модуле. Если задан параметр <save-scope>, то появляется также возможность задать одно или несколько имен конструкций defclass. В этом случае сохраняются только экземпляры, соответствующие указанным конструкциям defclass (но имена конструкций defclass все равно должны соответствовать спецификации local или visible). Если задано также ключевое слово **inherit**, то сохраняются и экземпляры подклассов указанных классов, если подклассы также соответствуют спецификации local или visible. Возвращаемым значением этой команды является количество сохраненных экземпляров.

Команды bsave-instances и bload-instances аналогичны командам save-instances и load-instances, за исключением того, что для хранения экземпляров используется двоичный, а не текстовый формат. В связи с этим команда bload-instances может применяться только к файлам, сохраненным с помощью команды bsave-instances. Преимуществом этих команд является то, что при наличии большого количества экземпляров загрузка в двоичном формате осуществляется быстрее, чем в текстовом формате. А их недостаток состоит в том, что двоичные файлы обычно не переносятся с одной компьютерной платформы на другую.

11.18 Резюме

В настоящей главе приведено вводное описание языка COOL (CLIPS Object-Oriented Language — объектно-ориентированный язык CLIPS). Экземпляры (или объекты) служат в качестве еще одного способа представления данных, предусмотренного в языке CLIPS. Атрибуты экземпляра задаются с помощью конструкций `defclass`. С использованием конструкций `defmessage-handler` с классами может быть связан процедурный код, оформленный в виде обработчиков сообщений. Механизм наследования позволяет использовать в классе слоты и обработчики сообщений, связанные с другим классом. Язык COOL поддерживает и единичное, и множественное наследование. Класс, который наследует свойства от другого класса, называется *подклассом* этого класса. Класс, от которого подкласс наследует свои свойства, называется *суперклассом* этого класса. Подкласс наследует от своих суперклассов атрибуты и обработчики сообщений, но при наличии в подклассе и суперклассе дублирующихся определений подкласс перекрывает определения суперкласса. С помощью атрибута `role` могут создаваться абстрактные классы, применяемые только для наследования. Экземпляры абстрактных классов не могут быть созданы. В отличие от этого, конкретные классы могут использоваться и для наследования, и для создания экземпляров этих классов. В языке COOL заранее определен целый ряд примитивных классов. Большинство классов, создаваемых пользователем, наследуют свои свойства от системного класса `USER`. Конструкция `definstances` позволяет создавать множество указанных экземпляров после выдачи команды `reset`.

Кроме атрибутов слотов, предусмотренных в конструкции `deftemplate`, несколько дополнительных атрибутов слотов поддерживается также конструкцией `defclass`. Атрибут `source` позволяет наследовать атрибуты слотов от суперклассов. Атрибут `propagation` дает возможность отменить наследование слота. Атрибут `pattern-match` позволяет указать, должен ли слот или класс участвовать в сопоставлении с шаблонами. Атрибут `visibility` позволяет управлять тем, может ли осуществляться непосредственный доступ к слоту со стороны обработчиков сообщений подклассов. Атрибут `access` непосредственно ограничивает разрешенный тип доступа к слоту. Атрибут `create-accessor` используется для управления автоматическим созданием обработчиков `get-` и `put-` для слотов класса. Атрибут `storage` позволяет указать, является ли значение слота совместно используемым всеми экземплярами класса, или каждый экземпляр имеет собственное значение этого слота.

В системе CLIPS предоставляется несколько заранее определенных системных обработчиков сообщений, предназначенных для создания, инициализации, вывода на внешнее устройство и удаления экземпляров. Кроме того, могут быть также созданы обработчики сообщений, определяемые пользователем. Обработчики сообщений вызываются путем передачи экземпляру имени сообщения на-

ряду с относящимися к сообщению параметрами с помощью команды `send`. Обработчик сообщений может относиться к одному из четырех типов: `primary`, `around`, `before` или `after`. Обработчик типа `primary` обычно используется в качестве основного обработчика, предназначенного для формирования ответа на сообщение. Обработчики `before` и `after` вызываются соответственно до и после обработчика `primary`. Обработчик `around` именуется также обработчиком-оболочкой, поскольку он заключает в себе обработчики `before`, `after` и `primary`, выполняя свой код и до, и после того, как выполняется код этих обработчиков. Обработчики `around` должны явно вызывать обработчики других типов. Обработчики сообщений `primary` перекрывают (или, как принято говорить, *затеняют*) обработчик сообщений `primary` для того же сообщения, унаследованного от суперкласса, хотя и предусмотрена возможность вызывать затененные обработчики. Но обработчики сообщений суперклассов `around`, `before` и `after` не затеняются определением подкласса.

Сопоставление с шаблоном объекта предоставляет ряд возможностей, которые не могут обеспечиваться с помощью согласования с шаблоном фактов. Во-первых, единственный шаблон объекта может согласовываться с экземплярами нескольких классов. Во-вторых, изменения в значениях слотов, которые не заданы в шаблоне объекта, не приводят к повторной активизации правила, к которому принадлежит шаблон. В-третьих, изменения в значениях слотов, которые не заданы в шаблоне объекта в пределах условного элемента `logical`, не удаляют логическое обоснование, предусмотренное соответствующим правилом.

В языке COOL предусмотрено несколько функций запроса множества экземпляров, которые позволяют непосредственно запрашивать множества экземпляров, отвечающих заданному множеству условий. Некоторые из этих функций позволяют также осуществлять определенные действия над результатами запросов. В ограничениях параметров для универсальных функций могут применяться определяемые пользователем классы, кроме предопределенных типов, предусмотренных в языке CLIPS. Для сохранения экземпляров в файле и загрузки экземпляров из файла могут использоваться функции `save-instances`, `bsave-instances`, `load-instances`, `restore-instances` и `bload-instances`.

Задачи

11.1. Модифицируйте программу, разработанную в результате решения задачи 10.3 (см. с. 845), чтобы предусмотреть в ней возможности объяснения. После вывода информации о кустарнике, в наибольшей степени соответствующем требованиям пользователя, программа должна вывести приглашение для пользователя, чтобы он мог определить, желает ли он получить

объяснение по поводу выбора указанного кустарника. Если будет нажата клавиша ввода, то программа должна остановиться. Если же будет введено название кустарника, то программа должна перечислить требования, которым он соответствует, требования, которым он не соответствует, а также другие требования, которым этот кустарник мог бы соответствовать. После вывода этого объяснения пользователю должен поступить еще один запрос, чтобы он мог указать, желает ли он получить объяснение по поводу кустарника другого вида.

11.2. Модифицируйте программу, разработанную в результате решения задачи 10.7 (см. с. 846), таким образом, чтобы с ее помощью можно было составлять расписание для нескольких учащихся. Входные данные для программы должны считываться из файла. Вы вправе сами определять формат входных данных, но данные должны по меньшей мере включать имя каждого учащегося, названия предметов, которые должны быть включены в расписание, а также данные о том, какие преподаватели и какое время проведения занятий являются для учащегося более или менее предпочтительными. Выходные данные программы должны быть записаны в файл. Выходной файл должен содержать имя каждого учащегося, за которым следует расписание занятий по всем предметам для данного учащегося.

11.3. Модифицируйте программу, разработанную в результате решения задачи 10.6 (см. с. 845), для демонстрации динамически реконфигурируемых меню. Например, предположим, что после выбора пункта меню 1 в субменю 1 должны отображаться два пункта меню в субменю 2, а после выбора пункта меню 2 в субменю 1 — четыре пункта меню в субменю 2.

11.4. Применительно к следующей конструкции `defclass` напишите обработчик `before` для обработчика `get-side`, который должен запрашивать у пользователя значение слота `side`, если текущим значением является значение `unspecified`, заданное по умолчанию:

```
(defclass SQUARE
   (is-a USER)
   (slot side (default unspecified)))
```

11.5. Создайте конструкцию `defclass` с именем `ARRAY` и связанные с ней обработчики сообщений, которые позволяют представить многомерный массив. Должны быть предусмотрены обработчики сообщений для получения и задания значений элементов массива. Кроме того, требуется также обеспечить возможность задавать применяемое по умолчанию значение для элементов массива и указывать начальное множество значений элементов массива в вызове `make-instance`, используемом для создания экземпляра `ARRAY`. Наконец, должен быть предусмотрен обработчик сообщений для отображения содержимого массива. Одномерные массивы должны отображаться

в виде строки значений, а двумерные массивы — в виде таблицы, состоящей из строк и столбцов. Массивы с более высоким количеством измерений должны отображаться с помощью листинга, в котором на каждой строке представлено одно значение, а этому значению предшествуют индексы.

11.6. С использованием конструкции `defclass` с именем `ARRAY` и обработчиков сообщений, разработанных в результате решения задачи 11.5, перегрузите функцию `*` методом, который перемножает два двумерных массива. Применяйте дополнительные методы для проверки таких условий ошибки, как количество строк и столбцов двух массивов, несовместимое для операции умножения массивов.

11.7. Создайте конструкцию `defclass` с именем `LINKED-LIST` и соответствующие обработчики сообщений, которые позволяют создавать связные списки. Эта конструкция `defclass` должна быть спроектирована таким образом, чтобы от нее могли наследовать свойства существующие классы для приобретения функциональных возможностей работы со связными списками. Необходимо предусмотреть обработчики сообщений, обеспечивающие выборку следующего и предыдущего объектов в списке, вставку объекта в список, удаление объекта из списка и вывод списка на внешнее устройство. Напишите программу, демонстрирующую использование конструкции `defclass` с именем `LINKED-LIST` на примере конструкции `defclass`, которая наследует от нее свойства, а также на примере другого класса.

11.8. Создайте конструкцию `defclass` с именем `ITERATOR` и соответствующие обработчики сообщений, которые обеспечивают итеративную обработку полей многозначного значения. Значения, по которым осуществляется итерация, должны считываться и задаваться в ходе осуществления вызова `make-instance`, применяемого для создания экземпляра этого класса, но во всех остальных отношениях ни один из слотов данного класса не должен быть доступным, кроме как для обработчиков сообщений этого класса. Обработчик сообщений `first` должен быть предназначен для инициализации процесса итеративной обработки и возврата первого значения в списке итеративно обрабатываемых элементов. Обработчики сообщений `next` и `previous` должны соответственно обеспечивать выборку следующего или предыдущего значения в итеративном списке.

11.9. Создайте конструкцию `defclass` с именем `MEASUREMENT`, которая обеспечивает хранение данных о значении длины и единицах измерения длины. Используя методы, разработанные в результате решения задач 10.20 и 10.21, создайте новый метод, который складывает данные, представленные двумя экземплярами `MEASUREMENT`, и возвращает новый экземпляр, содержащий вычисленную сумму.

11.10. Создайте конструкцию `defclass` с именем `STACK` наряду с обработчиками сообщений, поддерживающими операции со стеком `push` (задвинуть) и `pop` (вытолкнуть).

11.11. Создайте конструкцию `defclass` с именем `SHUFFLER`, которая реализует обработчик сообщений `shuffle` (тасовать), обеспечивающий переупорядочение случайным образом списка значений, хранимого в экземпляре `SHUFFLER`.

11.12. Создайте конструкцию `defclass` с именем `CARD` для представления игральной карты. Создайте конструкцию `defclass` с именем `DECK`, которая инициализируется таким образом, чтобы в ней содержалась информация о колоде из 52 игральных карт. Предусмотрите обработчик сообщений `shuffle`, в котором используется конструкция `defclass` с именем `SHUFFLER`, разработанная в результате решения задачи 10.11, чтобы можно было с его помощью перетасовать колоду.

11.13. Перекройте функции `-`, `*` и `/` методами, которые обеспечивают выполнение операций вычитания, умножения и деление над экземплярами класса `COMPLEX`.

11.14. Напишите конструкцию `deffunction`, которая подсчитывает все вхождения каждого знака в строке и выводит итоговые данные о количестве обнаруженных знаков алфавита.

11.15. Создайте конструкцию `defclass` с именем `DIRECTORY`, предназначенную для хранения имен и номеров телефонов. Должны быть предусмотрены обработчики сообщений, позволяющие добавлять и удалять записи в этом телефонном справочнике, а также осуществлять поиск в справочнике по имени или по номеру и выводить все найденные записи.

Глава 12

Примеры проектов экспертных систем

12.1 Введение

В настоящей главе приведено несколько примеров программ CLIPS. Первый пример демонстрирует, как может быть представлена неопределенность в языке CLIPS. Следующие два примера показывают способы эмуляции других подходов к представлению знаний с использованием CLIPS. В одном из них показано, как можно представить в языке CLIPS деревья решений, а во втором продемонстрирован способ представления правил обратного логического вывода в языке CLIPS. В четвертом и последнем примере создается инфраструктура простой экспертной системы, предназначенной для текущего контроля над показаниями группы датчиков.

12.2 Коэффициенты достоверности

В языке CLIPS непосредственно не предусмотрены какие-либо возможности учета неопределенности. Тем не менее в программу CLIPS несложно включить средства учета неопределенности, помещая информацию, касающуюся неопределенности, непосредственно в факты и правила. В качестве примера достаточно указать, что с помощью языка CLIPS может быть эмулирован механизм учета неопределенности, применяемый в системе MYCIN. В настоящем разделе будет показано, как можно перезаписать на языке CLIPS следующее правило MYCIN [25]:

```
IF
   The stain of the organism is gramneg and
   The morphology of the organism is rod and
   The patient is a compromised host
THEN
   There is suggestive evidence (0.6) that the
      identity of the organism is pseudomonas
```

В системе MYCIN фактическая информация представлена в виде троек "объект–атрибут–значение" (Object–Attribute–Value — OAV). Такие тройки OAV могут быть представлены в языке CLIPS с помощью следующей конструкции `deftemplate` (эта конструкция будет помещена в собственный модуль в целях создания повторно применимого программного компонента):

```
(defmodule OAV (export deftemplate oav))
(deftemplate OAV::oav
   (multislot object (type SYMBOL))
   (multislot attribute (type SYMBOL))
   (multislot value))
```

Эта конструкция `deftemplate` позволяет представить некоторые факты, требуемые для части `IF` приведенного выше правила MYCIN, следующим образом:

```
(oav (object organism)
     (attribute stain)
     (value gramneg))
(oav (object organism)
     (attribute morphology)
     (value rod))
(oav (object patient)
     (attribute is a)
     (value compromised host))
```

Кроме того, в системе MYCIN с каждым фактом ассоциируется коэффициент достоверности (Certainty Factor — CF), который характеризует степень доверия к факту. Коэффициент достоверности может иметь значение от `-1` до `1`; значение `-1` показывает, что факт является заведомо ложным, значение `0` говорит о том, что какая-либо информация об этом факте отсутствует (налицо полная неопределенность), а значение `1` свидетельствует, что факт является заведомо истинным.

В системе CLIPS коэффициенты достоверности не учитываются автоматически, поэтому необходимо обеспечить сопровождение и данной информации. В этих целях в каждом факте будет использоваться дополнительный слот, представляющий коэффициент достоверности. После этого конструкция `deftemplate` с именем `oav` для каждого факта принимает такой вид:

```
(deftemplate OAV::oav
   (multislot object (type SYMBOL))
   (multislot attribute (type SYMBOL))
   (multislot value)
   (slot CF (type FLOAT) (range - 1.0 +1.0)))
```

В качестве примеров фактов можно привести следующее:

```
(oav (object organism)
     (attribute stain)
     (value gramneg)
     (CF 0.3))
(oav (object organism)
     (attribute morphology)
     (value rod)
     (CF 0.7))
(oav (object patient)
     (attribute is a)
     (value compromised host)
     (CF 0.8))
```

Для того чтобы факты `oav` функционировали должным образом, в программу на языке CLIPS необходимо внести еще одну модификацию. Система MYCIN позволяет осуществить логический вывод одних и тех же троек OAV с помощью отдельных правил. Затем эти тройки OAV комбинируются для получения единственной тройки OAV, в которой комбинируются коэффициенты достоверности исходных троек OAV. Применяемая в настоящее время конструкция `deftemplate` с именем `oav` позволяет вносить в список фактов две идентичные тройки OAV только в том случае, если в них имеются различные коэффициенты достоверности (поскольку система CLIPS в обычных условиях не позволяет вносить в список фактов два дублирующихся факта). Для того чтобы обеспечить возможность внесения в список фактов идентичных троек OAV, имеющих одинаковые коэффициенты достоверности, можно использовать **команду set-fact-duplication** для отмены применяемого в системе CLIPS принципа работы, согласно которому предотвращается внесение дублирующихся фактов в список фактов. Указанный принцип действия отменяется с помощью команды, имеющий следующий синтаксис:

```
(set-fact-duplication TRUE)
```

Аналогичным образом, команда, имеющая следующую форму, исключает возможность внесения в список фактов дублирующихся фактов:

```
(set-fact-duplication FALSE)
```

Как уже было сказано, в системе MYCIN две идентичные тройки OAV комбинируются в одну тройку OAV, имеющую комбинированное значение коэффициента достоверности. Для вычисления нового коэффициента достоверности в системе

MYCIN используется следующая формула, если оба коэффициента достоверности двух фактов (обозначенные как CF_1 и CF_2) больше или равны нулю:

$$\text{New Certainty} = (CF_1 + CF_2) - (CF_1 * CF_2)$$

Например, предположим, что в списке фактов имеются следующие факты:

```
(oav (object organism)
     (attribute morphology)
     (value rod)
     (CF 0.7))
(oav (object organism)
     (attribute morphology)
     (value rod)
     (CF 0.5))
```

Допустим, что CF_1 обозначает коэффициент достоверности первого факта, равный 0.7, а CF_2 — коэффициент достоверности второго факта, равный 0.5; в таком случае новый коэффициент достоверности для комбинации этих двух фактов вычисляется таким образом:

$$\begin{aligned}\text{New Certainty} &= (0.7 + 0.5) - (0.7 * 0.5) \\ &= 1.2 - 0.35 \\ &= 0.85\end{aligned}$$

а новый факт, заменяющий два первоначальных факта, принимает следующий вид:

```
(oav (object organism)
     (attribute morphology)
     (value rod)
     (CF 0.85))
```

Как уже было сказано, система CLIPS не обрабатывает автоматически коэффициенты достоверности, относящиеся к фактам; из этого следует, что CLIPS также не комбинирует автоматически две тройки OAV, полученные с помощью разных правил. Но комбинирование троек OAV можно легко обеспечить с помощью правила, которое осуществляет поиск в списке фактов идентичных троек OAV, подлежащих комбинированию. Ниже показано правило и описан метод, которые демонстрируют, как осуществляются указанные действия применительно к таким попарно обрабатываемым тройкам OAV, в которых коэффициенты достоверности больше или равны нулю.

```
(defmethod OAV::combine-certainties
   ((?C1 NUMBER (> ?C1 0))(?C2 NUMBER (> ?C2 0)))
    (- (+ ?C1 ?C2)(* ?C1 ?C2)))
```

```
(defrule OAV::combine-certainties
   (declare (auto-focus TRUE))
   ?fact1 <- (oav (object $?o)
                  (attribute $?a)
                  (value $?v)
                  (CF ?C1))
   ?fact2 <- (oav (object $?o)
                  (attribute $?a)
                  (value $?v)
                  (CF ?C2))
   (test (neq ?fact1 ?fact2))
   =>
   (retract ?fact1)
   (modify ?fact2
(CF (combine-certainties ?C1 ?C2))))
```

Обратите внимание на то, что идентификаторы фактов `?fact1` и `?fact2` сравниваются друг с другом в условном элементе `test`. Такая операция применяется для получения гарантий того, что правило не согласовано с фактом с использованием точно такого же факта для первых двух шаблонов. Адреса фактов позволяют сравнить функции `eq` и `neq`. Кроме того, следует отметить, что для данного правила разрешен атрибут `auto-focus`. Это позволяет гарантировать, что две тройки OAV будут скомбинированы, прежде чем будет разрешен запуск других правил, шаблонам которых соответствуют обе эти тройки.

Следующим шагом на пути к внедрению средств поддержки коэффициентов достоверности в систему CLIPS является связывание коэффициентов достоверности фактов, согласующихся с левой частью правила, с коэффициентами достоверности фактов, внесенных в список фактов с помощью правой части правила. В системе MYCIN логический вывод коэффициента достоверности, ассоциирующегося с левой частью правила, осуществляется с использованием следующих формул:

$$CF(P_1 \text{ or } P_2) = \max\{CF(P_1), CF(P_2)\}$$
$$CF(P_1 \text{ and } P_2) = \min\{CF(P_1), CF(P_2)\}$$
$$CF(\text{not } P) = -CF(P)$$

В этих формулах P, P_1 и P_2 обозначают шаблоны из левой части правила. Кроме того, если коэффициент достоверности в левой части правила меньше 0.2, то правило рассматривается как неприменимое и запуск его не происходит.

Логический вывод значения коэффициента достоверности факта, внесенного в список фактов под действием правой части правила, осуществляется путем умножения коэффициента достоверности вносимого факта на коэффициент до-

стоверности, заданный в левой части правила. Ниже приведен результат преобразования правила MYCIN, приведенного в начале данного раздела, в правило CLIPS; это правило показывает, как вычисляются коэффициенты достоверности в левой и правой частях правила. Правило помещается в модуль `IDENTIFY`, который импортирует конструкцию `deftemplate` с именем `oav` из модуля `OAV`.

```
defmodule IDENTIFY (import OAV deftemplate oav))
(defrule IDENTIFY::MYCIN-to-CLIPS-translation
   (oav (object organism)
        (attribute stain)
        (value gramneg)
        (CF ?C1))
   (oav (object organism)
        (attribute morphology)
        (value rod)
        (CF ?C2))
   (oav (object patient)
        (attribute is a)
        (value compromised host)
        (CF ?C3))
   (test (> (min ?C1 ?C2 ?C3) 0.2))
   =>
   (bind ?C4 (* (min ?C1 ?C2 ?C3) 0.6))
   (assert (oav (object organism)
           (attribute identity)
           (value pseudomonas)
           (CF ?C4))))
```

Для завершения эмуляции коэффициентов достоверности MYCIN требуется еще один, последний шаг. В правиле `combine-certainties` предусмотрен единственный метод `combine-certainties`, в котором учитывается лишь такой случай, когда оба коэффициента достоверности являются положительными. А введение дополнительных методов должно позволить учитывать другие случаи комбинирования коэффициентов достоверности. Ниже описаны оставшиеся случаи комбинирования.

$$\text{New Certainty} = (CF_1 + CF_2) + (CF_1 * CF_2) \quad \text{если } CF_1 \leqslant 0 \text{ и } CF_2 \leqslant 0$$

$$\text{New Certainty} = \frac{CF_1 + CF_2}{1 - \min\{CF_1, CF_2\}} \quad \text{если } -1 < CF_1 * CF_2 < 0$$

12.3 Деревья решений

Как было указано в главе 3, деревья решений лежат в основе удобного подхода к решению задач классификации некоторых типов. Деревья решений позволяют находить решения путем сокращения множества возможных ответов с помощью осуществления ряда выборов или ответов на вопросы, которые отсекают лишние ветви в пространстве поиска. Задачи, пригодные для решения с помощью деревьев решений, характеризуются тем, что ответ в них отыскивается из заранее заданного множества возможных ответов. Например, для решения одной из задач таксономии может потребоваться идентификация драгоценного камня путем выбора из множества всех известных драгоценных камней, а для решения задачи диагностики может потребоваться выбрать один возможный способ исправления ситуации из множества способов исправления ситуации или одну причину нарушения из множества возможных причин. Но, вообще говоря, деревья решений не очень хорошо подходят для задач составления расписаний, планирования и синтеза, в которых приходится вырабатывать решения, а не только выбирать среди существующих решений, поскольку при использовании деревьев решений множество ответов должно быть определено заранее.

Напомним, что деревья решений состоят из узлов и ребер. Узлы представляют различные местонахождения в дереве. Ребра, направленные сверху вниз, соединяют родительские узлы с дочерними, а ребра, направленные снизу вверх, соединяют дочерние узлы с родительскими. Узел, находящийся в вершине дерева и не имеющий родительских узлов, называется *корневым узлом*. Следует отметить, что в дереве каждый узел имеет только один родительский узел, за исключением корневого узла, не имеющего родительского узла. Узлы, не имеющие дочерних узлов, называются листовыми узлами.

Листовые узлы дерева решений представляют все возможные решения, которые могут быть получены логическим путем с помощью этого дерева. Такие узлы именуются *узлами ответов*, а все другие узлы в дереве называются *узлами принятия решений*. Каждый узел принятия решений представляет вопрос, на который нужно дать ответ, или решение, которое нужно принять, после чего определяется соответствующее ребро дерева решений, по которому необходимо проследовать дальше. В простых деревьях решений в качестве вопроса могут применяться вопросы, требующие однозначного ответа, “да” или “нет”, например: “Является ли данное животное теплокровным?”. Левое ребро, отходящее от узла, представляет путь, по которому нужно проследовать, если ответ положителен, а правое ребро — путь дальнейшего продвижения, если ответ отрицателен. Но, вообще говоря, в узле принятия решений для выбора ребра, по которому необходимо проследовать дальше, могут применяться любые критерии, при условии, что процесс выбора всегда приводит к получению решения, указывающего только на единственное ребро. Таким образом, в узлах принятия решений может выбираться ребро, со-

ответствующее множеству или диапазону значений, ряду случаев или функциям, отображающим состояния, заданные в узле принятия решений, на ребра, исходящие из узла принятия решений. Структура узлов принятия решений может также быть настолько сложной, чтобы с их помощью осуществлялся перебор с возвратами или вероятностные рассуждения.

Для иллюстрации процесса функционирования дерева решений рассмотрим следующие эвристические правила выбора вина, которое может подаваться на стол вместе с различными блюдами:

```
IF the main course is red meat
THEN serve red wine
IF the main course is poultry and it is turkey
THEN serve red wine
IF the main course is poultry and it is not turkey
THEN serve white wine
IF the main course is fish
THEN serve white wine
```

Представление этих эвристических правил по выбору вина в виде бинарного дерева решений показано на рис. 12.1. Узлы принятия решений сформированы на основе предположения, что на каждый вопрос может быть получен только положительный или отрицательный ответ. На тот случай, что главным блюдом не является ни черное мясо (баранина, говядина), ни мясо домашней птицы, ни рыба, к множеству эвристических правил добавлен применяемый по умолчанию узел ответа, содержащий ответ “наиболее подходящий тип вина неизвестен”.

Процедура прохождения по дереву и достижения узла ответа весьма проста. Процесс логического вывода начинается с того, что текущее местонахождение в дереве решений устанавливается на корневой узел. Если текущим местонахождением является один из узлов принятия решений, то необходимо дать в определенной форме ответ на вопрос, связанный с данным узлом принятия решений (как правило, такой ответ предоставляет лицо, получающее консультацию с использованием дерева решений). Если ответ является положительным, то текущее местонахождение устанавливается на дочерний узел, соединенный с ребром положительного ответа (или левым ребром), отходящим от текущего местонахождения.

Если ответ на вопрос является отрицательным, то в качестве текущего местонахождения устанавливается дочерний узел, соединенный с ребром отрицательного ответа (или с правым ребром), отходящим от узла текущего местонахождения. Если в какой-либо момент времени текущим местонахождением становится узел ответа, то значение узла ответа рассматривается как ответ, полученный логическим путем с помощью проведения консультации на основе дерева решений. В противном случае процедура обработки узлов принятия решений продолжа-

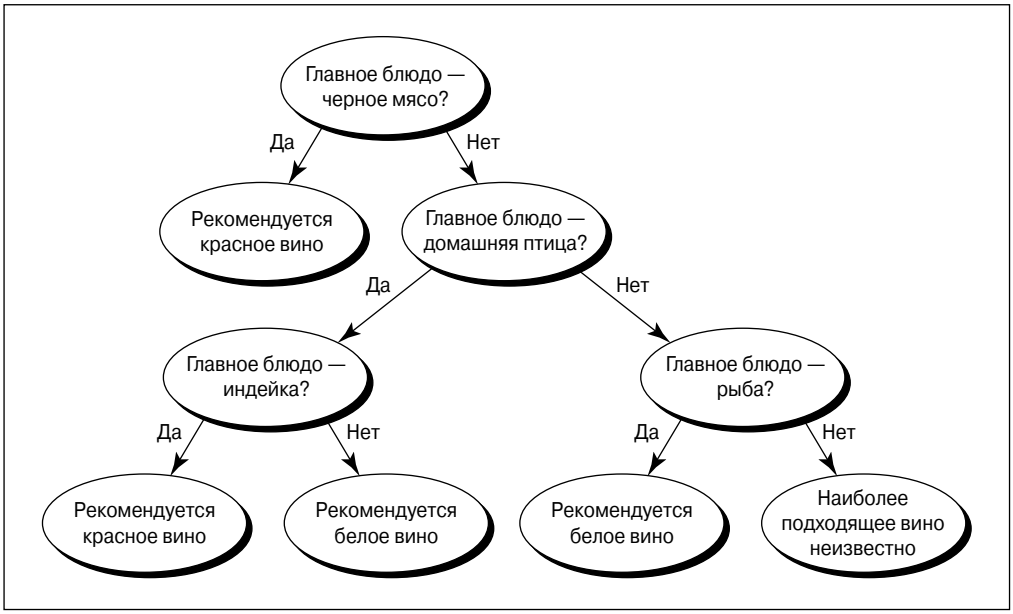


Рис. 12.1. Бинарное дерево решений

ется до тех пор, пока не будет достигнут один из узлов ответа. Ниже приведен псевдокод для такого алгоритма.

```
procedure Solve_Binary_Tree
   Set the current location in the tree
      to the root node.
   while the current location is a decision node do
      Ask the question at the current node.
      if the reply to the question is yes
         Set the current node to the yes branch.
      else
         Set the current node to the no branch.
      end if
   end do
   Return the answer at the current node.
end procedure
```

Деревья решений с несколькими ребрами

При использовании деревьев решений, которые допускают формирование только узлов принятия решений с бинарными ребрами, затрудняется представление решений, допускающих множество ответов или ряд случаев. Наглядным приме-

ром низкой эффективности таких деревьев решений может служить бинарное дерево решений, подготовленное для решения задачи с выбором вина. В том случае, если основным блюдом является рыба, приходится принимать три решения, для того чтобы определить, что вином наиболее подходящего типа является белое вино: отвечать на вопросы “Является ли основным блюдом черное мясо?”, “Является ли основным блюдом домашняя птица?” и “Является ли основным блюдом рыба?” Гораздо более удобным вопросом, позволяющим кратко представить данный узел принятия решений, мог бы стать следующий вопрос: “Что является основным блюдом?” Узел принятия решений, способный обрабатывать подобный вопрос, должен допускать выбор одного из нескольких ребер с учетом ряда возможных решений (в данном случае — черное мясо, домашняя птица, рыба и т.д.). На рис. 12.2 показано модифицированное дерево решений, впервые приведенное на рис. 12.1, которое теперь допускает использование нескольких ребер; такая возможность достигнута благодаря описанной ниже простой модификации алгоритма `Solve_Binary_Tree`.

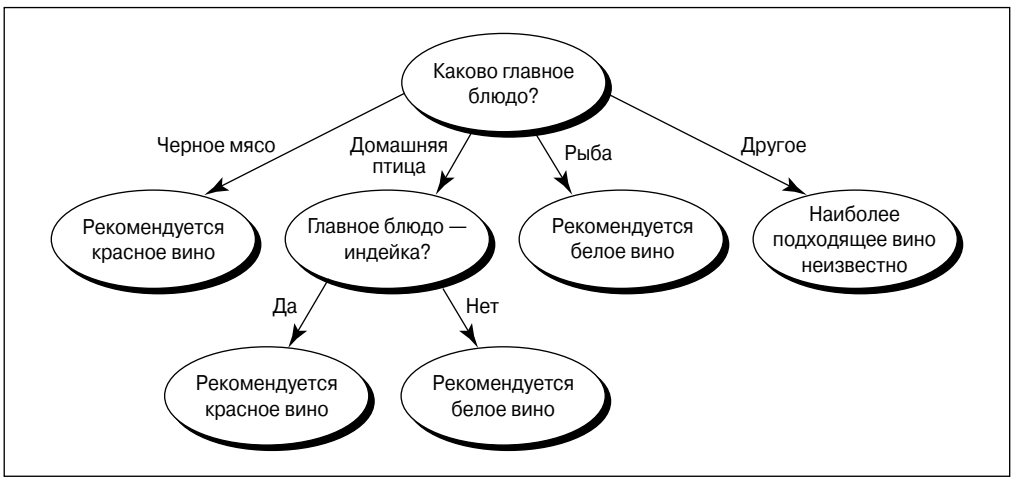


Рис. 12.2. Дерево решений с несколькими ребрами

```
procedure Solve_Tree
   Set the current tree location to the root node.
   while the current location is a decision node do
      Ask the question at the current node until
         an answer in the set of valid choices
         for this node has been provided.
      Set the current node to the child node of
         the branch associated with the choice  selected.
   end do
```

```
   Return the answer at the current node.
end procedure
```

Деревья решений, позволяющие проводить обучение

Иногда возникает необходимость вводить в деревья решений новые знания в процессе их обучения; в качестве примера подобного приложения принято использовать дерево решений для идентификации животных. После достижения в дереве решений некоторого ответа приложение выводит вопрос пользователю, является ли ответ правильным, и в случае подтверждения какие-либо дальнейшие действия не предпринимаются. Но если ответ оказался неправильным, то дерево решений модифицируется в целях включения правильного ответа. Узел ответа заменяется узлом принятия решений, который содержит вопрос, позволяющий провести различие между старым ответом, присутствовавшим в этом узле, и ответом, который не был угадан правильно. На рис. 12.3 показано дерево решений, которое позволяет определить вид животного по его характерным признакам. Это дерево решений является весьма упрощенным (в нем имеется информация только о трех животных) и необходимо провести обучение этого дерева.

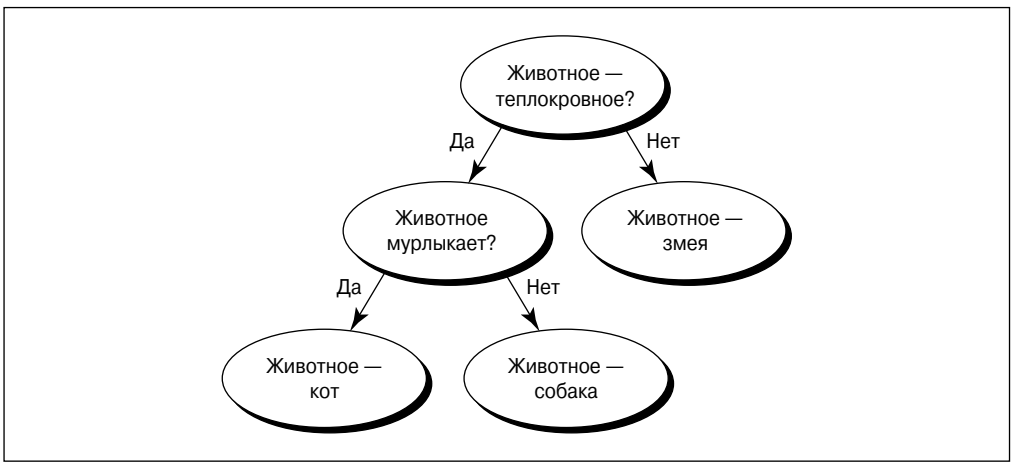


Рис. 12.3. Дерево решений для идентификации животных

В качестве примера рассмотрим сеанс выдвижения предположений с помощью этого дерева решений, который может проходить следующим образом:

```
Is the animal warm-blooded? (yes or no) yes↵
Does the animal purr? (yes or no) no↵
I guess it is a dog
Am I correct? (yes or no) no↵
What is the animal? bird↵
```

```
What question when answered yes will distinguish
   a bird from a dog? Does the animal fly?↵
Now I can guess bird
Try again? (yes or no) no↵
```

Такой сеанс может продолжаться неограниченно долго, в ходе чего дерево решений будет воспринимать в процессе обучения все больше и больше информации. На рис. 12.4 показано представление того же дерева решений, но после проведения описанного выше сеанса. Но обучение в такой форме имеет один недостаток — в конечном итоге полученное дерево решений может либо не иметь качественную иерархическую структуру, либо не отличаться очень высокой эффективностью выработки предположений, касающихся наиболее подходящего животного. Эффективное дерево решений должно иметь примерно одинаковое количество ребер от корня к узлам ответов по всем путям.

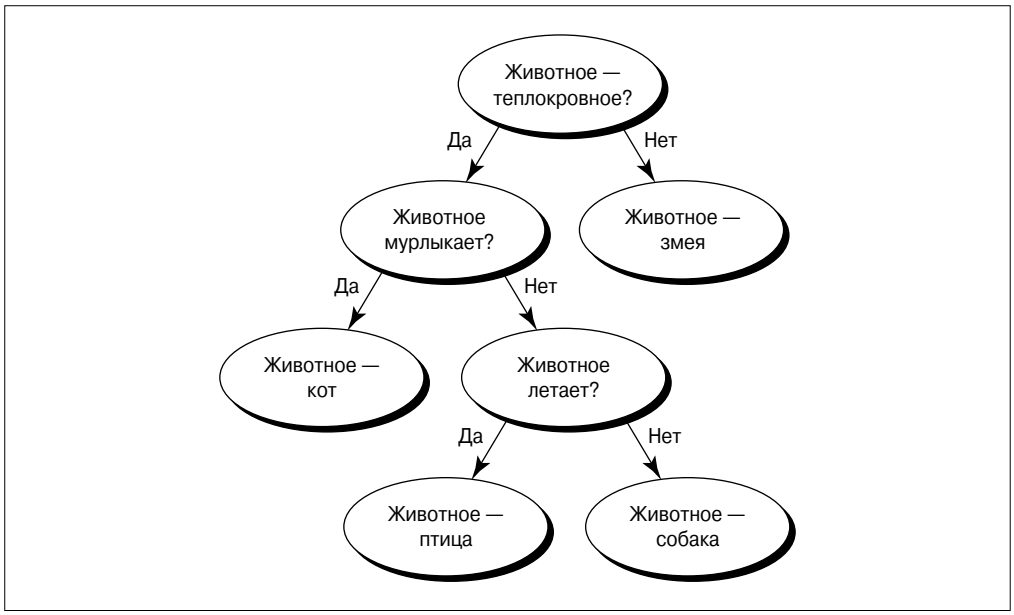


Рис. 12.4. Дерево решений для идентификации животных после обучения распознаванию птиц

Ниже приведен псевдокод, позволяющий модифицировать алгоритм `Solve_Tree` таким образом, чтобы в нем была реализована возможность обучения.

```
procedure Solve_Tree_and_Learn
   Set the current location in the tree
      to the root node
   while the current location is a decision node do
```

```
      Ask the question at the current node.
      if the reply to the question is yes
         Set the current node to the yes branch.
      else
         Set the current node to the no branch.
      end if
   end do
   Ask if the answer at the current node is correct.
   if the answer is correct
      Return the correct answer.
   else
      Determine the correct answer.
      Determine a question which when answered yes
         will distinguish the answer at the current
         node from the correct answer.
      Replace the answer node with a decision node
         that has as its no branch the current
         answer node and as its yes branch an
         answer node with the correct answer.
         The decision node's question should be
         the question which distinguishes the
         two answer nodes.
   end if
end procedure
```

Программа для работы с деревом решений, основанная на правилах

Первый шаг к выяснению того, как может быть реализовано в языке CLIPS обучающееся дерево решений, состоит в создании наиболее подходящего варианта представления знаний. Поскольку дерево решений должно обеспечивать обучение, по-видимому, целесообразно представить дерево в виде фактов, а не правил, в связи с тем, что факты могут быть легко добавлены и удалены для обновления дерева в ходе его обучения. Для перехода по дереву решений в ходе реализации алгоритма `Solve_Tree_and_Learn` с использованием подхода на основе правил может применяться множество правил CLIPS.

Каждый узел дерева решений будет представлен в виде факта. Для представления и узлов ответов, и узлов принятия решений будет использоваться следующая конструкция `deftemplate`:

```
(deftemplate node
   (slot name)
   (slot type)
   (slot question)
   (slot yes-node)
   (slot no-node)
   (slot answer))
```

В этом определении слот `name` содержит уникальное имя узла, а слот `type` обозначает тип узла и содержит значение `answer` или `decision`. Слоты `question`, `yes-node` и `no-node` используются только в узлах принятия решений. Слот `question` содержит вопрос, который должен быть задан при переходе к узлу принятия решений. Слот `yes-node` указывает на узел, к которому должен быть выполнен переход, если ответ на вопрос является положительным, а слот `no-node` указывает на узел, к которому должен быть выполнен переход, если ответ на вопрос является отрицательным. Слот `answer` используется только в узлах ответа и представляет собой ответ, формируемый деревом решений после перехода к узлу ответа.

Поскольку эта программа идентификации животных должна обучаться, то возникает необходимость сохранять информацию о том, что было усвоено программой в процессе обучения от одного прогона к другому. В данном случае для представления структуры дерева решений используется коллекция фактов, поэтому было бы удобно сохранять эти факты в файле с применением формата команды `load-facts`, затем вносить их в список фактов с помощью команды `load-facts` в начале работы программы и снова сохранять с использованием команды `save-facts` после завершения работы программы. Факты, предназначенные для этой программы, хранятся в файле `animal.dat`. Если в качестве исходного дерева решений используется дерево, показанное на рис. 12.3, то файл `animal.dat` должен содержать текст, приведенный ниже. Обратите внимание на то, что корневой узел (`root`) обозначен как таковой, а каждому из прочих узлов присвоено уникальное имя. Кроме того, следует отметить, что некоторые слоты (такие как `decision`, `yes-node` и `no-node` для узлов ответа) остаются незаданными, поскольку им должны присваиваться значения, предусмотренные по умолчанию (а для разрабатываемой программы не важно, какие значения будут помещены в эти слоты).

```
(node (name root) (type decision)
      (question "Is the animal warm-blooded?")
      (yes-node node1) (no-node node2))
(node (name node1) (type decision)
      (question "Does the animal purr?")
      (yes-node node3) (no-node node4))
```

```
(node (name node2) (type answer) (answer snake))
(node (name node3) (type answer) (answer cat))
(node (name node4) (type answer) (answer dog))
```

Теперь необходимо разработать правила, применяемые для прохождения по дереву решений. Инициализация программы обучения дерева решений осуществляется с помощью следующего правила:

```
(defrule initialize
   (not (node (name root)))
   =>
   (load-facts "animal.dat")
   (assert (current-node root)))
```

Запуск этого правила `initialize` происходит, если в списке фактов отсутствует факт, соответствующий корневому узлу. Действия, предусмотренные в правиле `initialize`, обеспечивают загрузку представления дерева решений в список фактов и внесение в список фактов того факта, который указывает, что текущим узлом, представляющим интерес, является корневой узел.

Ниже приведены конструкция `deffunction` и правило, которые выводят вопрос, связанный с узлом принятия решений, а затем вносят в список фактов тот факт, который содержит ответ на вопрос.

```
(deffunction ask-yes-or-no (?question)
   (printout t ?question " (yes or no) ")
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no))
      (printout t ?question " (yes or no) ")
      (bind ?answer (read)))
   (return ?answer))
(defrule ask-decision-node-question
   ?node <- (current-node ?name)
            (node (name ?name)
            (type decision)
            (question ?question))
   (not (answer ?))
   =>
   (assert (answer (ask-yes-or-no ?question))))
```

Второй шаблон согласуется с текущим узлом, только если этот узел представляет собой узел принятия решений. В третьем шаблоне проверяется, что ответ на заданный вопрос еще не получен. Конструкция `deffunction` с именем `ask-yes-or-no`, применяемая в правой части правила, позволяет повторно задавать вопрос до тех пор, пока не будет получен один из допустимых ответов,

yes или no. После получения ответа на вопрос запускается одно из следующих правил:

```
(defrule proceed-to-yes-branch
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (yes-node ?yes-branch))
   ?answer <- (answer yes)
   =>
   (retract ?node ?answer)
   (assert (current-node ?yes-branch)))
(defrule proceed-to-no-branch
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (no-node ?no-branch))
   ?answer <- (answer no)
   =>
   (retract ?node ?answer)
   (assert (current-node ?no-branch)))
```

В обоих правилах извлекается факт current-node, а затем вместо него в список фактов вносится новый факт, зависящий от ответа на вопрос. Факт с ответом извлекается для того, чтобы снова было активизировано правило ask-decision-node-question.

Следующее правило обеспечивает вывод вопроса о том, было ли предположение, представленное в узле, приемлемым; это правило действует аналогично правилу ask-decision-node-question:

```
(defrule ask-if-answer-node-is-correct
   ?node <- (current-node ?name)
   (node (name ?name) (type answer)
         (answer ?value))
         (not (answer ?))
   =>
   (printout t "I guess it is a " ?value crlf)
   (assert
(answer (ask-yes-or-no "Am I correct?"))))
```

При получении любого ответа, отличного от yes или no, правило bad-answer еще раз активизирует правило ask-if-answer-node-is-correct. А если ответом является yes или no, то активизируется одно из двух приведенных ниже правил. Если пользователь подтверждает правильность ответа, хранящегося

в узле ответа, то в список фактов вносится факт ask-try-again для указания на то, что пользователю необходимо задать вопрос, желает ли он продолжить работу с программой. Если же пользователь не подтверждает правильность ответа, то происходит переход в режим обучения и в список фактов вносится факт replace-answer-node для указания имени узла, который должен быть заменен. Но и в том и в другом случае извлекаются факт current-node и факт, соответствующий узлу ответа.

```
(defrule answer-node-guess-is-correct
   ?node <- (current-node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer yes)
   =>
   (assert (ask-try-again))
   (retract ?node ?answer))
(defrule answer-node-guess-is-incorrect
   ?node <- (current-node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer no)
   =>
   (assert (replace-answer-node ?name))
   (retract ?node ?answer))
```

Для определения того, желает ли пользователь продолжить работу, используются три приведенных ниже правила. Правило ask-try-again выводит вопрос "Try again?" (Сделать еще одну попытку?). Еще раз отметим, что запуск правила bad-answer осуществляется, если на вопрос не дан ответ yes или no. Если пользователь даст на вопрос "Try again?" ответ yes, то правило one-more-time переустанавливает факт current-node на корневой узел, чтобы можно было снова начать процесс выдвижения предположений. Если пользователь даст ответ no, то факты, представляющие дерево решений, сохраняются в файле animal.dat с помощью команды save-facts.

```
(defrule ask-try-again
   (ask-try-again)
   (not (answer ?))
   =>
   (assert (answer (ask-yes-or-no "Try again?"))))
(defrule one-more-time
   ?phase <- (ask-try-again)
   ?answer <- (answer yes)
   =>
   (retract ?phase ?answer)
```

```
   (assert (current-node root)))
(defrule no-more
   ?phase <- (ask-try-again)
   ?answer <- (answer no)
   =>
   (retract ?phase ?answer)
   (save-facts "animal.dat" local node))
```

Наконец, если пользователь ответит, что идентификация животного в узле ответа является неверной, то с помощью следующего правила добавляется новый узел принятия решений, которое позволяет провести обучение дерева решений:

```
(defrule replace-answer-node
   ?phase <- (replace-answer-node ?name)
   ?data <- (node (name ?name)
                  (type answer)
                  (answer ?value))
   =>
   (retract ?phase)
   ; Определить, каким должно быть предположение
   (printout t "What is the animal? ")
   (bind ?new-animal (read))
   ; Узнать, какой вопрос соответствует этому
   ; предположению
   (printout t "What question when answered yes ")
   (printout t "will distinguish " crlf " a ")
   (printout t ?new-animal " from a " ?value "? ")
   (bind ?question (readline))
   (printout t "Now I can guess " ?new-animal crlf)
   ; Создать новые узлы, полученные в~результате обучения
   (bind ?newnode1 (gensym*))
   (bind ?newnode2 (gensym*))
   (modify ?data (type decision)
                 (question ?question)
                 (yes-node ?newnode1)
                 (no-node ?newnode2))
   (assert (node (name ?newnode1)
                 (type answer)
                 (answer ?new-animal)))
   (assert (node (name ?newnode1)
                 (type answer)
                 (answer ?value)))
```

```
; Определить, желает ли игрок предпринять еще одну
; попытку
(assert (ask-try-again)))
```

С помощью правила `replace-answer-node` формируется вопрос о том, как следует идентифицировать данное животное и какой вопрос позволяет получить информацию, с помощью которой данное животное можно отличить от животного, название которого идентифицировалось до сих пор в дереве решений как допустимый ответ. Старый узел ответа заменяется узлом принятия решений и двумя узлами ответа, созданными для представления ответов на вопрос, вновь освоенный в результате обучения. С помощью **функции gensym*** (которая вырабатывает уникальный символ при каждом ее вызове) подготавливаются имена каждого из двух новых узлов ответа. После этого в список фактов вносится факт `ask-try-again` для определения того, должен ли быть выполнен еще один прогон программы.

Пошаговая трассировка программы обучения дерева решений

За функционированием этой программы обучения деревьев решений можно понаблюдать, отслеживая ее выполнение. Предположим, что правила поддержки дерева решений загружены и создан файл `animal.dat`, содержащий факты, которые представляют начальное дерево решений; в таком случае следующий диалог показывает состояние системы после выполнения команды `reset`:

```
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (reset)↵
==> f-0        (initial-fact)
CLIPS> (agenda)↵
0        initialize: f-0,
For a total of 1 activation.
CLIPS>
```

В связи с отсутствием факта корневого узла было активизировано правило `initialize`. Запуск правила `initialize` разрешен, поэтому загружаются факты, составляющие дерево решений. Обратите внимание на то, что часть вывода в приведенных ниже трассировках обозначена отступами для удобства чтения.

```
CLIPS> (run 1)↵
FIRE 1 initialize: f-0,
==> f-1  (node (name root) (type decision)
      (question "Is the animal warm-blooded?")
```

```
         (yes-node node1) (no-node node2)
         (answer nil))
==> f-2  (node (name node1) (type decision)
      (question "Does the animal purr?")
         (yes-node node3) (no-node node4)
         (answer nil))
==> f-3  (node (name node2) (type answer)
      (question nil) (yes-node nil)
         (no-node nil) (answer snake))
==> f-4  (node (name node3) (type answer)
      (question nil) (yes-node nil)
         (no-node nil) (answer cat))
==> f-5  (node (name node4) (type answer)
      (question nil) (yes-node nil)
         (no-node nil) (answer dog))
==> f-6  (current-node root)
CLIPS> (agenda)↵
0      ask-decision-node-question: f-6,f-1,
For a total of 1 activation.
CLIPS>
```

В правиле `initialize` для загрузки фактов в дерево решений используется функция `load-facts`. В качестве факта `current-node` задается факт корневого узла. Корневой узел — это узел принятия решений, поэтому активизируется правило `ask-decision-node-question`. Запуск этого правила и связанного с ним правила `proceed-to-yes-branch` разрешен, поэтому формируется следующий диалог:

```
CLIPS> (run 2)↵
FIRE     1 ask-decision-node-question: f-6,f-1,
Is the animal warm-blooded? (yes or no) yes↵
==> f-7     (answer yes)
FIRE     2 proceed-to-yes-branch: f-6,f-1,f-7
<== f-6     (current-node root)
<== f-7     (answer yes)
==> f-8     (current-node node1)
CLIPS> (agenda)↵
0      ask-decision-node-question: f-8,f-2,
For a total of 1 activation.
CLIPS>
```

С корневым узлом принятия решений связан вопрос: `"Is the animal warm-blooded?"` (Является ли животное теплокровным?) Поскольку ответ на

этот вопрос является положительным, под действием правила proceed-to-yes-branch текущим узлом становится узел, находящийся слева от данного узла принятия решений (узел node1). Узел node1 также является узлом принятия решений, поэтому снова активизируется правило ask-decision-node-question. После того как будет разрешен запуск очередных двух правил, формируется следующий диалог:

```
CLIPS> (run 2)↵
FIRE     1 ask-decision-node-question: f-8,f-2,
Does the animal purr? (yes or no) no↵
==> f-9     (answer no)
FIRE     2 proceed-to-no-branch: f-8,f-2,f-9
<== f-8     (current-node node1)
<== f-9     (answer no)
==> f-10    (current-node node4)
CLIPS> (agenda)↵
0      ask-if-answer-node-is-correct: f-10,f-5,
For a total of 1 activation.
CLIPS>
```

С узлом принятия решений node1 связан вопрос: "Does the animal purr?" (Это животное мурлыкает?) Поскольку ответ на вопрос является отрицательным, под действием правила proceed-to-no-branch текущим узлом становится узел, находящийся справа от узла принятия решений (узел node4). Узел node4 является узлом ответа, поэтому активизируется правило ask-if-answer-node-is-correct. После того как будет разрешен запуск этого правила и следующего за ним правила, формируется такой диалог:

```
CLIPS> (run 2)↵
FIRE     1 ask-if-answer-node-is-correct: f-10,f-5,
I guess it is a dog
Am I correct? (yes or no) no↵
==> f-11       (answer no)
FIRE     2 answer-node-guess-is-incorrect: f-10,f-5,
                                           f-11
==> f-12       (replace-answer-node node4)
<== f-10       (current-node node4)
<== f-11       (answer no)
CLIPS> (agenda)↵
0      replace-answer-node: f-12,f-5
For a total of 1 activation.
CLIPS>
```

С этим узлом ответа связано предположение, что искомым животным является собака. Но в данном случае предположение неверно, поэтому активизируется правило replace-answer-node для определения приемлемого ответа. После того как будет разрешен запуск этого правила, формируется следующий диалог:

```
CLIPS> (run 1)↵
FIRE     1 replace-answer-node: f-12,f-5
<== f-12       (replace-answer-node node4)
What is the animal? bird↵
What question when answered yes will distinguish
   a bird from a dog? Does the animal fly?↵
Now I can guess bird
<== f-5     (node (name node4) (type answer)
                  (question nil)
                  (yes-node nil) (no-node nil)
                  (answer dog))
==> f-13    (node (name node4) (type decision)
                  (question  "Does the animal fly?")
                  (yes-node gen1) (no-node gen2)
                  (answer dog))
==> f-14    (node (name gen1) (type answer)
                  (question nil)
                  (yes-node nil) (no-node nil)
                  (answer bird))
==> f-15    (node (name gen2) (type answer)
                  (question nil)
                  (yes-node nil) (no-node nil)
                  (answer dog))
==> f-16    (ask-try-again)
CLIPS> (agenda)↵
0      ask-try-again: f-16,
For a total of 1 activation.
CLIPS>
```

Прежде всего с помощью команды (replace-answer-node node4) извлекается управляющий факт. Затем определяется приемлемое предположение наряду с тем, какой вопрос к пользователю должен служить для определения приемлемого предположения. Узел неверного ответа модифицируется и становится узлом принятия решений, а затем формируются два узла ответов для этого нового узла принятия решений. Наконец в список фактов вносится управляющий факт ask-try-again для определения того, нужно ли идентифицировать еще

одно животное. После того как будет разрешен запуск правила ask-try-again, а затем правила no-more, формируется следующий диалог:

```
CLIPS> (run 2)↵
FIRE     1 ask-try-again: f-16,
Try again? (yes or no) no↵
==> f-17       (answer no)
FIRE     2 no-more: f-16,f-17
<== f-16       (ask-try-again)
<== f-17       (answer no)
CLIPS> (agenda)↵
CLIPS>
```

С помощью правила ask-try-again система задает пользователю вопрос о том, нужно ли предпринять еще одну попытку идентификации. Ответ на этот вопрос является отрицательным, поэтому с помощью правила no-more снова происходит сохранение дерева решений в файле animal.dat. После завершения этого сеанса файл animal.dat имеет такую форму:

```
(node (name root) (type decision)
      (question "Is the animal warm-blooded?")
      (yes-node node1) (no-node node2)
      (answer nil))
(node (name node1) (type decision)
      (question "Does the animal purr?")
      (yes-node node3) (no-node node4)
      (answer nil))
(node (name node2) (type answer)
      (question nil)
      (yes-node nil) (no-node nil) (answer snake))
(node (name node3) (type answer)
      (question nil)
      (yes-node nil) (no-node nil) (answer cat))
(node (name node4) (type decision)
      (question "Does the animal fly?")
      (yes-node gen1) (no-node gen2) (answer dog))
(node (name gen1) (type answer)
      (question nil)
      (yes-node nil) (no-node nil) (answer bird))
(node (name gen2) (type answer)
      (question nil)
      (yes-node nil) (no-node nil) (answer dog))
```

Узел ответа `node4` заменен узлом принятия решений, который ссылается на два новых узла ответа. Кроме того, теперь применяемое по умолчанию значение `nil`, автоматически присваиваемое некоторым слотам конструкции `deftemplate`, после сохранения фактов стало заданным явно.

12.4 Обратный логический вывод

Система CLIPS в составе средств своей машины логического вывода непосредственно не реализует обратный логический вывод. Тем не менее обратный логический вывод может быть эмулирован с использованием правил прямого логического вывода системы CLIPS. В настоящем разделе будет показано, как можно создать на языке CLIPS простую систему обратного логического вывода. Следует отметить, что язык CLIPS разработан для применения в качестве языка прямого логического вывода, поэтому если для решения некоторой задачи в большей степени приемлем подход на основе обратного логического вывода, то следует использовать язык, в котором обратный логический вывод непосредственно реализован в составе средств машины логического вывода, такой как PROLOG.

В настоящем разделе система обратного логического вывода CLIPS будет создана с учетом описанных ниже возможностей и ограничений.

- Факты будут представлены как пары “атрибут–значение”.
- Обратный логический вывод будет начинаться с внесения в список фактов единственного начального атрибута `goal` (цель).
- В качестве условия в антецеденте правила будет проверяться только равенство атрибута конкретному значению.
- Единственным действием в антецеденте правила будет присваивание значения единственного атрибута.
- Если значение атрибута `goal` не может быть определено с помощью правил, то программа обратного логического вывода будет запрашивать предоставление значения для этого атрибута. Атрибутам не может присваиваться неопределенное значение.
- Атрибут может иметь только единственное значение. Гипотетические рассуждения о различных значениях атрибутов, полученных на основе разных правил, не поддерживаются.
- Неопределенность не будет представлена.

Алгоритм работы программы обратного логического вывода

Прежде чем приступить к разработке механизма обратного логического вывода и осуществлению с помощью языка CLIPS попыток реализации подхода,

основанного на правилах, необходимо рассмотреть процедурный алгоритм. Для определения значения атрибута goal с использованием подхода на основе обратного логического вывода с возможностями и ограничениями, описанными выше, может применяться следующая процедура на псевдокоде:

```
procedure Solve_Goal(goal)
   goal: the current goal to be solved
   if value of the goal attribute is known
      Return the value of the goal attribute.
   end if
   for each rule whose consequent is the goal
   attribute do
      call Attempt_Rule with the rule
      if Attempt_Rule succeeds then
         Assign the goal attribute the value
         indicated by the consequent of the rule.
         Return the value of the goal attribute.
      end if
   end do
   Ask the user for the value of the goal
   attribute.
   Set the goal attribute to the value supplied
   by the user.
   Return the value of the goal attribute.
end procedure
```

Атрибут goal передается процедуре Solve_Goal в качестве параметра. Эта процедура определяет значение атрибута goal и возвращает его. Вначале в процедуре Solve_Goal проверяется, известно ли уже значение атрибута goal. Это значение уже могло быть присвоено в консеквенте другого правила или задано пользователем программы обратного логического вывода. Если данное значение действительно известно, происходит его возврат.

Если же значение атрибута неизвестно, то в процедуре Solve_Goal предпринимается попытка определить это значение путем поиска правила, в консеквенте которого данному атрибуту присваивается значение. В процедуре Solve_Goal осуществляются попытки проверить каждое правило, в консеквенте которого атрибуту goal присваивается значение до тех пор, пока проверка одного из правил не завершится успешно. Для проведения таких попыток каждое правило с желаемым атрибутом goal передается процедуре Attempt_Rule (которая будет подробно рассматриваться немного позже). Если проверка антецедента правила, рассматриваемого в очередной попытке, оканчивается успешно, то считается, что правило успешно прошло проверку; в противном случае проверка считается

неудачной. Если проверка правила завершилась успешно, то значение атрибута в консеквенте правила присваивается атрибуту `goal` и возвращается процедурой `Solve_Goal`. Если же проверка правила оканчивается неудачей, то предпринимается попытка проверить следующее правило, в консеквенте которого присваивается значение атрибуту `goal`.

Если ни одна из проверок правил не завершается успешно, то для определения значения атрибута `goal` должен быть передан запрос пользователю. В таком случае процедура `Solve_Goal` возвращает значение, предоставленное пользователем.

Для определения того, выполнены ли условия в антецеденте правила, применяется процедура `Attempt_Rule`. Если условия в антецеденте правила выполнены, то консеквент правила может использоваться для присваивания значения атрибуту `goal`. Псевдокод этой процедуры приведен ниже.

```
procedure Attempt_Rule(rule)
   rule: rule to be attempted to solve goal
   for each condition in the antecedent
      of the rule do
      call Solve_Goal with condition attribute
      if the value returned by solve_goal is not
         equal to the value required by the condition
      then
         Return unsuccessful.
      end if
   end for
   Return successful
end procedure
```

Работа процедуры `Attempt_Rule` начинается с проверки первого условия правила и попытки доказать его; после этого осуществляются попытки доказать остальные условия правила. Для определения того, выполнено ли условие, процедура `Attempt_Rule` должна иметь информацию о том, какое значение имеет атрибут, проверяемый в данном условии. Для определения этого значения рекурсивно вызывается процедура `Solve_Goal`. Если значение, возвращенное процедурой `Solve_Goal`, не равно значению, возвращенному при проверке условия, то процедура `Attempt_Rule` завершает свою работу и возвращает значение, свидетельствующее о неудаче (напомним, что в условиях проверяется только равенство). В противном случае проверяется очередное условие правила. Если выполнены все условия правила, то процедура `Attempt_Rule` оканчивает свою работу, возвращая значение, свидетельствующее об успешном завершении.

Представление правил обратного логического вывода в языке CLIPS

И в данном случае первым шагом к решению задачи становится определение того, как должны быть представлены знания. Система CLIPS не осуществляет автоматически обратный логический вывод, поэтому удобнее всего было бы представлять правила обратного логического вывода как факты, для того чтобы антецеденты и консеквенты правил можно было проверять с помощью правил, действующих в качестве механизма обратного логического вывода. Конструкция `deftemplate`, предназначенная для представления правил обратного логического вывода, показана ниже. Эта конструкция будет храниться в конструкции `defmodule` с именем `BC` (которая приведена в конце данного подраздела, после описания всех конструкций `deftemplate`, необходимых для данной машины обратного логического вывода).

```
(deftemplate BC::rule
   (multislot if)
   (multislot then))
```

Антецедент и консеквент каждого правила должны храниться в слотах `if` и `then` соответственно. Каждый антецедент содержит либо одну пару "атрибут–значение" в приведенном ниже формате, либо ряд таких пар "атрибут–значение", соединенных символом `and`.

```
<attribute> is <value>
```

В консеквент каждого правила допускается включение только одной пары "атрибут–значение".

В качестве примера представления правил с использованием этого формата рассмотрим дерево решений, приведенное на рис. 12.2. Данное дерево можно легко представить в виде правил, в которых применяются пары "атрибут–значение" (см. выше). Ниже приведен псевдокод таких преобразованных правил.

```
IF main-course is red-meat
THEN best-color is red
IF main-course is poultry and
   meal-is-turkey is yes
THEN best-color is red
IF main-course is poultry and
   meal-is-turkey is no
THEN best-color is white
IF main-course is fish
THEN best-color is white
```

Атрибутами, используемыми в этих правилах, являются `main-course`, `meal-is-turkey` и `best-color`. Атрибут `main-course` соответствует ответу, определяемому с помощью вопроса из дерева решений: `"What is the main course?"` (Что является основным блюдом?) Атрибут `meal-is-turkey` соответствует ответу, определяемому с помощью вопроса: `"Is the main course turkey?"` (Является ли основным блюдом индейка?) Обратите внимание на то, что ребро дерева решений, определяющее, что наиболее подходящий тип вина неизвестен, не представлено в виде правила, поскольку одно из ограничений рассматриваемой программы обратного логического вывода состоит в том, что с ее помощью нельзя представить неизвестные значения. Если основным блюдом не является такое блюдо, для которого в правилах предусмотрен ответ, то пользователю поступает запрос указать значение атрибута `best-color`.

Следующая конструкция `deffacts` показывает, как можно представить правила, касающиеся выбора наиболее подходящего вина, с помощью формата правил обратного логического вывода. Эти факты, определяющие правило, не являются неотъемлемой частью машины обратного логического вывода, поэтому размещаются в модуле `MAIN` (напомним, что модуль `MAIN` импортирует конструкции из всех других модулей, поэтому конструкция `deftemplate` с именем `rule` также будет в нем видимой).

```
(deffacts MAIN::wine-rules
   (rule (if main-course is red-meat)
         (then best-color is red))
   (rule (if main-course is fish)
         (then best-color is white))
   (rule (if main-course is poultry and
             meal-is-turkey is yes)
         (then best-color is red))
   (rule (if main-course is poultry and
             meal-is-turkey is no)
   (then best-color is white)))
```

Такой способ представления обеспечивает значительную гибкость при манипулировании правилами обратного логического вывода. Например, если определено, что атрибут `main-course` имеет значение `poultry`, то факты

```
(rule (if main-course is red-meat)
      (then best-color is red))
```

и

```
(rule (if main-course is fish)
      (then best-color is white))
```

могут быть удалены из списка фактов для указания на то, что соответствующие правила не применимы, а факты

```
(rule (if main-course is poultry and
          meal-is-turkey is yes)
      (then best-color is red))
```

и

```
(rule (if main-course is poultry and
          meal-is-turkey is no)
      (then best-color is white))
```

могут быть соответствующим образом модифицированы с получением фактов:

```
(rule (if meal-is-turkey is yes)
      (then best-color is red))
```

и

```
(rule (if meal-is-turkey is no)
      (then best-color is white))
```

для указания на то, что первое условие этих двух правил было выполнено.

По мере осуществления обратного логического вывода вырабатываются подцели, предназначенные для определения значений атрибутов. Для представления информации об атрибутах `goal` потребуется определенный факт. Представление атрибутов `goal` будет осуществляться с использованием упорядоченных фактов, которые имеют следующий формат:

```
(deftemplate BC::goal
   (slot attribute))
```

Первоначально атрибутом `goal` становится атрибут `best-color`, который может быть представлен с помощью конструкции `deffacts`:

```
(deffacts MAIN::initial-goal
   (goal (attribute best-color)))
```

После определения значений атрибутов их необходимо сохранить с помощью следующей конструкции `deftemplate`:

```
(deftemplate BC::attribute
   (slot name)
   (slot value))
```

Теперь, после определения всех конструкций `deftemplate`, можно привести определение модуля `BC`, как показано ниже. Напомним, что при загрузке файла с конструкциями, которые должна содержать конструкция `defmodule`, другие конструкции должны быть определены прежде, чем содержащие их конструкции.

```
(defmodule BC
   (export deftemplate rule goal attribute))
```

Машина обратного логического вывода на языке CLIPS

Машина обратного логического вывода может быть реализована с использованием двух множеств правил. Первая группа правил применяется для выработки целей, при достижении которых выявляются значения атрибутов, и для передачи запроса пользователю, чтобы он предоставил значения атрибутов, если эти значения невозможно определить с помощью правил. Вторая группа правил осуществляет операции обновления. К числу операций обновления относятся операции модификации правил после выполнения их условий и операции удаления целей после их достижения. Первое множество правил приведено ниже.

```
(defrule BC::attempt-rule
   (goal (attribute ?g-name))
   (rule (if ?a-name $?)
         (then ?g-name $?))
   (not (attribute (name ?a-name)))
   (not (goal (attribute ?a-name)))
   =>
   (assert (goal (attribute ?a-name))))
(defrule BC::ask-attribute-value
   ?goal <- (goal (attribute ?g-name))
   (not (attribute (name ?g-name)))
   (not (rule (then ?g-name $?)))
   =>
   (retract ?goal)
   (printout t "What is the value of " ?g-name "? ")
   (assert (attribute (name ?g-name)
   (value (read)))))
```

С помощью правила `attempt-rule` осуществляется поиск правил, антецеденты которых задают значение атрибута для атрибута `goal`. Первый шаблон сопоставляется с фактом `goal`. Во втором шаблоне происходит поиск всех правил, в антецедентах которых присваивается значение атрибуту `goal`. В третьем шаблоне проверяется, определено ли уже значение атрибута `goal`. С помощью четвертого шаблона подтверждается, что цель определения значения атрибута еще не достигнута. Для каждого найденного правила в правой части правила `attempt-rule` в список фактов вносится цель определения значения атрибута, проверенного в первом условии правила.

Правило `ask-attribute-value` весьма напоминает правило `attempt-rule`. Первые два шаблона этого правила идентичны. А в третьем шаблоне про-

веряется, что отсутствуют оставшиеся правила, которые могут использоваться для определения значения атрибута goal. В таком случае пользователю передается запрос, чтобы он сам задал значение этого атрибута. В список фактов вносится факт, представляющий значение атрибута, и извлекается факт goal, относящийся к этому атрибуту.

Для обновления правил обратного логического вывода и целей, которые представлены как факты, используются приведенные ниже четыре правила. Этим правилам присвоена значимость (salience), равная 100, для того чтобы обновления происходили до того, как будут сделаны какие-либо попытки вырабатывать новые цели или запрашивать у пользователя значения атрибутов.

```
(defrule BC::goal-satisfied
   (declare (salience 100))
   ?goal <- (goal (attribute ?g-name))
   (attribute (name ?g-name))
   =>
   (retract ?goal))
(defrule BC::rule-satisfied
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name)
              (value ?a-value))
   ?rule <- (rule (if ?a-name is ?a-value)
                  (then ?g-name is ?g-value))
   =>
   (retract ?rule)
   (assert (attribute (name ?g-name)
                      (value ?g-value))))
(defrule BC::remove-rule-no-match
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name) (value ?a-value))
   ?rule <- (rule (if ?a-name is ~?a-value)
                  (then ?g-name is ?g-value))
   =>
   (retract ?rule))
(defrule BC::modify-rule-match
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name) (value ?a-value))
   ?rule <- (rule (if ?a-name is ?a-value and
```

```
                    $?rest-if)
          (then ?g-name is ?g-value))
   =>
   (retract ?rule)
   (modify ?rule (if $?rest-if)))
```

С помощью правила `goal-satisfied` удаляются все цели, для которых было определено значение атрибута.

С помощью правила `rule-satisfied` осуществляется поиск всех правил, в которых имеется единственное оставшееся условие. Если существует атрибут, который соответствует этому оставшемуся условию, и имеется цель определения значения этого атрибута, то в список фактов вносится значение атрибута из консеквента данного правила.

С помощью правила `remove-rule-no-match` осуществляется поиск правил, антецеденты которых могут предоставить значение атрибута для атрибута `goal` и которые содержат одно или несколько таких условий, что первое из них конфликтует со значением, присвоенным атрибуту. Если обнаруживается такая ситуация, соответствующее правило удаляется из списка фактов, поскольку является неприменимым.

С помощью правила `modify-rule-match` осуществляется поиск правил, антецеденты которых могут предоставить значение атрибута для атрибута `goal` и которые содержат два или несколько таких условий, что первое из них выполняется при наличии некоторого значения, присвоенного атрибуту. Если обнаруживается такое правило, то первое условие удаляется из этого правила, чтобы в нем присутствовали только оставшиеся условия, подлежащие проверке.

Итак, теперь представлены все правила обратного логического вывода, и все, что требуется для запуска процесса обратного логического вывода, состоит в том, чтобы перевести фокус на модуль `BC`. Это действие можно выполнить, добавив к модулю `MAIN` следующее правило:

```
(defrule MAIN::start-BC
   =>
   (focus BC))
```

Поэтапная трассировка работы программы обратного логического вывода

За тем, как действует машина обратного логического вывода на языке CLIPS, реализованная с помощью правил, можно наблюдать, отслеживания ее работу. Ниже показано начальное состояние системы после выполнения команды `reset`, при условии, что загружены конструкции `deffacts` с именами `wine-rules` и `initial-goal`, а также правила машины обратного логического вывода

(и в данном случае некоторые строки вывода обозначена отступами для удобства чтения).

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (facts)↵
f-0      (initial-fact)
f-1      (goal (attribute best-color))
f-2      (rule (if main-course is red-meat)
               (then best-color is red))
f-3      (rule (if main-course is fish)
               (then best-color is white))
f-4      (rule (if main-course is poultry and
                   meal-is-turkey is yes)
               (then best-color is red))
f-5      (rule (if main-course is poultry and
                   meal-is-turkey is no)
               (then best-color is white))
For a total of 6 facts.
CLIPS> (agenda)↵
0     start-BC: f-0
For a total of 1 activation.
CLIPS>
```

Правило `start-BC` переводит фокус на модуль `BC`. После запуска этого правила текущий фокус перемещается на модуль `BC` следующим образом:

```
CLIPS> (run 1)↵
CLIPS> (agenda)↵
0     attempt-rule: f-1,f-5,,
0     attempt-rule: f-1,f-4,,
0     attempt-rule: f-1,f-3,,
0     attempt-rule: f-1,f-2,,
For a total of 4 activations.
CLIPS>
```

Обратите внимание на то, что рабочий список правил содержит четыре активизации правила `attempt-rule`. Начальная цель состоит в определении значения атрибута `best-color`, о чем свидетельствует факт `f-1`. Значение атрибуту `best-color` присваивается в каждом из фактов `f-2`, `f-3`, `f-4` и `f-5`, заданных консеквентами правил, поэтому необходимо попытаться проверить каждое из этих правил, чтобы определить значение атрибута `best-color`.

Следующим шагом в ходе выполнения программы становится запуск первой активизации правила attempt-rule. Но, как показано ниже, перед запуском этого правила активизируются отслеживаемые элементы rules и facts.

```
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (run 1)↵
FIRE     1 attempt-rule: f-1,f-5,,
==> f-6     (goal (attribute main-course))
CLIPS> (agenda)↵
0     ask-attribute-value: f-6,,
For a total of 1 activation.
CLIPS>
```

Запуск правила attempt-rule отчасти обусловлен наличием факта f-5, который представляет показанное ниже правило обратного логического вывода.

```
IF main-course is poultry and
   meal-is-turkey is no
THEN best-color is white
```

Возможность применить это правило для присваивания значения атрибуту best-color появится только после того, как будут выполнены условные элементы в антецедентах правила. Для проверки первого условия требуется иметь значение атрибута main-course. Поскольку значение этого атрибута не известно, создается цель для его определения, которая представляется с помощью факта f-6. Правила, присваивающие значение атрибуту main-course, отсутствуют, поэтому активизируется правило ask-attribute-value.

После перехода к дальнейшему выполнению запускается правило ask-attribute-value для определения значения атрибута main-course:

```
CLIPS> (run 1)↵
FIRE     1 ask-attribute-value: f-6,,
<== f-6     (goal (attribute main-course))
What is the value of main-course? poultry↵
==> f-7     (attribute (name main-course)
                       (value poultry))
CLIPS> (agenda)↵
100   remove-rule-no-match: f-1,f-7,f-3
100   remove-rule-no-match: f-1,f-7,f-2
100   modify-rule-match: f-1,f-7,f-5
100   modify-rule-match: f-1,f-7,f-4
For a total of 4 activations.
CLIPS>
```

Пользователю был передан запрос, чтобы он ввел значение атрибута main-course, и это значение было получено, поэтому цель f-6 получения данного атрибута уничтожается. Значение, введенное пользователем, вносится в список фактов в виде факта attribute с индексом f-7. Вставка этого факта в список фактов приводит к тому, что в рабочий список правил помещаются четыре новые активизации. Для каждого из правил, представленных фактами f-4 и f-5, в качестве первого условия требуется, чтобы атрибут main-course имел значение poultry. Поскольку значением атрибута main-course является poultry, оба этих факта с определениями правил должны быть модифицированы с учетом того, что их первое условие было выполнено. Таким образом, оба факта приводят к двукратной активизации правила modify-rule-match. А для обоих правил, представленных фактами f-2 и f-3, требуется, чтобы в качестве первого условия был задан атрибут main-course, имеющий некоторое значение, отличное от poultry. Поэтому больше ни один из этих фактов не применим. Активизируется правило remove-rule-no-match, что приводит к удалению обоих этих фактов.

После того как будет разрешен запуск двух активизаций правила remove-rule-no-match, формируются следующие выходные результаты:

```
CLIPS> (run 2)↵
FIRE     1 remove-rule-no-match: f-1,f-7,f-3
<== f-3     (rule (if main-course is fish)
                  (then best-color is white))
FIRE     2 remove-rule-no-match: f-1,f-7,f-2
<== f-2     (rule (if main-course is red-meat)
                  (then best-color is red))
CLIPS> (agenda)↵
100   modify-rule-match: f-1,f-7,f-5
100   modify-rule-match: f-1,f-7,f-4
For a total of 2 activations.
CLIPS>
```

Факты f-2 и f-3 удаляются из списка фактов; это говорит о том, что правила, представленные этими фактами, больше не применимы. После удаления указанных фактов из рабочего списка правил удаляются относящиеся к ним активизации правила attempt-rule.

После перехода к выполнению двух активизаций правила modify-rule-match формируется следующий вывод:

```
CLIPS> (run 2)↵
FIRE     1 modify-rule-match: f-1,f-7,f-5
<== f-5     (rule (if main-course is poultry and
                      meal-is-turkey is no)
```

```
                  (then best-color is white))
==> f-8     (rule (if meal-is-turkey is no)
                  (then best-color is white))
FIRE     2 modify-rule-match: f-7,f-4
<== f-4     (rule (if main-course is poultry and
                      meal-is-turkey is yes)
                  (then best-color is red))
==> f-9     (rule (if meal-is-turkey is yes)
                  (then best-color is red))
CLIPS> (agenda)↵
0     attempt-rule: f-1,f-9,,
0     attempt-rule: f-1,f-8,,
For a total of 2 activations.
CLIPS>
```

Первый запуск правила `modify-rule-match` отчасти обусловлен наличием факта `f-5`, который представляет следующее правило обратного логического вывода:

```
IF main-course is poultry and
   meal-is-turkey is no
THEN best-color is white
```

Действия, предусмотренные в правиле `modify-match-rule`, модифицируют указанное правило обратного логического вывода таким образом:

```
IF meal-is-turkey is no
THEN best-color is white
```

Новый факт, представляющий модифицированное правило, вносится в список фактов под индексом `f-8`. Этот новый факт представляет условия начального правила, которые остаются после выполнения первого условия, и вызывает еще одну активизацию правила `attempt-rule` применительно к данному правилу обратного логического вывода. В этой новой активизации в список фактов вносится новая цель для определения значения атрибута "`meal-is-turkey`", поэтому консеквент данного правила может применяться для присваивания значения атрибута `best-color`.

Второй запуск правила `modify-rule-match` осуществляется аналогично первому запуску. Факт, представляющий правило обратного логического вывода,

```
IF main-course is poultry and
   meal-is-turkey is red
THEN best-color is red
```

модифицируется и принимает вид следующего правила, для представления которого в список фактов вносится факт с индексом `f-9`:

```
IF meal-is-turkey is yes
THEN best-color is red
```

Аналогичным образом, этот новый факт активизирует правило attempt-rule для замещения активизации, потерянной после извлечения факта, представляющего это правило.

После того как будет разрешен запуск первой активизации правила attempt-rrule, формируется следующий вывод:

```
CLIPS> (run 1)↵
FIRE     1 attempt-rule: f-1,f-9,,
==> f-10       (goal (attribute meal-is-turkey))
CLIPS> (agenda)↵
0     ask-attribute-value: f-10,,
For a total of 1 activation.
CLIPS>
```

В список фактов вносится факт с индексом f-10, представляющий цель определения значения атрибута meal-is-turkey. Значение этого атрибута не присваивается ни в одном из правил, поэтому активизируется правило ask-attribute-value для определения данного значения.

После того как будет разрешен запуск правила ask-attribute-value, вырабатывается следующий вывод:

```
CLIPS> (run 1)↵
FIRE     1 ask-attribute-value: f-10,,
<== f-10    (goal (attribute meal-is-turkey))
What is the value of meal-is-turkey? yes↵
==> f-11    (attribute (name meal-is-turkey)
                       (value yes))
CLIPS> (agenda)↵
100   rule-satisfied: f-1,f-11,f-9
100   remove-rule-no-match: f-1,f-11,f-8
For a total of 2 activations.
CLIPS>
```

В результате запуска этого правила в список фактов вносится факт attribute, представляющий значение атрибута meal-is-turkey. Кроме того, удаляется факт goal, предназначенный для определения значения этого атрибута. Появление в списке фактов нового факта attribute вызывает две активизации. Первая активизация относится к правилу remove-rule-no-match. Но первое условие факта f-8 несовместимо со значением нового атрибута, и правило, представленное этим фактом, больше не применимо, поэтому данный факт должен быть удален. Вторая активизация относится к правилу rule-satisfied. Остав-

шееся условие факта f-8 удовлетворяется с помощью нового факта attribute, поэтому может быть применен консеквент факта f-8.

Для завершения этого процесса обратного логического вывода осталось лишь выполнить запуск последних активизированных правил:

```
CLIPS> (run)↵
FIRE     1 rule-satisfied: f-1,f-11,f-9
<== f-9  (rule (if meal-is-turkey is yes)
               (then best-color is red))
==> f-12    (attribute (name best-color)
                       (value red))
FIRE     2 goal-satisfied: f-1,f-12
<== f-1     (goal (attribute best-color))
CLIPS> (agenda)↵
CLIPS> (facts)↵
f-0      (initial-fact)
f-7      (attribute (name main-course)
                    (value poultry))
f-8      (rule (if meal-is-turkey is no)
               (then best-color is white))
f-11     (attribute (name meal-is-turkey)
                    (value yes))
f-12     (attribute (name best-color) (value red))
For a total of 5 facts.
CLIPS>
```

Происходит запуск правила rule-satisfied для присваивания значения атрибуту best-color в составе консеквента правила, представленного фактом f-9. Факт attribute, внесенный в список фактов этим правилом, соответствует факту goal, оставшемуся в списке фактов. Активизируется правило goal-satisfied, после чего осуществляется его запуск для удаления оставшегося факта goal.

Результаты выполнения команды agenda показывают, что не осталось больше правил, запуск которых мог бы быть выполнен. А команда facts показывает атрибуты, которым были присвоены значения. Факт f-12 свидетельствует о том, что исходному целевому атрибуту best-color было присвоено значение red.

12.5 Задача текущего контроля

В этом разделе приведен пример поэтапной разработки программы CLIPS, предназначенный для решения одной несложной задачи. Этапы разработки включают исходное описание задачи, принятие предположений о характере задачи

и составление первоначальных определений для представления знаний о задаче. За этим следует постепенное накопление правил, позволяющих решить данную задачу.

Формулировка задачи

В настоящем разделе рассматривается решение задачи, которая может служить примером создания простой системы текущего контроля. Обычно задачи текущего контроля легко поддаются решению с помощью языков прямого логического вывода, основанных на правилах, в силу самого характера этих задач, управляемых данными. При этом, вообще говоря, в каждом цикле работы программы производится чтение входных значений или значений показаний датчиков. Затем формируются логические выводы, до тех пор, пока не достигаются все возможные заключения, которые могут быть получены на основании этих входных данных. Такой принцип работы совместим с подходом к созданию программ, управляемых данными, в которых цепь рассуждений строится от данных к заключениям, выводимым на основании данных.

В рассматриваемом примере решается такая задача текущего контроля, которая находит очень широкое распространение на практике. Гипотетическая обрабатывающая установка состоит из нескольких устройств, работа которых подлежит непрерывному контролю. Функционирование некоторых устройств зависит от нормальной работы других устройств. На каждом устройстве установлены один или несколько датчиков, вырабатывающих числовые показания, по которым можно судить об общем состоянии устройства. Каждый датчик вырабатывает показания, свидетельствующие о том, находится ли контролируемый параметр в пределах рабочего и допустимого диапазонов. Для этого используются значения нижнего предела рабочего диапазона (Low Guard Line — LGL), нижнего предела допустимого диапазона (Low Red Line — LRL), верхнего предела рабочего диапазона (High Guard Line — HGL) и верхнего предела допустимого диапазона (High Red Line — HRL). Показания, не выходящие за пределы нижнего и верхнего значений рабочего диапазона, рассматриваются как нормальные. Показания, находящиеся выше верхнего предела рабочего диапазона, но ниже верхнего предела допустимого диапазона или ниже нижнего предела рабочего диапазона, но выше нижнего предела допустимого диапазона, рассматриваются как приемлемые, но свидетельствуют о том, что устройство может вскоре стать неисправным. А показания выше верхнего предела допустимого диапазона или ниже нижнего предела допустимого диапазона свидетельствуют о том, что устройство неисправно и его работа должна быть остановлена. При выходе показаний датчика любого устройства за пределы рабочего диапазона (в верхний или нижний допустимый диапазон) должны формироваться предупреждающие сообщения. Кроме того, любое устройство, для которого показания датчиков остаются в одном из допустимых

диапазонов слишком продолжительное время, должно быть остановлено. Итоговые сведения о том, какие действия должны быть предприняты при обнаружении конкретных значений датчика, приведены в табл. 12.1.

Таблица 12.1. Действия, выполняемые при обнаружении различных показаний датчика

Значение показаний датчика	Действие
Меньше или равно нижнему пределу допустимого диапазона	Остановить устройство
Больше нижнего предела допустимого диапазона и меньше или равно нижнему пределу рабочего диапазона	Выдать предупреждающее сообщение или остановить устройство
Больше нижнего предела рабочего диапазона и меньше верхнего предела рабочего диапазона	Не предпринимать никаких действий
Больше или равно верхнему пределу рабочего диапазона и меньше верхнего предела допустимого диапазона	Выдать предупреждающее сообщение или остановить устройство
Больше или равно верхнему пределу допустимого диапазона	Остановить устройство

Программа текущего контроля должна обладать способностью считывать показания датчика, оценивать данные, полученные от датчиков, а также выдавать предупреждающие сообщения и останавливать работу устройств на основании оценки показаний датчиков и наблюдаемых тенденций. Пример результатов выполнения программы текущего контроля может выглядеть примерно так:

```
Cycle 20 - Sensor 4 in high guard line
Cycle 25 - Sensor 4 in high red line
   Shutting down device 4
Cycle 32 - Sensor 3 in low guard line
Cycle 38 - Sensor 1 in high guard line for 6 cycles
   Shutting down device 1
```

На рис. 12.5 показаны соединения между устройствами и датчиками, применяемыми в установке, которая рассматривается в данном примере, а в табл. 12.2 перечислены пределы, контролируемые каждым датчиком.

На определенном этапе решения задачи общее описание этой задачи используется для уточнения конкретных деталей, для которых также необходимо найти решение, прежде чем реализовать общее решение задачи. Обычно в ходе этого требуется неоднократно консультироваться с экспертами, обладающими знаниями в данной предметной области, но не всегда способными полностью сформулировать задание, для выполнения которого должна быть предназначена экспертная система. Поэтому приходится разрабатывать прототипы решения задачи, позво-

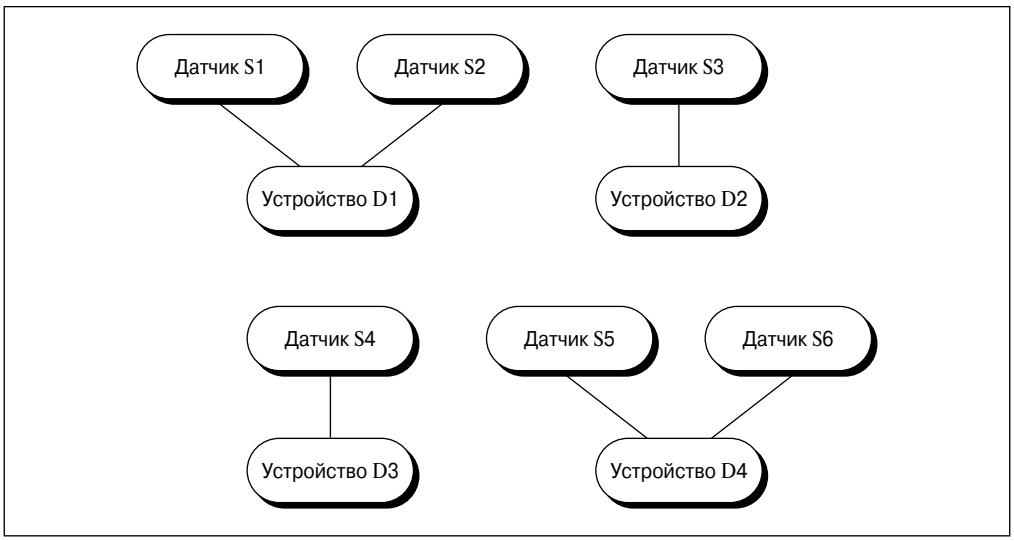


Рис. 12.5. Схема соединений устройств и датчиков в контролируемой системе

Таблица 12.2. Пределы, контролируемые каждым датчиком

Датчик	Нижний предел допустимого диапазона	Нижний предел рабочего диапазона	Верхний предел рабочего диапазона	Верхний предел допустимого диапазона
S1	60	70	120	130
S2	20	40	160	180
S3	60	70	120	130
S4	60	70	120	130
S5	65	70	120	125
S6	110	115	125	130

ляющие выявить детали, недостающие в исходной постановке задачи. После этого с помощью консультаций с экспертами можно уточнить необходимые детали и разработать еще один прототип, с помощью которого, возможно, удастся выявить какие-то другие недостающие детали в спецификации задачи. В конечном итоге указанный итеративный подход к разработке программного обеспечения позволяет полностью составить спецификацию задачи.

Пока еще остались без ответа многие вопросы, касающиеся деталей спецификации данной задачи: “Как должна быть представлена информация о датчиках и устройствах? Насколько общими или конкретными должны быть факты и правила? Как должна осуществляться выборка данных от датчиков? Как долго показания датчика должны находиться в одном из допустимых диапазонов, для

того чтобы потребовался останов устройства, к которому он относится? Какие действия должны выполняться программой текущего контроля при обнаружении неисправного устройства?"

Уточнение деталей, с которого должна начинаться разработка

Одна из основных проблем, с которой приходится сталкиваться при создании экспертных систем, состоит в том, что задача чаще всего имеет некачественную постановку. Безусловно, целью создания экспертной системы является замена эксперта, но никто кроме эксперта не обладает знаниями, достаточными для того, чтобы определить все нюансы своей работы. Обычно известно лишь то, какие действия должна осуществлять система, но не совсем ясно, в какой форме должны подготавливаться эти действия. Эксперт может испытывать затруднения при попытке точно сформулировать, какие шаги он выполняет в процессе выработки решения. Разумеется, сам подход к созданию экспертных систем, основанный на методах итеративной разработки, который реализуется в этой области так естественно, позволяет относительно легко создавать экспертные системы, которые решают недостаточно полно определенные задачи. Но это отнюдь не означает, что экспертные системы позволяют находить решения задач, которые еще никогда не решались раньше, или задач, само определение которых еще не понято окончательно.

Поэтому к разработке экспертной системы для данного примера текущего контроля можно будет приступить только после уточнения еще некоторых аспектов задачи. Прежде всего необходимо определить, какие действия должна осуществлять экспертная система, в том числе описать исходную информацию и информацию, которая должна вырабатываться системой. Но это не означает, что созданные при этом спецификации не подлежат изменению. Дальнейший ход разработки программы может показать, что область охвата данной задачи должна быть сужена или, возможно, расширена, что повлияет на определение исходных данных и конечных результатов. Кроме того, необходимо принять некоторые предположения и уточнить подробности того, как экспертная система должна выполнять поставленные перед ней задания. Решения такого рода также не являются окончательными. Эксперты часто могут показать, как они приходят к решению задачи, но не могут легко сформулировать используемые ими правила. При описании правил, которыми они пользуются, эксперты иногда не упоминают очевидные для них подробности или забывают об исключениях.

При решении рассматриваемой задачи текущего контроля необходимо с самого начала выбрать несколько важных направлений. Первым из них является способ реализации. Должно ли быть решение узко направлено на осуществление всех нюансов спецификации задачи или должно оставаться достаточно общим, чтобы

существовала возможность в дальнейшем легко модернизировать или модифицировать эту систему? Иными словами, один вариант состоит в том, что могут быть написаны конкретные правила для каждого из контролируемых устройств, а второй может предусматривать написание общего правила для контроля над всеми устройствами. Складывается впечатление, что для данной задачи в большей степени подходит вариант с использованием общих правил, поскольку отдельные моделируемые устройства и датчики не имеют уникальных характеристик. Благодаря такой общности должно упроститься введение в дальнейшем дополнительных устройств и датчиков.

К тому же отсутствуют подробные сведения о том, как должна происходить передача управления в системе текущего контроля. Но в условиях рассматриваемой задачи может применяться простой цикл текущего контроля. Каждый цикл текущего контроля должен состоять из трех фаз. В течение первой фазы считываются значения, поступающие от датчиков, во второй фазе выполняется анализ показаний датчиков, а в третьей осуществляются все действия, соответствующие обстоятельствам.

Необходимо также принять определенные предположения в части того, как должна происходить выборка данных от датчиков. Должны ли эти данные считываться непосредственно с датчиков? Нужно ли обеспечить моделирование показаний датчиков? Всегда ли данные, поступающие от датчиков, будут доступными в тот момент, когда потребуется? Являются ли показания датчиков надежными или подвержены ошибкам? В обычных ситуациях, когда разрабатывается прототип, можно провести интервью с экспертами для определения этой информации, но в ходе решения данной учебной задачи мы сами должны принять разумные предположения, чтобы уточнить все детали.

Приведенные выше вопросы показывают, как ликвидируются упущения, которые могут быть обнаружены в некачественной постановке задачи. Список предположений, вопросов и возможных несовместимостей, относящийся к спецификации решаемой задачи, необходимо сопровождать в течение всей разработки. А если разработка программы ведется по итерационному принципу, такой список должен стать сосредоточением всех дискуссий, чтобы экспертам было проще следить за тем, всегда ли постановка задачи совпадает со взглядами экспертов на то, как должна быть решена задача. Ниже приведена часть указанного списка для задачи текущего контроля, которая начинается с предположений.

- Показания датчиков всегда надежны и всегда доступны по запросу.
- Должна быть предусмотрена возможность считывать показания датчиков непосредственно с датчиков. Программа должна также предоставлять возможность поддерживать моделируемые показания датчиков.
- На остановленном устройстве показания датчиков не должны контролироваться.

- Предполагается, что действия, необходимость в проведении которых указана системой текущего контроля, должны быть обязательно реализованы (т.е. предполагается, что их либо выполняет внимательный персонал установки, либо контроль над устройствами непосредственно осуществляется программой).
- Процедура решения задачи разделяется на три фазы: чтение показаний датчиков, анализ показаний датчиков и выполнение соответствующих действий, таких как останов устройства.

Кроме уточнения всех нюансов постановки задачи, необходимо принять решение в отношении всех деталей реализации программы. К этим деталям относится то, как должна быть представлена доступная информация, как осуществляется передача управления и как проводится проверка экспертной системы.

Определения структур представления знаний

И в данном случае приступим к решению задачи с уточнения того, как должны быть представлены знания. Удобным вступительным шагом к решению данной задачи может стать попытка закодировать знания, как показано на рис. 12.5 и в табл. 12.2. Для описания каждого из устройств будет использоваться следующая конструкция `deftemplate`:

```
(defmodule MAIN (export ?ALL))
(deftemplate MAIN::device
   (slot name (type SYMBOL))
   (slot status (allowed-values on off)))
```

В этом определении слот `name` содержит имя устройства, а слот `status` показывает, включено данное устройство или выключено. Используя рис. 12.5 и приняв предположение о том, что все устройства первоначально были включены, опишем начальное состояние устройств с помощью такой конструкции `deffacts`:

```
(deffacts MAIN::device-information
(device (name D1) (status on))
(device (name D2) (status on))
(device (name D3) (status on))
(device (name D4) (status on)))
```

На рис. 12.5 также показано, какие датчики связаны с теми или иными устройствами. Для представления этого отношения будет использоваться следующая конструкция `deftemplate`:

```
(deftemplate MAIN::sensor
(slot name (type SYMBOL))
(slot device (type SYMBOL))
(slot raw-value (type SYMBOL NUMBER)
```

```
                (allowed-symbols none)
                (default none))
   (slot state (allowed-values low-red-line
                               low-guard-line
                               normal
                               high-red-line
                               high-guard-line)
               (default normal))
   (slot low-red-line (type NUMBER))
   (slot low-guard-line (type NUMBER))
   (slot high-guard-line (type NUMBER))
   (slot high-red-line (type NUMBER)))
```

В этом определении слот `name` содержит имя датчика, а слот `device` — имя устройства, к которому подключен этот датчик. Слот `raw-value` содержит значение данных, полученное непосредственно от датчика и еще не подвергшееся обработке. Слот `state` содержит информацию о текущем состоянии показаний датчика (например, находятся ли они в рабочем диапазоне, в нижнем допустимом диапазоне, в верхнем допустимом диапазоне и т.д.). Слоты `low-red-line` (нижний предел допустимого диапазона), `low-guard-line` (нижний предел рабочего диапазона), `expected-average-value` (ожидаемое среднее значение), `high-guard-line` (верхний предел рабочего диапазона) и `high-red-line` (верхний предел допустимого диапазона) используются для хранения информации, описанной в табл. 12.2. Для описания датчиков, представленных на рис. 12.5, может применяться такая конструкция `deffacts`:

```
(deffacts MAIN::sensor-information
   (sensor (name S1) (device D1)
           (low-red-line 60) (low-guard-line 70)
           (high-guard-line 120)
           (high-red-line 130))
   (sensor (name S2) (device D1)
           (low-red-line 20) (low-guard-line 40)
           (high-guard-line 160)
           (high-red-line 180))
   (sensor (name S3) (device D2)
           (low-red-line 60) (low-guard-line 70)
           (high-guard-line 120)
           (high-red-line 130))
   (sensor (name S4) (device D3)
           (low-red-line 60) (low-guard-line 70)
           (high-guard-line 120)
```

```
        (high-red-line 130))
  (sensor (name S5) (device D4)
        (low-red-line 65) (low-guard-line 70)
        (high-guard-line 120)
        (high-red-line 125))
  (sensor (name S6) (device D4)
        (low-red-line 110) (low-guard-line 115)
        (high-guard-line 125)
        (high-red-line 130)))
```

Как уже было сказано, данная система текущего контроля должна действовать циклически, поэтому необходимо предусмотреть факт, обозначающий текущий номер цикла. Первый цикл обозначается номером один, после чего значение номера наращивается с наступлением каждого нового цикла. Упорядоченный факт, используемый для представления этой информации, имеет следующий формат:

```
(cycle <number>)
```

В этом определении параметр `<number>` обозначает номер текущего цикла. Кроме того, поскольку показания датчиков могут поступать из нескольких источников (например, если один источник предназначен для моделирования, а другой — для реальной эксплуатации), то желательно иметь в наличии факт, обозначающий источник показаний датчиков. Этот факт можно представить с помощью такого формата:

```
(data-source <source>)
```

В этом определении параметр `<source>` обозначает имя экземпляра, представляющего тот источник, из которого считываются данные. Экземпляр `<source>` является экземпляром класса `DATA-SOURCE`, который будет вскоре описан. Потенциальными источниками могут служить датчики, программа моделирования, текстовый файл, множество фактов или данные, введенные пользователем.

Ниже приведена конструкция `deffacts`, содержащая начальную информацию:

```
(deffacts MAIN::cycle-start
(data-source [user])
(cycle 0))
```

Обратите внимание на то, что показания датчиков может вводить пользователь. Для данного примера ввод показаний датчиков с клавиатуры удобнее, чем чтение данных из файла.

Управление работой программы

В приведенном выше списке предположений, принятых до начала решения этой задачи, указано, что процесс текущего контроля состоит из трех конкретных фаз. В первой фазе считываются введенные пользователем, моделируемые или действительные значения, полученные от датчиков. Во второй фазе проверяются условия выхода показаний датчиков за пределы рабочего и допустимого диапазонов и определяются все складывающиеся при этом тенденции. После выявления сложившихся тенденций в показаниях датчиков система текущего контроля может выдать все соответствующие предупреждающие сообщения, остановить неисправное оборудование и перезапустить оборудование, которое снова вернулось в рабочее состояние. После выполнения всех соответствующих действий начинается новый цикл — с чтения вновь сформированных показаний датчиков.

Управление работой этой экспертной системы текущего контроля будет осуществляться с помощью методов, аналогичных описанным в главе 9. Для этого предусматриваются три отдельных модуля: `INPUT`, `TRENDS` и `WARNINGS`. В каждой итерации цикла обновляется значение факта `cycle` и фокус переходит с одного обрабатывающего модуля на другой в должном порядке. Эти действия осуществляются с помощью следующего правила:

```
(defrule MAIN::Begin-Next-Cycle
   ?f <- (cycle ?current-cycle)
   (exists (device (status on)))
   =>
   (retract ?f)
   (assert (cycle (+ ?current-cycle 1)))
   (focus INPUT TRENDS WARNINGS))
```

Правило `Begin-Next-Cycle` повторно активизирует само себя, пока еще имеются работающие устройства. Следующее правило гарантирует останов системы после того, как все устройства будут выключены:

```
(defrule MAIN::End-Cycles
   (not (device (status on)))
   =>
   (printout t "All devices are off" crlf)
   (printout t "Halting monitoring system" crlf)
   (halt))
```

Чтение бесформатных показаний датчиков

Следующим шагом в создании системы текущего контроля является обеспечение чтения показаний, поступающих от датчиков. Логически обоснован подход, в котором эти данные считываются, когда в текущем фокусе находится модуль

INPUT. Для отладки и проверки удобно также обеспечить возможность чтения показаний датчиков из некоторых других источников. В настоящем разделе будет показано, как обеспечить чтение данных непосредственно от датчиков, из экземпляров, из файла или с клавиатуры (после ввода пользователем). Для обозначения источника данных для системы текущего контроля используется факт data-source, описанный выше. Показания датчиков, считанные как "бесформатные" данные, сохраняются непосредственно в слотах raw-value фактов sensor, а обработка и анализ этих бесформатных данных осуществляются в фазе анализа.

Для того чтобы программа оставалась гибкой и позволяла в дальнейшем вводить новые источники входных данных, авторы определили все источники входных данных как экземпляры конструкции defclass с именем DATA-SOURCE, описанной следующим образом:

```
(defclass INPUT::DATA-SOURCE
   (is-a USER))
(defmessage-handler INPUT::DATA-SOURCE
get-data (?name)
   (printout t "Input value for sensor "
   ?name ": ")
   (read))
(defmessage-handler INPUT::DATA-SOURCE
next-cycle (?cycle))
(definstances INPUT::user-data-source
   ([user] of DATA-SOURCE))
```

Конструкция defclass с именем DATA-SOURCE имеет два обработчика сообщений: get-data и next-cycle. Обработчик сообщений get-data используется для выборки бесформатных значений от датчика, указанного параметром ?name. По умолчанию этот обработчик просто запрашивает требуемое значение у пользователя. Обработчик сообщений next-cycle применяется для выполнения всех необходимых операций обработки источника данных в начале нового цикла. По умолчанию этот обработчик не выполняет никаких действий. В конструкции definstances с именем user-data-source определяется экземпляр [user], который в случае его использования в качестве источника данных запрашивает у пользователя необходимые данные — показания датчиков.

В реальных условиях эксплуатации показания датчиков, скорее всего, будут поступать непосредственно от датчиков, а не от пользователя, поэтому для выполнения указанной работы требуются внешние функции. Предположим, что для возврата текущих показаний датчика используется функция get-sensor-value, вызываемая с параметром, представляющим идентификатор датчика, показания которого требуются в программе. (Для того чтобы иметь возможность вызвать

функцию, написанную на языке C, Ada, FORTRAN или каком-то другом языке программирования, необходимо перекомпилировать исходный код CLIPS.) После создания подкласса класса DATA-SOURCE и формирования ссылки на экземпляр этого подкласса появляется возможность считывать показания датчиков непосредственно с датчиков, а не получать их от пользователя:

```
(defclass INPUT::SENSOR-DATA-SOURCE
   (is-a DATA-SOURCE))
(defmessage-handler INPUT::SENSOR-DATA-SOURCE
            get-data (?name)
(get-sensor-value ?name))
(definstances INPUT::sensor-data-source
   ([sensor] of SENSOR-DATA-SOURCE))
```

Обратите внимание на то, что единственным существенным изменением стало перекрытие обработчика get-data, для того чтобы он вызывал функцию get-sensor-value.

При выполнении тестовых прогонов или в таком режиме проверки, когда желательно не предусматривать ввод пользователем большого объема данных, может оказаться более целесообразным чтение данных из "сценария", а не фактическое чтение от датчика или передача пользователю запросов на ввод каждого значения. Один из методов достижения этой цели может предусматривать сохранение данных в экземплярах, доступ к которым может быть затем получен с помощью правил. Ниже приведена конструкция defclass, которая описывает экземпляр, предназначенный для хранения значений данных.

```
(defclass INPUT::SENSOR-DATA
   (is-a USER)
   (multislot data))
```

В этом определении слот data содержит фактический список значений данных, относящихся к рассматриваемому датчику.

Создаваемые с помощью этой конструкции defclass конструкции definstances, содержащие тестовые значения, могут выглядеть следующим образом:

```
(definstances INPUT::sensor-instance-data-values
   ([S1-DATA-SOURCE] of SENSOR-DATA
      (data 100 100 110 110 115 120))
   ([S2-DATA-SOURCE] of SENSOR-DATA
      (data 110 120 125 130 130 135))
   ([S3-DATA-SOURCE] of SENSOR-DATA
      (data 100 120 125 130 130 125))
   ([S4-DATA-SOURCE] of SENSOR-DATA
      (data 120 120 120 125 130 135))
```

```
   ([S5-DATA-SOURCE] of SENSOR-DATA
      (data 110 120 125 130 135 135))
   ([S6-DATA-SOURCE] of SENSOR-DATA
      (data 115 120 125 135 130 135)))
```

Обратите внимание на то, что имя датчика, для представления которого предназначены эти данные, включено в состав имени экземпляра. Например, экземпляр `[S1-DATA-SOURCE]` содержит данные для датчика `S1`.

Для чтения показаний датчиков из экземпляра `SENSOR-DATA` используется класс `INSTANCE-DATA-SOURCE`:

```
(defclass INPUT::INSTANCE-DATA-SOURCE
 (is-a DATA-SOURCE))
(defmessage-handler INPUT::INSTANCE-DATA-SOURCE
            get-data (?name)
   ;; Найти экземпляр SENSOR-DATA
   (bind ?sensor-data
      (instance-name
         (sym-cat ?name -DATA-SOURCE)))
   (if (not (instance-existp ?sensor-data))
      then (return nil))
   ;; Проверить наличие оставшихся значений данных
   (bind ?data (send ?sensor-data get-data))
   (if (= (length$ ?data) 0)
      then
      (return nil))
   ;; Удалить первое значение из списка и~возвратить его
   (send ?sensor-data put-data (rest$ ?data))
   (nth$ 1 ?data))
(definstances INPUT::instance-data-source
   ([instance] of INSTANCE-DATA-SOURCE))
```

Обработчик сообщений `get-data` класса `INSTANCE-DATA-SOURCE` немного сложнее по сравнению с двумя описанными выше обработчиками `get-data`. Прежде всего, имя датчика в нем дополняется символом `-DATA-SOURCE` для формирования имени требуемого экземпляра `SENSOR-DATA`, а затем проверяется, существует ли этот экземпляр. После этого проверяется, есть ли еще оставшиеся значения бесформатных данных. Если оставшиеся значения еще имеются, то обработчик сообщений извлекает из списка первое из этих значений и возвращает его. Если на любом из этапов обнаруживается ошибка, обработчик возвращает символ `nil`.

Последний метод ввода, рассматриваемый в этом разделе, относится к чтению информации из файла данных. Этот метод сложнее по сравнению с описан-

ными в предыдущих примерах, поскольку требует решения определенных задач. Вначале файл данных необходимо открыть, а затем считывать данные из файла последовательно. Методы ввода, которые рассматривались перед этим, не были зависимыми от последовательного чтения данных, поскольку позволяли обращаться к новым данным, поступающих от датчиков в любом порядке. Кроме того, не было необходимости знать, какое количество показаний датчиков должно быть считано, ведь в целом желательно избежать жесткого задания в коде такой информации. К еще большему усложнению приводит то, что в случае принятия предположения о том, что значения данных, полученных от датчика, могут оставаться незаданными, необходимо также исходить из того, что используются бесформатные показания датчика, сохранившиеся от предыдущего цикла. Вариант, в котором в качестве источника данных применяется файл, будет реализован с помощью следующей конструкции `defclass`:

```
(defclass INPUT::FILE-DATA-SOURCE
   (is-a DATA-SOURCE)
   (slot file-logical-name (default FALSE))
   (multislot sensor)
   (multislot value))
```

Слот `file-logical-name` служит для хранения логического имени, связанного с открытым файлом. По мере считывания значений из файла имя датчика сохраняется в слоте `sensor`, а бесформатное значение — в соответствующей позиции в слоте `value`.

Первая проблема, которая должна быть решена, состоит в том, что первоначально файл данных должен быть открыт. Обработчик `get-file` передает пользователю запрос для ввода имени файла, а затем открывает указанный файл:

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
            get-file ()
   (bind ?logical-name (gensym*))
   (while TRUE
      (printout t
      "What is the name of the data file? ")
      (bind ?file-name (readline))
      (if (open ?file-name ?logical-name "r")
         then
         (bind ?self:file-logical-name
         ?logical-name)
         (return))))
```

Вначале с помощью вызова функции `gensym*` создается уникальное логическое имя, которое должно быть связано с открытым файлом. В обработчике можно было бы жестко закодировать конкретное имя, поскольку в рассматриваемом при-

мере в качестве источника данных должен использоваться только один файл, но вполне можно допустить, что в другом варианте программы потребуется обеспечить чтение показаний датчиков из нескольких разных файлов, поэтому данную возможность будет проще учесть, если уникальное логическое имя формируется автоматически. В обработчике предусмотрен цикл `while`, с помощью которого может повторно выводиться запрос к пользователю, чтобы он указал файл данных, который необходимо открыть. Если попытка открыть файл завершается успешно, то слоту `file-logical-name` присваивается значение сгенерированного логического имени и происходит выход из данного обработчика. В противном случае пользователь еще раз получает запрос на ввод имени файла с исходными данными, и это происходит до тех пор, пока вызов функции `open` для открытия файла с указанным именем не завершится успешно.

Еще одно уточнение касается того, каким должен быть формат файла, предназначенного для хранения входных данных датчика. Желательно предусмотреть возможность задавать только те показания датчика, которые изменились со времени выполнения последнего цикла. Поэтому неизвестно, сколько значений данных, полученных от датчиков, должно быть считано. Кроме того, невозможно определить заранее, в каком порядке будут поступать показания от разных датчиков. Эти предположения диктуют формат данных, в котором должно быть предусмотрено применение имени датчика, а также подлежащего считыванию бесформатного значения показания датчика. А поскольку количество значений данных, которые должны быть считаны, остается неизвестным, то должна быть обозначена такая ситуация, что заканчиваются значения данных, относящихся к текущему циклу; для этого используется ключевое слово `end-of-cycle`. Данные, представленные с применением описанного формата, выглядят следующим образом:

```
S1 100
S2 110
S3 100
S4 120
S5 110
S6 115
end-of-cycle
S2 120
S3 120
S5 120
S6 120
end-of-cycle
S1 110
S2 125
S3 125
```

```
S5 125
S6 125
end-of-cycle
...
```

Для сохранения в экземпляре FILE-DATA-SOURCE значений, полученных из файла, будет применяться следующий обработчик сообщений put-sensor-value:

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
            put-sensor-value (?sensor ?value)
   (bind ?position (member$ ?sensor ?self:sensor))
   (if ?position
      then
         (bind ?self:value
         (replace$ ?self:value ?position
            ?position ?value))
      else
      (bind ?self:sensor ?self:sensor ?sensor)
   (bind ?self:value ?self:value ?value)))
```

В этом обработчике сообщений прежде всего осуществляется поиск позиции датчика с указанным именем в значении слота sensor. Если имя датчика содержится в слоте sensor, то в слоте value соответствующее этому датчику значение заменяется новым бесформатным значением. В противном случае обработчик добавляет имя датчика и значение к концу списков значений, содержащихся в слотах value и sensor. При такой организации работы происходит только добавление или замена значений, поэтому если в каком-то конкретном цикле показания одного из датчиков остаются не заданными, то сохраняется предыдущее значение показаний, которое остается доступным при выполнении запроса к источнику данных.

Для закрытия файла, после того, как в нем больше не остается доступных значений показаний датчика, используется обработчик сообщений close-data-source:

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
            close-data-source ()
   (close ?self:file-logical-name)
   (bind ?self:sensor (create$))
   (bind ?self:value (create$)))
```

Для закрытия файла, связанного с логическим именем, хранящимся в слоте file-logical-name, вызывается функция close. Кроме того, уничтожается

содержимое слотов sensor и value, для того чтобы из них больше нельзя было выбирать значения показаний датчиков.

Подкласс FILE-DATA-SOURCE класса DATA-SOURCE отличается от других рассматривавшихся ранее подклассов этого класса тем, что обработчик сообщений next-cycle перекрывается в нем так, чтобы выборка всех значений показаний датчиков, относящихся к следующему циклу, осуществлялась одновременно, из одного и того же файла:

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
            next-cycle (?cycle)
   (if (not ?self:file-logical-name)
      then (send ?self get-file))
   (bind ?name (read ?self:file-logical-name))
   (if (eq ?name EOF)
      then
      (send ?self close-data-source)
      (return))
   (while (and (neq ?name end-of-cycle)
      (neq ?name EOF))
      (bind ?raw-value
         (read ?self:file-logical-name))
      (if (eq ?raw-value EOF)
         then
         (send ?self close-data-source)
         (return))
      (send ?self put-sensor-value
         ?name ?raw-value)
      (bind ?name (read ?self:file-logical-name))
      (if (eq ?name EOF)
         then
         (send ?self close-data-source)
         (return))))
```

Прежде всего, если еще не было изменено первоначальное заданное по умолчанию значение FALSE слота file-logical-name, то экземпляру передается сообщение get-file для получения имени файла и открытия файла. Затем этот обработчик считывает все значения данных из файла данных до тех пор, пока не встретится ключевое слово end-of-cycle. Если же достигается конец файла, то экземпляру передается сообщение close-data-source для закрытия файла и удаления всех существующих показаний датчиков. В противном случае считанные значения данных сохраняются в экземпляре (для последующей выборки) путем передачи сообщения put-sensor-value.

Кроме того, должен быть также перекрыт обработчик сообщений get-data:

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
            get-data (?name)
   (bind ?position (member$ ?name ?self:sensor))
   (if ?position
      then
      (nth$ ?position ?self:value)
      else
      (return nil)))
```

Обработчик сообщений get-data выполняет поиск позиции указанного датчика в слоте sensor, а затем, если информация об этом датчике существует, возвращает соответствующее значение, хранящееся в слоте value. Наконец, необходимо также добавить конструкцию definstances, содержащую экземпляр класса FILE-DATA-SOURCE:

```
(definstances INPUT::file-data-source
   ([file] of FILE-DATA-SOURCE))
```

Итак, предусмотрены четыре различных источника данных, в которых используются обработчики сообщений с одинаковыми определениями, поэтому для чтения данных из любого источника данных можно применять всего два правила и конструкцию deffacts, как показано ниже.

```
(deffacts local-cycle
   (local-cycle 0))
(defrule INPUT::next-cycle
(cycle ?cycle)
   ?f <- (local-cycle ~?cycle)
   (data-source ?source)
   (object (is-a DATA-SOURCE) (name ?source))
=>
   (send ?source next-cycle ?cycle)
   (retract ?f)
   (assert (local-cycle ?cycle)))
(defrule INPUT::Get-Sensor-Value-From-Data-Source
(cycle ?cycle)
(local-cycle ?cycle)
(data-source ?source)
(object (is-a DATA-SOURCE) (name ?source))
   ?s <- (sensor (name ?name)
                 (raw-value none)
                 (device ?device))
```

```
                (device (name ?device) (status on))
=>
   (bind ?raw-value (send ?source get-data ?name))
   (if (not (numberp ?raw-value))
      then
      (printout t "No data for sensor "
                  ?name crlf)
      (printout t "Halting monitoring system"
            crlf)
      (halt)
      else
      (modify ?s (raw-value ?raw-value))))
```

Для того чтобы указать, было ли обработано сообщение `next-cycle`, относящееся к текущему циклу работы программы контроля, используется факт `local-cycle`. С помощью правила `next-cycle` проверяется, отличается ли цикл, указанный с помощью факта `cycle`, от цикла, на который указывает факт `local-cycle`. Если действительно имеет место такая ситуация, то экземпляру `DATA-SOURCE`, с которым выполнено согласование в левой части правила, передается сообщение `next-cycle`, а факт `local-cycle` обновляется путем сохранения в нем такого же значения, как в факте `cycle`. Применительно к экземплярам `DATA-SOURCE`, `SENSOR-DATA-SOURCE` и `INSTANCE-DATA-SOURCE` обработчик сообщений `next-cycle` не выполняет никаких действий, а применительно к экземпляру `FILE-DATA-SOURCE` обработчик `next-cycle` считывает все значения данных, относящиеся к текущему циклу, в данный экземпляр.

После согласования фактов `cycle` и `local-cycle` правило `Get-Sensor-Value-From-Data-Source` используется для выборки бесформатных значений, относящихся к каждому датчику, который подключен к работающему устройству. Экземпляру `DATA-SOURCE`, согласованному в левой части правила, передается сообщение `get-data` для получения бесформатного значения, относящегося к конкретному датчику. В случае возврата нечислового значения система текущего контроля останавливается, поскольку нечисловые значения применяются для указания на то, что данных больше нет. В противном случае бесформатное значение показаний датчика сохраняется в слоте `raw-value` факта `sensor`.

Распознавание тенденций

В следующей фазе цикла работы программы, в фазе распознавания тенденций, определяется текущее состояние датчиков и проводятся вычисления, с помощью которых могут быть обнаружены складывающиеся тенденции. При этом необходимо определить текущее состояние показаний датчика (находятся в рабочем диапазоне, в нижнем или верхнем допустимом диапазоне либо за пределами ниж-

него или верхнего допустимого диапазона) на основании бесформатного значения показаний датчика, полученного в фазе ввода. Текущее состояние показаний датчика сохраняется в слоте `state` конструкции `deftemplate` с именем `sensor`. Следующее правило определяет, находится ли датчик в рабочем состоянии:

```
(defrule TRENDS::Normal-State
   ?s <- (sensor (raw-value ?raw-value&~none)
            (low-guard-line ?lgl)
            (high-guard-line ?hgl))
   (test (and (> ?raw-value ?lgl)
            (< ?raw-value ?hgl)))
   =>
   (modify ?s (state normal) (raw-value none)))
```

Для определения того, что, согласно показаниям датчиков, все устройства находятся в рабочем состоянии, используются первый шаблон и следующий за ним условный элемент `test`. Значение слота `raw-value` в шаблоне `sensor` сравнивается со значением `none`, чтобы символ не сравнивался с числовыми значениями в условном элементе `test` данного правила. Диапазон, необходимый для проверки того, соответствуют ли показания датчика рабочему состоянию устройства, задается между значениями нижнего и верхнего пределов рабочего диапазона (`low guard line` и `high guard line`). Условный элемент `test` ограничивает возможность применения этого правила теми ситуациями, в которых бесформатное значение показаний датчика относится к диапазону значений, характерных для рабочего состояния. Единственным действием в данном правиле является внесение факта, констатирующего состояние показаний датчика, в список фактов. Кроме того, слоту `raw-value` присваивается литеральное значение `none`, чтобы именно это значение было считано в следующей фазе ввода.

Для анализа оставшихся четырех возможных состояний показаний датчика требуются еще четыре правила. Каждое из этих правил аналогично правилу `Normal-State`, за исключением того, что для определения текущего состояния используются другие значения, извлеченные из факта `sensor`, и другой условный элемент `test`. Все пять правил анализа состояния могут быть записаны в виде одного правила; для этого необходимо предусмотреть в правой части правила выражение `if`, позволяющее определить соответствующее состояние датчика. Но разработка кода программы по такому принципу противоречит назначению системы, управляемой данными. Кроме того, при наличии одного чрезвычайно громоздкого правила стали бы гораздо более затруднительными дальнейшие модификации программы, касающиеся действий или условий, а также подмножества возможных состояний. Остальные четыре правила анализа состояния приведены ниже.

```
(defrule TRENDS::High-Guard-Line-State
   ?s <- (sensor (raw-value ?raw-value&~none)
            (high-guard-line ?hgl)
            (high-red-line ?hrl))
   (test (and (>= ?raw-value ?hgl)
   (< ?raw-value ?hrl)))
   =>
            (modify ?s (state high-guard-line)
   (raw-value none)))
(defrule TRENDS::High-Red-Line-State
   ?s <- (sensor (raw-value ?raw-value&~none)
            (high-red-line ?hrl))
   (test (>= ?raw-value ?hrl))
   =>
   (modify ?s (state high-red-line)
            (raw-value none)))
(defrule TRENDS::Low-Guard-Line-State
   ?s <- (sensor (raw-value ?raw-value&~none)
            (low-guard-line ?lgl)
            (low-red-line ?lrl))
   (test (and (> ?raw-value ?lrl)
            (<= ?raw-value ?lgl)))
   =>
   (modify ?s (state low-guard-line)
            (raw-value none)))
(defrule TRENDS::Low-Red-Line-State
   ?s <- (sensor (raw-value ?raw-value&~none)
            (low-red-line ?lrl))
   (test (<= ?raw-value ?lrl))
   =>
   (modify ?s (state low-red-line)
            (raw-value none)))
```

Пять описанных выше правил позволяют определить состояние показаний датчика в текущем цикле. Но одним из назначений модуля `TRENDS` является обнаружение тенденций в показаниях датчиков, поэтому необходимо сопровождать информацию о прошлом состоянии показаний датчиков. Для хранения этой информации используется следующая конструкция `deftemplate`. Эта конструкция `deftemplate` должна применяться и в модуле `TRENDS`, и в модуле `WARNINGS`, поэтому ее следует поместить в модуле `MAIN`.

```
(deftemplate MAIN::sensor-trend
   (slot name)
   (slot state (default normal))
   (slot start (default 0))
   (slot end (default 0))
   (slot shutdown-duration (default 3)))
```

Слот `name` содержит имя датчика, слот `state` соответствует наиболее актуальному состоянию показаний датчика, слот `start` обозначает первый цикл, в течение которого показания датчика находились в своем текущем состоянии, слот `end` содержит номер текущего цикла, а слот `shutdown-duration` указывает, в течение какого время показания датчика должны находиться в одном из допустимых диапазонов, для того чтобы можно было остановить устройство, относящееся к датчику.

Функционирование правил, обновляющих информацию о тенденциях, зависит от наличия факта `sensor-trend`, относящегося к каждому датчику, в списке фактов. По этой причине начальные значения продолжительности интервалов, определяющих тенденции изменения показаний датчиков, сохраняются в следующей конструкции `deffacts`:

```
(deffacts MAIN::start-trends
(sensor-trend (name S1) (shutdown-duration 3))
(sensor-trend (name S2) (shutdown-duration 5))
(sensor-trend (name S3) (shutdown-duration 4))
(sensor-trend (name S4) (shutdown-duration 4))
(sensor-trend (name S5) (shutdown-duration 4))
(sensor-trend (name S6) (shutdown-duration 2)))
```

Благодаря наличию этой информации появляется возможность определить два правила, предназначенных для отслеживания тенденций изменения показаний датчиков. В одном правиле отслеживается тенденция, не изменившаяся со времени выполнения последнего цикла, а в другом — тенденция, которая в текущем цикле стала иной по сравнению с предыдущим циклом, как показано ниже.

```
(defrule TRENDS::State-Has-Not-Changed
   (cycle ?time)
   ?trend <- (sensor-trend
(name ?sensor) (state ?state)
         (end ?end-cycle&~?time))
   (sensor (name ?sensor) (state ?state)
           (raw-value none))
   =>
   (modify ?trend (end ?time)))
(defrule TRENDS::State-Has-Changed
```

```
      (cycle ?time)
   ?trend <- (sensor-trend
(name ?sensor) (state ?state)
               (end ?end-cycle&~?time))
   (sensor (name ?sensor)
           (state ?new-state&~?state)
           (raw-value none))
   =>
   (modify ?trend (start ?time)
            (end ?time)
            (state ?new-state)))
```

В первом шаблоне обоих правил устанавливается цикл. В следующем шаблоне происходит поиск факта `sensor-trend`, относящегося к предыдущему циклу. Ограничение, налагаемое на слот `end`, гарантирует, что эти правила не перейдут в бесконечный цикл. В правиле `State-Has-Not-Changed` следующий шаблон применяется для проверки того, что состояние в предыдущем цикле совпадает с состоянием в текущем цикле. А в правиле `State-Has-Changed` ограничение `?new-state&~?state` используется для выполнения прямо противоположной проверки, позволяющей убедиться в том, что состояние в текущем цикле изменилось по сравнению с состоянием в предыдущем цикле. Проверка того, не имеет ли слот `raw-value` литеральное значение `none`, исключает такое развитие событий, что определение тенденции начнется до выявления современного состояния показаний датчика. В обоих правилах обновляется время окончания цикла. Если состояние изменилось, то должны быть также обновлены значение состояния и время начала цикла.

Формирование предупреждающих сообщений

Последней фазой цикла работы системы текущего контроля является фаза формирования предупреждающих сообщений. В течение этой фазы должны быть осуществлены действия следующих трех типов: отключены устройства, связанные с датчиками, показания которых вышли за пределы допустимых диапазонов, отключены устройства, связанные с датчиками, показания которых оставались в одном из допустимых диапазонов в течение указанного количества циклов, и сформированы предупреждающие сообщения, относящиеся к датчикам, показания которых находятся в одном из допустимых диапазонов, но относящиеся к ним устройства еще не отключены.

Для останова устройств, подключенных к датчикам, показания которых вышли за пределы допустимых диапазонов, применяется следующее правило:

```
(defrule WARNINGS::Shutdown-In-Red-Region
   (cycle ?time)
   (sensor-trend
      (name ?sensor)
      (state ?state&high-red-line | low-red-line))
   (sensor (name ?sensor) (device ?device))
   ?on <- (device (name ?device) (status on))
   =>
   (printout t "Cycle " ?time " - ")
   (printout t "Sensor " ?sensor " in "
?state crlf)
   (printout t " Shutting down device "
?device crlf)
   (modify ?on (status off)))
```

Состояние показаний датчиков проверяется с помощью факта `sensor-trend`. Если показания датчика выходят за пределы допустимого диапазона, осуществляется останов соответствующего устройства. Как только показания датчиков выходят за пределы допустимых диапазонов, немедленно осуществляется останов соответствующих устройств, поэтому нет смысла проверять, как долго показания датчика находились в этом состоянии.

Правило `Shutdown-In-Guard-Region` подобно предыдущему правилу, за исключением того, что необходимым условием останова устройства является пребывание показаний датчика в одном из допустимых диапазонов в течение некоторого периода времени (определяемого значением слота `shutdown-duration` факта `sensor-trend`). Продолжительность времени, в течение которого показания датчика относились к какому-то конкретному состоянию, можно определить, вычитая значение слота `start` из значения слота `end` факта `sensor-trend`, следующим образом:

```
(defrule WARNINGS::Shutdown-In-Guard-Region
   (cycle ?time)
   (sensor-trend
      (name ?sensor)
      (state ?state&high-guard-line | low-guard-line)
   (shutdown-duration ?length)
   (start ?start) (end ?end))
   (test (>= (+ (- ?end ?start) 1) ?length))
   (sensor (name ?sensor) (device ?device))
   ?on <- (device (name ?device) (status on))
 =>
   (printout t "Cycle " ?time " - ")
```

```
   (printout t "Sensor " ?sensor " in " ?state " ")
   (printout t "for " ?length " cycles " crlf)
   (printout t " Shutting down device " ?device crlf)
   (modify ?on (status off)))
```

Это правило отличается от правила `Shutdown-In-Red-Region` главным образом тем, что в шаблон `sensor-trend` добавлены слот `shutdown-duration` и соответствующий условный элемент `test`. Эти два шаблона позволяют определить, достаточно ли долго показания датчика находились в одном из допустимых диапазонов для того, чтобы нужно было выполнить останов устройства, связанного с этим датчиком.

Последнее правило этой системы текущего контроля вырабатывает предупреждающие сообщения, относящиеся к датчикам, показания которых находятся в одном из допустимых диапазонов, но еще не были в этом диапазоне достаточно долго, чтобы потребовался останов связанных с ними устройств:

```
(defrule WARNINGS::Sensor-In-Guard-Region
   (cycle ?time)
   (sensor-trend
      (name ?sensor)
      (state ?state&high-guard-line | low-guard-line)
      (shutdown-duration ?length)
      (start ?start) (end ?end))
   (test (< (+ (- ?end ?start) 1) ?length))
   =>
   (printout t "Cycle " ?time " - ")
   (printout t "Sensor " ?sensor " in " ?state crlf))
```

Эти правила действуют как дополнение к правилу `Shutdown-In-Guard-Region`. В нем условный элемент `test` модифицирован так, чтобы обеспечивалась проверка условия, что показания датчика находились в одном из допустимых диапазонов на протяжении количества циклов, меньшего по сравнению с количеством циклов, необходимым для останова устройства, связанного с датчиком. По условиям данного правила не требуется останавливать работу устройства, связанного с датчиком, поэтому шаблоны, позволяющие определить, какое устройство связано с датчиком, не предусмотрены.

После формулировки этого правила разработка основы очень простой системы текущего контроля заканчивается. А с помощью дополнительных правил можно было бы отражать конкретные ситуации, подлежащие текущему контролю, или ввести в действие более универсальную модель, позволяющую учитывать сложные ситуации контроля и более полно представлять связи “датчик–устройство”.

12.6 Резюме

В настоящей главе прежде всего рассматривается метод представления коэффициентов достоверности в стиле MYCIN в системе CLIPS. Для представления троек “объект–атрибут–значение” (Object–Attribute–Value — OAV) используются факты. Информация о коэффициенте достоверности, соответствующем факту, содержится в дополнительном слоте, предусмотренном в каждом факте. Для вычисления значений достоверности в правых частях правил, представленных фактами, вновь вводимыми в список фактов, по значениям достоверности, присвоенным в левых частях правил, используются отдельные правила. Затем отдельное правило используется для комбинирования двух вхождений тройки OAV в одно вхождение, с новым коэффициентом достоверности, вычисленным по коэффициентам достоверности двух исходных троек.

Применяемый в системе CLIPS подход, основанный на прямом логическом выводе, позволяет также представлять деревья решений. Разработан целый ряд алгоритмов для прохождения по деревьям решений различных типов, включая бинарные деревья решений (характеризующихся тем, что из каждого узла могут исходить самое большее два ребра) и деревья решений с несколькими исходящими ребрами в каждом узле, а также алгоритмов обучения деревьев решений с несколькими исходящими ребрами. В настоящей главе приведен пример реализации на языке CLIPS алгоритма, с помощью которого осуществляется обучение деревьев решений с несколькими исходящими ребрами.

Язык CLIPS позволяет также эмулировать стратегию обратного логического вывода. В данной главе показано, как создать с использованием правил на языке CLIPS машину обратного логического вывода. Этот пример отчасти основан на алгоритме осуществления обратного логического вывода, предназначенном для реализации на процедурном языке. Правила обратного логического вывода представляются в виде фактов и подвергаются обработке с помощью правил прямого логического вывода языка CLIPS. На основе поэтапной трассировки, проводимой в ходе сеанса обратного логического вывода, показано, как описанный замысел осуществляется в системе CLIPS.

Последним примером в данной главе является простая экспертная система текущего контроля. На этом примере показано, как приступить к созданию экспертной системы, составив исходные требования к решению задачи текущего контроля. Вначале формулируются исходные предположения, а по мере введения в систему текущего контроля все новых и новых правил уточняются дополнительные подробности. Процесс эксплуатации данной системы текущего контроля подразделяется на три фазы. Фаза ввода характеризуется тем, что в ней осуществляется сбор бесформатных значений данных от датчиков. В главе показано несколько различных методов получения значений данных. В фазе анализа тенденций анализируются бесформатные значения данных, полученные от датчиков,

и распознаются тенденции, складывающиеся в процессе эксплуатации системы. Наконец, в фазе формирования предупреждающих сообщений выдаются предупреждающие сообщения и выполняются соответствующие действия на основе результатов, полученных в фазе анализа тенденций.

Задачи

12.1. Напишите правила CLIPS, позволяющие объединить коэффициенты достоверности MYCIN для двух приведенных ниже случаев, как показано в разделе 12.2.

$$\text{New Certainty} = (CF_1 + CF_2) + (CF_1 * CF_2) \quad \text{если } CF_1 \leqslant 0 \text{ и } CF_2 \leqslant 0$$

$$\text{New Certainty} = \frac{CF_1 + CF_2}{1 - \min\{CF_1, CF_2\}} \quad \text{если } -1 < CF_1 * CF_2 < 0$$

12.2. Покажите, как можно было бы включить в систему CLIPS средства поддержки классических вероятностей с использованием методов такого типа, как показано в разделе 12.2. Перечислите возможные преимущества и недостатки использования классических вероятностей в правилах.

12.3. Реализуйте алгоритм `Solve_Tree_and_Learn`, описанный в разделе 12.3, на основе процедурного языка, такого как LISP, C или PASCAL. Проверьте подготовленную вами реализацию с использованием в качестве примера системы идентификации животных.

12.4. Реализуйте алгоритмы `Solve_Goal` и `Attempt_Rule`, описанные в разделе 12.4, на основе процедурного языка, такого как LISP, C или PASCAL. Проверьте подготовленную вами реализацию с использованием в качестве примера системы предоставления консультаций по выбору вина.

12.5. Модифицируйте обработчик сообщений `DATA-SOURCE` с именем `get-data` таким образом, чтобы он допускал ввод пользователем только числовых данных. Обработчик должен возвращать нечисловое значение только в том случае, если пользователь введет слово `halt`.

12.6. Создайте подкласс конструкции `defclass` с именем `DATA-SOURCE`, который позволял бы пользователю нажимать клавишу ввода, не вводя фактические значения данных, для указания на то, что должно быть сохранено предыдущее значение показаний каждого датчика. Например, если предыдущими значениями показаний датчиков 1 и 2 были 100 и 120, то следующие введенные данные:

```
What is the value for sensor 1? ↵
What is the value for sensor 2? 130↵
```

должны установить значение бесформатных данных датчика 1, равное `100`, и значение бесформатных данных датчика 2, равное `130`. Какие должны быть приняты предположения и какие действия должны быть выполнены, если предыдущее значение датчика неизвестно?

12.7. Модифицируйте программу так, чтобы контроль показаний датчиков продолжался даже после отключения устройства. А после того, как показания все датчиков устройства возвратятся в нормальное состояние, устройство должно быть снова включено.

12.8. Правило `Sensor-In-Guard-Region` выдает предупреждение, касающееся датчика, даже если в рабочем списке правил имеется другое правило, после чего происходит отключение устройства, связанного с датчиком. Какое изменение необходимо внести в это правило, чтобы исключить выдачу с его помощью предупреждения, если в модуле `WARNINGS` активизировано другое правило, согласно которому должно быть отключено соответствующее устройство?

12.9. Введите дополнительные правила для вывода на внешнее устройство в фазе `warnings` сообщений, указывающих на то, что показания датчика соответствовали нормальному режиму работы в течение по крайней мере *n* циклов, где значение *n* задано в некотором факте. Вывод этого сообщения должен осуществляться только через каждые *n* циклов.

12.10. Как можно модифицировать правила обратного логического вывода, описанные в разделе 12.4, чтобы обеспечить прямой логический вывод?

12.11. Внесите изменения в программу, разработанную в результате решения задачи 11.1 (см. с. 945), таким образом, чтобы факты, представляющие кустарники, загружались из файла с использованием функции `load-facts`. После предоставления пользователю объяснения должен быть выведен запрос для определения того, желает ли пользователь выбрать еще один кустарник, и в случае положительного ответа произведен повторный запуск программы.

12.12. Используя программу, разработанную в результате решения задачи 11.3, измените программу, разработанную в результате решения задачи 10.10 (см. с. 847), таким образом, чтобы в субменю “Add Gizmo” перечислялись только невыбранные приспособления, а в субменю “Remove Gizmo” — только выбранные ранее приспособления. В главном меню опция меню “Add Gizmo” должна отображаться, только если есть невыбранные приспособления, а опция меню “Remove Gizmo” — только если есть выбранные приспособления.

12.13. Используя программу, разработанную в результате решения задачи 11.3 (см. с. 946), измените программу, разработанную в результате решения зада-

чи 10.4 (см. с. 845), таким образом, чтобы в ней использовался интерфейс, управляемый с помощью меню. Пользователю необходимо предоставить отдельные опции меню для указания цвета, твердости и плотности идентифицируемого драгоценного камня. Выбор цвета должен осуществляться с использованием субменю, содержащего допустимые цвета. Должна быть предусмотрена опция меню, позволяющая просматривать список драгоценных камней, соответствующих заданным к этому времени критериям. Например, если пользователь указал лишь то, что драгоценный камень имеет черный цвет, то с помощью этой опции меню должен быть составлен список только тех драгоценных камней, которые могут иметь черный цвет. Необходимо также предусмотреть еще две опции меню. Одна из этих опций должна предоставлять возможность получить перечень текущих значений, указанных в качестве критериев, а другая — сбросить значения всех критериев в неопределенное состояние.

Приложение A

Некоторые широко применяемые эквивалентности

$$\sim \sim p \equiv p$$

$$p \to q \equiv \sim p \vee q \equiv \sim q \to \sim p$$

$$\sim(p \wedge q) \equiv \sim p \vee \sim q$$
$$\sim(p \vee q) \equiv \sim p \wedge \sim q$$

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$
$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

$$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$
$$(p \vee q) \vee r \equiv p \vee (q \vee r)$$

$$p \wedge q \equiv q \wedge p$$
$$p \vee q \equiv q \vee p$$

$$\sim(\forall x)P(x) \equiv (\exists x) \sim P(x)$$
$$\sim(\exists x)P(x) \equiv (\forall x) \sim P(x)$$

$$(\forall x)P(x) \wedge (\forall x)Q(x) \equiv (\forall x)(P(x) \wedge Q(x))$$
$$(\exists x)P(x) \vee (\exists x)Q(x) \equiv (\exists x)(P(x) \vee Q(x))$$

Примечание. Квантор $\forall$ не распределяется по связке $\vee$, а квантор $\exists$ не распределяется по связке $\wedge$, поэтому для формулировки следующих двух эквивалентностей требуется фиктивная переменная, z:

$$\begin{aligned}(\forall x)P(x) \vee (\forall x)Q(x) &\equiv (\forall x)P(x) \vee (\forall z)Q(z)\\ &\equiv (\forall x)(\forall z)(P(x) \vee Q(z))\end{aligned}$$

$$\begin{aligned}(\exists x)P(x) \wedge (\exists x)Q(x) &\equiv (\exists x)P(x) \wedge (\exists z)Q(z)\\ &\equiv (\exists x)(\exists z)(P(x) \wedge Q(z))\end{aligned}$$

Приложение Б

Некоторые элементарные кванторы и их значение

Формула	**Значение**
$(\forall x)(P(x) \to Q(x))$	Для всех x, все P есть Q
$(\forall x)(P(x) \to \sim Q(x))$	Для всех x, ни один P не есть Q
$(\exists x)(P(x) \wedge Q(x))$	Для некоторых x, x есть P и Q
$(\exists x)(P(x) \wedge \sim Q(x))$	Для некоторых x, x есть P и не Q
$(\forall x)P(x)$	Для всех x, x есть P
$(\exists x)P(x)$	Некоторые x есть P (или существует P)
$\sim(\forall x)P(x)$	Не все x есть P (или некоторые x есть P)
$(\forall x) \sim P(x)$	Все x есть не P
$(\forall x)(\exists y)P(x, y)$	Для всех x, существует y, такой что P
$(\exists x) \sim P(x)$	Некоторые x есть не P

Приложение B

Некоторые свойства множеств

Название	**Формальное представление**
Коммутативность	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Ассоциативность	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
Идемпотентность	$A \cup A = A$ $A \cap A = A$
Дистрибутивность	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
Закон исключенного третьего	$A \cup A' = U$
Закон противоречия	$A \cap A' = \varnothing$
Аксиома тождества	$A \cup \varnothing = A$ $A \cap U = A$
Поглощение	$A \cup (A \cap B) = A$ $A \cap (A \cup B) = A$
Законы де Моргана	$(A \cup B)' = A' \cap B'$ $(A \cap B)' = A' \cup B'$
Двойное отрицание	$(A')' = A$
Эквивалентность	$(A' \cup B) \cap (A \cup B') = (A' \cap B') \cup (A \cap B)$
Симметрическая разность	$(A' \cap B) \cup (A \cap B') = (A' \cup B') \cap (A \cup B)$

Приложение Г

Информация о поддержке CLIPS

К настоящей книге прилагается компакт-диск, содержащий исполняемые файлы CLIPS 6.22 для MS-DOS, Windows 2000/XP и MacOS X. Кроме того, на компакт-диске находятся полный исходный код CLIPS, трехтомное справочное руководство *CLIPS Reference Manual*, а также руководство пользователя *CLIPS User's Guide*. Том I руководства *CLIPS Reference Manual*, *The Basic Programming Guide*, содержит полное описание синтаксиса CLIPS и примеры применения языка. В томе II, *The Advanced Programming Guide*, приведены подробные сведения о настройке конфигурации системы CLIPS, добавлении новых функций к языку CLIPS, внедрении интерпретатора CLIPS в другие системы, а также дано описание других сложных средств. Том III, *The Interfaces Guide*, содержит информацию о различных интерфейсах для CLIPS, зависящих от среды. Версия CLIPS для X Windows может быть создана путем компиляции кода интерфейса X Windows и исходного кода CLIPS, приведенного на компакт-диске.

Исправления программных ошибок, обновления и другую информацию, связанную с языком CLIPS, можно найти на начальной странице CLIPS по адресу `http://www.ghg.net/clips/CLIPS.html`. Вопросы, касающиеся CLIPS, могут быть направлены по адресу электронной почты `clipsYYYY@ghg.net`, где `YYYY` — текущий год (например, 2006). Пользователи Usenet имеют возможность осуществлять поиск информации и отправлять свои вопросы, касающиеся CLIPS, в группу новостей `comp.ai.shells`.

Площадкой для обсуждения вопросов исследования, разработки и реализации экспертных систем CLIPS и сопутствующих технологий является список рассылки с поддержкой потоков, который поддерживается на узле *CLIPS Developers' Forum*: `http://www.cpbinc.com/clips`.

Пользователям CLIPS предоставляется также доступ к средствам проведения электронных конференций. После оформления подписки на эту услугу поль-

зователи могут отправлять на конференцию вопросы, замечания, ответы, отзывы и прочие сообщения в виде электронной почты. Копии этих сообщений рассылаются всем подписчикам по указанному ими адресу электронной почты. Чтобы стать подписчиком электронной конференции, отправьте по адресу `clips-request@discomsys.com` сообщение, состоящее из единственной строки, которая содержит слово "`subscribe`". Адрес, найденный в поле "`Reply:`", "`Reply to:`" или "`From:`", вводится в базу данных списка рассылки, а поле темы игнорируется. После оформления подписки пользователь получает почтовое сообщение с указаниями, как участвовать в работе конференции, начиная с этого момента.

Приложение Д

Общие сведения о командах и функциях CLIPS

Рабочий список правил

`(agenda [<module-name>])`

Создает список активизированных правил, находящихся в рабочем списке правил указанного модуля (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(clear-focus-stack)`

Возвращает все имена модулей из стека фокусов.

`(focus <module-name>+)`

Задвигает один или несколько модулей в стек фокусов. Указанные модули задвигаются в стек фокусов в порядке, обратном тому, в котором они заданы в списке параметров.

`(get-focus)`

Возвращает имя модуля, находящегося в текущем фокусе.

`(get-focus-stack)`

Возвращает все имена модулей, находящиеся в стеке фокусов, в виде многозначного значения.

`(get-salience-evaluation)`

Возвращает информацию о том, по какому принципу осуществляется в настоящее время оценка значимости.

```
(get-strategy)
```

Возвращает информацию о применяемой в настоящее время стратегии разрешения конфликтов.

```
(halt)
```

Останавливает выполнение правил.

```
(list-focus-stack)
```

Создает список всех имен модулей, находящихся в стеке фокусов.

```
(pop-focus)
```

Возвращает имя модуля, соответствующего текущему фокусу, и удаляет текущий фокус из стека фокусов.

```
(refresh-agenda [<module-name>])
```

Вызывает выполнение принудительной переоценки значимости правил, находящихся в рабочем списке правил указанного модуля (или текущего модуля, если имя модуля <module-name> не задано).

```
(run [<integer-expression>])
```

Запускает выполнение правил в текущем фокусе. Если задано целочисленное выражение <integer-expression>, то выполняется только указанное количество правил, в противном случае останов происходит после того, как рабочий список правил становится пустым.

```
(set-salience-evaluation <behavior>)
<behavior> ::= when-defined | when-activated |
               every-cycle
```

Регламентирует принцип, по которому осуществляется оценка значимости.

```
(set-strategy <strategy>)
<strategy> ::= depth | breadth | simplicity |
               complexity | lex | mea | random
```

Устанавливает текущую стратегию разрешения конфликтов.

Основные математические функции

```
(abs <numeric-expression>)
```

Возвращает абсолютное значение своего единственного параметра.

```
(div <numeric-expression> <numeric-expression>+)
```

Возвращает значение первого параметра, деленного на каждый из последующих параметров. Деление осуществляется с использованием целочисленной арифметики.

```
(float <numeric-expression>)
```

Возвращает свой единственный параметр, приведенный к типу с плавающей точкой.

```
(integer <numeric-expression>)
```

Возвращает свой единственный параметр, приведенный к целочисленному типу.

```
(max <numeric-expression> <numeric-expression>+)
```

Возвращает значение наибольшего из своих параметров.

```
(min <numeric-expression> <numeric-expression>+)
```

Возвращает значение наименьшего из своих параметров.

```
(+ <numeric-expression> <numeric-expression>+)
```

Возвращает сумму своих параметров.

```
(- <numeric-expression> <numeric-expression>+)
```

Возвращает значение первого параметра за вычетом суммы всех последующих параметров.

```
(* <numeric-expression> <numeric-expression>+)
```

Возвращает произведение своих параметров.

```
(/ <numeric-expression> <numeric-expression>+)
```

Возвращает значение первого параметра, разделенного на каждый из последующих параметров.

Преобразования

```
(deg-grad <numeric-expression>)
```

Возвращает значение своего параметра, преобразованного из единиц измерения *градусы* в единицы измерения *градиенты* (400 градиентов равны 360 градусам).

```
(deg-rad <numeric-expression>)
```

Возвращает значение своего параметра, преобразованного из единиц измерения *градусы* в единицы измерения *радианы*.

`(exp <numeric-expression>)`

Возвращает значение e, возведенное в степень, определяемую единственным параметром.

`(grad-deg <numeric-expression>)`

Возвращает значение своего параметра, преобразованного из единиц измерения *градиенты* в единицы измерения *градусы*.

`(log <numeric-expression>)`

Возвращает логарифм своего параметра по основанию e.

`(log10 <numeric-expression>)`

Возвращает логарифм своего параметра по основанию 10.

`(mod <numeric-expression> <numeric-expression>)`

Возвращает остаток от деления первого параметра на второй параметр.

`(pi)`

Возвращает значение π.

`(rad-deg <numeric-expression>)`

Возвращает значение своего параметра, преобразованного из единиц измерения *радианы* в единицы измерения *градусы*.

`(round <numeric-expression>)`

Возвращает значение своего параметра, округленного до ближайшего целого числа.

`(sqrt <numeric-expression>)`

Возвращает квадратный корень своего параметра.

`(** <numeric-expression> <numeric-expression>)`

Возвращает значение первого параметра, возведенное в степень, заданную вторым параметром.

Отладка

`(dribble-off)`

Прекращает передачу выходных данных в файл трассировки, открытый с помощью функции `dribble-on`. Возвращает символ `TRUE`, если файл трассировки был успешно закрыт; в противном случае возвращает символ `FALSE`.

```
(dribble-on <file-name>)
```

Направляет весь вывод, который обычно передается на дисплей, в файл трассировки <file-name>. Возвращает символ TRUE, если файл трассировки был успешно открыт; в противном случае возвращает символ FALSE.

```
(list-watch-items)
```

Отображает текущее состояние отслеживаемых элементов.

```
(unwatch <watch-item>)
```

Запрещает вывод на дисплей информационных сообщений при осуществлении определенных операций CLIPS.

```
(watch <watch-item>)
<watch-item> ::= activations | all | compilations |
   deffunctions | facts | focus |
   generic-functions | globals |
   instances | messages |
   message-handlers | methods |
   rules | slots | statistics
```

Разрешает вывод на дисплей информационных сообщений при осуществлении определенных операций CLIPS.

Конструкция defclass

```
(browse-classes [<class-name>])
```

Отображает информацию об отношениях наследования между указанным классом и его подклассами. Если никакой класс не указан, то по умолчанию в качестве значения параметра <class-name> принимается OBJECT.

```
(class-abstractp <class-name>)
```

Возвращает символ TRUE, если указанный класс является абстрактным; в противном случае возвращает символ FALSE.

```
(class-existp <class-name>)
```

Возвращает символ TRUE, если указанный класс определен; в противном случае возвращает символ FALSE.

```
(class-reactivep <class-name>)
```

Возвращает символ TRUE, если указанный класс является активизируемым; в противном случае возвращает символ FALSE.

```
(class-slots <class-name> [inherit])
```

Возвращает имена явно определенных слотов класса в многозначном значении. Если используется необязательное ключевое слово `inherit`, учитываются также унаследованные слоты.

```
(class-subclasses <class-name> [inherit])
```

Возвращает имена подклассов класса в многозначном значении, связанных с ним прямыми отношениями наследования. Если используется необязательное ключевое слово `inherit`, учитываются также подклассы, связанные непрямыми отношениями наследования.

```
(class-superclasses <class-name> [inherit])
```

Возвращает имена суперклассов класса в многозначном значении, связанных с ним прямыми отношениями наследования. Если используется необязательное ключевое слово `inherit`, учитываются также суперклассы, связанные непрямыми отношениями наследования.

```
(defclass-module [<class-name>])
```

Возвращает имя модуля, в котором определена указанная конструкция `defclass`.

```
(describe-classes [<class-name>])
```

Предоставляет подробное описание класса.

```
(get-class-defaults-mode)
```

Возвращает текущий режим применения значений по умолчанию, который использовался при определении классов.

```
(get-defclass-list [<module-name>])
```

Возвращает многозначное значение, содержащее список классов в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

```
(list-defclasses [<module-name>])
```

Создает список классов в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

```
(ppdefclass <class-name>)
```

Отображает текст указанных конструкций `definstance`.

```
(set-class-defaults-mode convenience | conservation)
```

Устанавливает режим применения значений по умолчанию, на основе которого определяются классы.

```
(slot-allowed-values <class-name> <slot-name>)
```

Возвращает допустимые значения для слота в многозначном значении.

```
(slot-cardinality <class-name> <slot-name>)
```

Возвращает информацию о минимальном и максимальном количестве полей, которые разрешены для мультислота в многозначном значении.

```
(slot-default-value <class-name> <slot-name>)
```

Возвращает заданное по умолчанию значение, связанное со слотом.

```
(slot-direct-accessp <class-name> <slot-name>)
```

Возвращает символ TRUE, если доступ к указанному слоту может осуществляться непосредственно; в противном случае возвращает символ FALSE.

```
(slot-existp <class-name> <slot-name> [inherit])
```

Возвращает символ TRUE, если указанный слот присутствует в классе; в противном случае возвращает символ FALSE. Если ключевое слово inherit указано, то слот может быть унаследован.

```
(slot-facets <class-name> <slot-name>)
```

Возвращает значение фасетов для указанного слота класса в многозначном значении.

```
(slot-initablep <class-name> <slot-name>)
```

Возвращает символ TRUE, если указанный слот является инициализируемым; в противном случае возвращает символ FALSE.

```
(slot-publicp <class-name> <slot-name>)
```

Возвращает символ TRUE, если указанный слот является общедоступным; в противном случае возвращает символ FALSE.

```
(slot-range <class-name> <slot-name>)
```

Возвращает минимальное и максимальное числовые значения, разрешенные для слота в многозначном значении.

```
(slot-sources <class-name> <slot-name>)
```

Возвращает имена классов, которые предоставляют фасеты для указанного слота указанного класса в многозначном значении.

```
(slot-types <class-name> <slot-name>)
```

Возвращает имена примитивных типов, разрешенных для указанного слота указанного класса в многозначном значении.

```
(slot-writeablep <class-name> <slot-name>)
```

Возвращает символ TRUE, если указанный слот предназначен для записи; в противном случае возвращает символ FALSE.

`(subclassp <class1-name> <class2-name>)`

Возвращает символ `TRUE`, если первый класс является подклассом второго класса.

`(superclassp <class1-name> <class2-name>)`

Возвращает символ `TRUE`, если первый класс является суперклассом второго класса.

`(undefclass <class-name>)`

Удаляет указанный класс.

Конструкции `deffact`

`(deffacts-module <deffacts-name>)`

Возвращает имя модуля, в котором определены указанные конструкции `deffact`.

`(get-deffacts-list [<module-name>])`

Возвращает список всех конструкций `deffact` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(list-deffacts [<module-name>])`

Создает список конструкций `deffact` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(ppdeffacts <deffacts-name>)`

Отображает текст указанной конструкции `deffact`.

`(undeffacts <deffacts-name>)`

Удаляет указанную конструкцию `deffact`.

Конструкция `deffunction`

`(deffunction-module <deffunction-name>)`

Возвращает имя модуля, в котором определена указанная конструкция `deffunction`.

`(get-deffunction-list [<module-name>])`

Возвращает многозначное значение, содержащее список конструкций `deffunction` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(list-deffunctions [<module-name>])`

Создает список конструкций `deffunction` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(ppdeffunction <deffunction-name>)`

Отображает текст указанной конструкции `deffunction`.

`(undeffunction <deffunction-name>)`

Удаляет указанную конструкцию `deffunction`.

Конструкция `defgeneric`

`(defgeneric-module <defgeneric-name>)`

Возвращает имя модуля, в котором определена указанная конструкция `defgeneric`.

`(get-defgeneric-list [<module-name>])`

Возвращает многозначное значение, содержащее список конструкций `defgeneric` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(list-defgenerics [<module-name>])`

Создает список конструкций `defgeneric` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(ppdefgeneric <defgeneric-name>)`

Отображает текст указанной конструкции `defgeneric`.

`(preview-generic <generic-function-name> <expression>*)`

Создает список всех применимых методов для конкретного вызова родовой функции в порядке уменьшения приоритета, но не выполняет эти методы.

`(type <expression>)`

Возвращает символ, который представляет собой имя типа (или класса) заданного параметра.

`(undefgeneric <defgeneric-name>)`

Удаляет указанную конструкцию `defgeneric` и все ее методы.

Конструкция `defglobal`

`(defglobal-module <defglobal-name>)`

Возвращает имя модуля, в котором определена указанная конструкция `defglobal`.

(get-defglobal-list [<module-name>])

Возвращает многозначное значение, содержащее список конструкций defglobal в указанном модуле (или в текущем модуле, если имя модуля <module-name> не задано).

(get-reset-globals)

Возвращает символ (TRUE или FALSE), который показывает, по какому принципу в настоящее время осуществляется переустановка значений глобальных переменных.

(list-defglobals [<module-name>])

Создает список конструкций defglobal в указанном модуле (или в текущем модуле, если имя модуля <module-name> не задано).

(ppdefglobal <defglobal-name>)

Отображает текст указанной конструкции defglobal.

(set-reset-globals <boolean-expression>)

Регламентирует принцип, по которому осуществляется переустановка значений глобальных переменных. Если применяется принцип, согласно которому переустановка разрешена (заданный по умолчанию и соответствующий символу TRUE), то при выполнении команды reset глобальные переменные переустанавливаются, т.е. им присваиваются первоначальные значения.

(show-defglobals [<module-name>])

Создает список конструкций defglobal и их текущих значений в указанном модуле (или в текущем модуле, если имя модуля <module-name> не задано).

(undefglobal <defglobal-name>)

Удаляет указанную конструкцию defglobal.

Конструкции definstance

(definstances-module <definstances-name>)

Возвращает имя модуля, в котором определены указанные конструкции definstance.

(get-definstances-list [<module-name>])

Возвращает многозначное значение, содержащее список конструкций definstance в указанном модуле (или в текущем модуле, если имя модуля <module-name> не задано).

```
(list-definstances [<module-name>])
```

Создает список конструкций `definstance` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

```
(ppdefinstances <definstances-name>)
```

Отображает текст указанных конструкций `definstance`.

```
(undefinstances <definstances-name>)
```

Удаляет указанные конструкции `definstance`.

Конструкция defmessage-handler

```
(call-next-handler)
```

Будучи вызванной из обработчика сообщений, вызывает следующий обработчик сообщений, который переопределен или затенен выполняющимся обработчиком сообщений.

```
(get-defmessage-handler-list <class-name> [inherit])
```

Возвращает многозначное значение, содержащее тройки из имен классов, имен обработчиков и типов обработчиков для указанного класса. Если указан параметр `inherit`, то в многозначное значение включаются также унаследованные обработчики сообщений.

```
(list-defmessage-handlers [<class-name> [inherit]])
```

Создает список конструкций `defmessage-handler` для указанного класса в текущем модуле. Если никакой класс не указан, то перечисляются обработчики сообщений для всех классов в текущем модуле. Если указан параметр `inherit`, то отображается также информация об унаследованных обработчиках сообщений.

```
(message-handler-existp <class-name><handler-name>
   [<handler-type>])
<handler-type> ::= around | before | primary | after
```

Возвращает символ `TRUE`, если указанный обработчик сообщений для указанного класса определен непосредственно (а не с помощью наследования); в противном случае возвращает символ `FALSE`.

```
(next-handlerp)
```

Возвращает символ `TRUE`, если есть другой обработчик сообщений, доступный для выполнения; в противном случае возвращает символ `FALSE`.

```
(override-next-handler <expression>*)
```

Вызывает следующий затененный обработчик и допускает внесение изменений в значения параметров.

```
(ppdefmessage-handler <class-name> <handler-name>
   [<handler-type>])
<handler-type> ::= around | before | primary | after
```

Отображает текст указанной конструкции `defmessage-handler`. Если параметр `<handler-type>` не определен, то предполагается, что он имеет значение `primary`.

```
(preview-send <class-name> <message-name>)
```

Отображает список применимых обработчиков для сообщения, передаваемого экземпляру указанного класса.

```
(undefmessage-handler <class-name> <handler-name>
   [<handler-type>])
<handler-type> ::= around | before | primary | after
```

Удаляет указанную конструкцию `defmessage-handler`. Если параметр `<handler-type>` не определен, то предполагается, что он имеет значение `primary`.

Конструкция `defmethod`

```
(call-next-method)
```

Вызывает следующий затененный метод.

```
(call-specific-method <defgeneric-name> <method-index>
   <expression>*)
```

Вызывает конкретный метод родовой функции, не учитывая приоритеты методов.

```
(get-defmethod-list [<defgeneric-name>])
```

Возвращает многозначное значение для всех методов в текущем модуле, которое содержит пары, состоящие из имен конструкций `defgeneric` и индексов методов. Если указано имя конструкции `defgeneric`, то в возвращаемое значение будут включены только методы, принадлежащие к этой конструкции `defgeneric`.

```
(Get-Method-Restrictions <defgeneric-name> <method-index>)
```

Возвращает многозначное значение, содержащее информацию об ограничениях для указанного метода.

```
(list-defmethods [<defgeneric-name>])
```

Создает список всех конструкций `defmethod` в текущем модуле в порядке приоритета. Если указано имя конструкции `defgeneric`, то перечисляются только методы, принадлежащие к этой конструкции `defgeneric`.

```
(next-methodp)
```

Если функция `next-methodp` вызывается из метода для родовой функции, то возвращает символ `TRUE` при наличии другого метода, затененного текущим. В противном случае функция возвращает символ `FALSE`.

```
(override-next-method <expression>*)
```

Вызывает следующий затененный метод, позволяя задавать новые значения параметров.

```
(ppdefmethod <defgeneric-name> <index>)
```

Отображает текст конструкции `defmethod`, связанной с указанной родовой функцией и индексом метода.

```
(preview-generic <generic-function-name> <expression>*)
```

Создает список всех применимых методов для конкретного родового вызова функции в порядке уменьшения приоритета, но не выполняет эти методы.

```
(undefmethod <defgeneric-name> <index>)
```

Удаляет конструкцию `defmethod`, связанную с указанной родовой функцией и индексом метода.

Конструкция `defmodule`

```
(get-current-module)
```

Возвращает имя текущего модуля.

```
(get-defmodule-list)
```

Возвращает список всех конструкций `defmodule`.

```
(list-defmodules)
```

Создает список всех конструкций `defmodule` в среде CLIPS.

```
(ppdefmodule <defmodule-name>)
```

Отображает текст указанной конструкции `defmodule`.

```
(set-current-module <defmodule-name>)
```

Устанавливает в качестве текущего модуля указанный модуль и возвращает имя модуля, который перед этим был текущим.

Конструкция defrule

`(defrule-module <defrule-name>)`

Возвращает имя модуля, в котором определена указанная конструкция `defrule`.

`(get-defrule-list [<module-name>])`

Возвращает список всех конструкций `defrule` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(get-incremental-reset)`

Возвращает значение, которое показывает, по какому принципу в настоящее время осуществляется инкрементная переустановка значений переменных.

`(list-defrules [<module-name>])`

Создает список конструкций `defrule` в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

`(matches <defrule-name>)`

Отображает список фактов и частичных соответствий, которые согласовываются с шаблонами указанного правила.

`(ppdefrule <defrule-name>)`

Отображает текст указанной конструкции `defrule`.

`(refresh <defrule-name>)`

Обновляет указанную конструкцию `defrule`. В рабочий список правил помещаются активизированные правила, соответствующие правилам, которые уже запущены, но все еще остаются действительными.

`(remove-break [<defrule-name>])`

Удаляет точку останова для указанного правила. Если никакое правило не указано, удаляются все точки останова.

`(set-break <defrule-name>)`

Устанавливает точку останова для указанного правила. Обнаружение точки останова приводит к тому, что выполнение правила останавливается до того, как произойдет запуск правила.

`(set-incremental-reset <boolean-expression>)`

Регламентирует принцип, согласно которому осуществляется инкрементная переустановка значений переменных.

`(show-breaks [<module-name>])`

Создает список правил с заданными точками останова в указанном модуле (или в текущем модуле, если имя модуля `<module-name>` не задано).

(undefrule <defrule-name>)

Удаляет указанную конструкцию defrule.

Конструкция deftemplate

(deftemplate-module <deftemplate-name>)

Возвращает имя модуля, в котором определена указанная конструкция deftemplate.

(get-deftemplate-list [<module-name>])

Возвращает список всех конструкций deftemplate в указанном модуле (или в текущем модуле, если имя модуля <module-name> не задано).

(list-deftemplates [<module-name>])

Создает список конструкций deftemplate в указанном модуле (или в текущем модуле, если имя модуля <module-name> не задано).

(ppdeftemplate <deftemplate-name>)

Отображает текст указанной конструкции deftemplate.

(undeftemplate <deftemplate-name>)

Удаляет указанную конструкцию deftemplate.

Среда

(apropos <lexeme>)

Отображает все символы, определенные в настоящее время в среде CLIPS, которые содержат указанную подстроку <lexeme>.

(batch <file-name>)

Обеспечивает "псевдопакетную" обработку интерактивных команд CLIPS, заменяя стандартный ввод содержимым файла <file-name>. Возвращает TRUE, если файл был выполнен успешно; в противном случае возвращает символ FALSE.

(batch* <file-name>)

Выполняет команды, заданные в файле. В отличие от команды batch, при использовании команды batch* происходит отработка всех команд в указанном файле и возврат, а не замена стандартного ввода.

(bload <file-name>)

Загружает двоичный образ из файла. Возвращает символ TRUE, если файл был загружен успешно; в противном случае возвращает символ FALSE.

`(bsave <file-name>)`

Сохраняет двоичный образ в файле.

`(clear)`

Удаляет все конструкции из среды CLIPS.

`(exit)`

Обеспечивает выход из среды CLIPS.

`(get-auto-float-dividend)`

Возвращает значение, указывающее на то, применяется ли в настоящее время автоматическое преобразование делимого в число с плавающей точкой.

`(get-dynamic-constraint-checking)`

Возвращает значение, указывающее на то, применяется ли в настоящее время динамическая проверка ограничений.

`(get-static-constraint-checking)`

Возвращает значение, указывающее на то, применяется ли в настоящее время статическая проверка ограничений.

`(load <file-name>)`

Загружает в среду CLIPS конструкции, хранимые в файле, который указан параметром `<file-name>`. Возвращает символ `TRUE`, если файл был загружен успешно; в противном случае возвращает символ `FALSE`.

`(load* <file-name>)`

Загружает в среду CLIPS конструкции, хранимые в файле, который указан параметром `<file-name>`, сопровождая процесс загрузки информационными сообщениями. Возвращает `TRUE`, если файл был загружен успешно; в противном случае возвращает символ `FALSE`.

`(options)`

Создает список значений флажков компилятора CLIPS.

`(reset)`

Переустанавливает среду CLIPS.

`(save <file-name>)`

Сохраняет все конструкции, которые определены в среде CLIPS, в файл, указанный параметром `<file-name>`.

`(set-auto-float-dividend <boolean-expression>)`

Если параметру `<boolean-expression>` присвоено значение `FALSE`, то автоматическое преобразование делимого в число с плавающей точкой запрещается,

а в противном случае разрешается. Функция возвращает старое значение, которое показывает, было ли перед этим разрешено автоматическое преобразование делимого в число с плавающей точкой (символ TRUE) или нет (символ FALSE).

```
(set-dynamic-constraint-checking <boolean-expression>)
```

Если параметр <boolean-expression> равен FALSE, динамическая проверка ограничений запрещается, в противном случае — разрешается. Функция возвращает старое значение, которое показывает, была ли перед этим разрешена динамическая проверка ограничений (символ TRUE) или нет (символ FALSE).

```
(set-static-constraint-checking <boolean-expression>)
```

Если параметр <boolean-expression> равен FALSE, статическая проверка ограничений запрещается, в противном случае — разрешается. Функция возвращает старое значение, которое показывает, было ли перед этим разрешена статическая проверка ограничений (символ TRUE) или нет (символ FALSE).

```
(system <lexeme-expression>*)
```

Соединяет свои параметры в виде строки и передает строку как команду на выполнение операционной системе.

Факты

```
(assert <RHS-pattern>)
```

Добавляет один или несколько фактов к списку фактов. Возвращает адрес последнего добавленного факта.

```
(assert-string <string-expression>)
```

Преобразовывает строку в факт и добавляет его к базе знаний. Возвращает адрес вновь добавленного факта.

```
(dependencies <fact-index-or-fact-address>)
```

Создает список всех частичных соответствий, от которых указанный факт принимает логическое обоснование.

```
(dependents <fact-index-or-fact-address>)
```

Создает список всех фактов, которые принимают логическое обоснование от указанного факта.

```
(duplicate <fact-index-or-fact-address> <RHS-slot>*)
```

Добавляет к базе знаний продублированную копию факта, заданного конструкцией deftemplate, после внесения изменений в одно или несколько значений слотов.

```
(facts [<module-name>]
   [<start-integer-expression>
   [<end-integer-expression>
   [<max-integer-expression>]]])
```

Отображает факты, находящиеся в списке фактов. Если задан параметр <module-name>, перечисляются только факты, видимые для указанного модуля, в противном случае перечисляются факты, видимые для текущего модуля. Факты с индексами фактов меньше, чем <start-integer-expression>, или больше, чем <end-integer-expression>, не перечисляются. Если задан параметр <max-integer-expression>, то не перечисляется больше фактов, чем указано его значением.

```
(fact-existp <fact-address-expression>)
```

Возвращает символ TRUE, если факт, указанный параметром <fact-address-expression> с обозначением индекса факта или адреса факта, существует; в противном случае возвращает символ FALSE.

```
(fact-index <fact-address-expression>)
```

Возвращает индекс факта, связанного с адресом факта.

```
(fact-relation <fact-address-expression>)
```

Возвращает имя конструкции deftemplate (имя отношения), связанной с указанным фактом.

```
(fact-slot-names <fact-address-expression>)
```

Возвращает имена слотов, связанных с фактом.

```
(fact-slot-value <fact-address-expression>
   <slot-name>)
```

Возвращает значение указанного слота от указанного факта.

```
(get-fact-duplication)
```

Возвращает значение, которое показывает, разрешено ли в настоящее время дублирование фактов.

```
(get-fact-list [<module-name>])
```

Возвращает список всех фактов в указанном модуле (или в текущем модуле, если имя модуля <module-name> не задано).

```
(load-facts <file-name>)
```

Добавляет в текущий модуль факты, содержащиеся в файле <file-name>. В случае успеха возвращает символ TRUE; в противном случае возвращает символ FALSE.

```
(modify <fact-index-or-fact-address> <RHS-slot>*)
```

Изменяет одно или несколько значений слотов в факте, заданном конструкцией deftemplate.

```
(retract <fact-index-or-fact-address>+)
```

Удаляет один или несколько фактов из списка фактов.

```
(save-facts <file-name>
   [visible | local <deftemplate-names>*])
```

Сохраняет указанные факты в файле <file-name>. В случае успеха возвращает символ TRUE; в противном случае — символ FALSE.

```
(set-fact-duplication <boolean-expression>)
```

Если параметр <boolean-expression> равен TRUE, то разрешается добавлять идентичные факты к списку фактов, в противном случае добавление дубликатов фактов запрещается. Функция возвращает старое значение, которое показывает, было ли разрешено добавление дубликатов (символ TRUE) или нет (символ FALSE).

Экземпляр

```
(any-instancep <instance-set-template> <query>)
<instance-set-template> ::=
                          (<instance-set-member-template>+)
<instance-set-member-template ::=
        (<single-field-variable> <class-name-expression>+)
<query> ::= <boolean-expression>
```

Возвращает символ TRUE, если обнаруживается какое-либо множество экземпляров, соответствующих указанному запросу; в противном случае возвращает символ FALSE.

```
(class <object-expression>)
```

Возвращает символ, представляющий собой имя класса указанного параметра.

```
(delayed-do-for-all-instances <instance-set-template>
   <query> <expression>*)
```

Вначале функция определяет все множества экземпляра, соответствующие указанному запросу, а затем вычисляет указанные выражения для каждого из обнаруженных множеств экземпляров.

```
(delete-instance)
```

Удаляет активизированный экземпляр во время вызова из тела обработчика сообщений.

```
(direct-slot-delete$ <mv-slot-name>
   <range-begin> <range-end>)
```

Разрешает удаление ряда полей в значении многозначного слота активизированного экземпляра из обработчика сообщений.

```
(direct-slot-insert$ <mv-slot-name> <index>
   <expression>+)
```

Разрешает вставку одного или нескольких значений в значении многозначного слота активизированного экземпляра из обработчика сообщений.

```
(direct-slot-replace$ <mv-slot-name> <range-begin>
   <range-end> <expression>+)
```

Разрешает замену ряда полей в значении многозначного слота активизированного экземпляра из обработчика сообщений.

```
(do-for-instance <instance-set-template> <query>
   <expression>*)
```

Вычисляет указанные выражения для первого множества экземпляров, соответствующего указанному запросу.

```
(do-for-all-instances <instance-set-template> <query>
   <expression>*)
```

Вычисляет указанные выражения для всех множеств экземпляров, соответствующих указанному запросу.

```
(dynamic-get <slot-name-expression>)
```

Возвращает значение указанного слота активизированного экземпляра.

```
(dynamic-put <slot-name-expression> <expression>*)
```

Устанавливает значение указанного слота активизированного экземпляра.

```
(find-instance <instance-set-template> <query>)
```

Возвращает многозначное значение, содержащее первое найденное множество экземпляров, соответствующее указанному запросу.

```
(find-all-instances <instance-set-template> <query>)
```

Возвращает многозначное значение, содержащее все найденные множества экземпляров, соответствующие указанному запросу.

```
(init-slots)
```

Реализовывает обработчик сообщений `init`, подключаемый к классу `USER`. Эту функцию ни в коем случае не следует вызывать непосредственно, если не определен такой обработчик сообщений `init`, что никогда не будет вызываться обработчик сообщений, подключаемый к классу `USER`.

```
(instance-address <instance-expression>)
```

Возвращает адрес экземпляра, указанного выражением `<instance-expression>`.

```
(instance-addressp <expression>)
```

Возвращает символ `TRUE`, если параметр `<expression>` представляет собой адрес экземпляра; в противном случае возвращает символ `FALSE`.

```
(instance-existp <instance-expression>)
```

Возвращает символ `TRUE`, если указанный экземпляр существует; в противном случае возвращает символ `FALSE`.

```
(instance-name <instance-expression>)
```

Возвращает имя экземпляра, указанного выражением `<instance-expression>`.

```
(instance-namep <expression>)
```

Возвращает символ `TRUE`, если параметр `<expression>` представляет собой имя экземпляра; в противном случае возвращает символ `FALSE`.

```
(instance-name-to-symbol <instance-name-expression>)
```

Возвращает символ, соответствующий имени экземпляра, указанного параметром `<instance-name-expression>`.

```
(instancep <expression>)
```

Возвращает символ `TRUE`, если параметр `<expression>` представляет собой имя экземпляра или адрес экземпляра; в противном случае возвращает символ `FALSE`.

```
(instances [<module-name> [<class-name> [inherit]]])
```

Возвращает перечень созданных экземпляров. Если задан необязательный параметр `<module-name>` с обозначением имени модуля, то в возвращаемый список включаются только экземпляры, содержащиеся в указанном модуле. Если для обозначения имени модуля используется звездочка (`*`), создается перечень всех экземпляров. При указании необязательного параметра `<class-name>` с именем класса перечисляются только экземпляры этого класса. Наконец, если также указано необязательное ключевое слово `inherit`, перечисляются все экземпляры, принадлежащие к подклассам.

```
(load-instances <file-name>)
```

Загружает экземпляры объектов из указанного файла.

```
(make-instance [<instance-name-expression>]
   of <class-name-expression> <slot-override>*)
<slot-override> ::= (<slot-name-expression> <expression>)
```

Создает и инициализирует экземпляр, заполняя слоты указанными значениями.

```
(ppinstance)
```

Будучи вызванной из тела обработчика сообщений, выводит информацию о слотах активизированного экземпляра.

```
(restore-instances <file-name>)
```

Загружает экземпляры объектов из указанного файла.

```
(save-instances <file-name>)
```

Сохраняет экземпляры объектов в указанном файле.

```
(send <object-expression> <message-name-expression>
   <expression>*)
```

Передает в указанный объект сообщение с указанными параметрами.

```
(slot-delete$ <instance-expression> <mv-slot-name>
   <range-begin> <range-end>)
```

Разрешает удаление ряда полей в значении многозначного слота.

```
(slot-insert$ <instance-expression> <mv-slot-name>
   <index> <expression>+)
```

Разрешает вставку одного или нескольких значений в значении многозначного слота.

```
(slot-replace$ <instance-expression> <mv-slot-name>
   <range-begin> <range-end>
   <expression>+)
```

Разрешает замену ряда полей в значении многозначного слота.

```
(symbol-to-instance-name <symbol-expression>)
```

Преобразовывает символ в имя экземпляра.

```
(unmake-instance <instance-expression> | *)
```

Удаляет указанный экземпляр (или все экземпляры, если указан символ `*`), передавая ему сообщение `delete`.

Ввод-вывод

`(close [<logical-name>])`

Закрывает файл, связанный с логическим именем `<logical-name>` (или все файлы, если этот параметр не задан). Возвращает символ `TRUE`, если файл был успешно закрыт; в противном случае возвращает символ `FALSE`.

`(format <logical-name> <string-expression> <expression>*)`

Вычисляет и выводит в виде отформатированного вывода в файл с логическим именем `<logical-name>` от нуля и больше выражений, отформатированных с использованием строкового выражения `<string-expression>`. Подробные сведения о флажках форматирования приведены в разделе 8.12.

`(open <file-name> <logical-name> [<mode>])`

Открывает файл `<file-name>` в указанном режиме (“`r`”, “`w`”, “`r+`” или “`a`”) и связывает с файлом логическое имя `<logical-name>`. Возвращает символ `TRUE`, если файл был успешно открыт; в противном случае возвращает символ `FALSE`.

`(printout <logical-name> <expression>*)`

Вычисляет и выводит в виде неотформатированного вывода в файл с логическим именем `<logical-name>` от нуля и больше выражений.

`(read [<logical-name>])`

Читает единственное поле из файла с указанным логическим именем `<logical-name>` (если логическое имя не задано, по умолчанию применяется `stdin`). В случае успеха возвращает поле, а если доступные входные данные отсутствуют, возвращает `EOF`.

`(readline [<logical-name>])`

Читает всю строку из файла с указанным логическим именем `<logical-name>` (если логическое имя не задано, по умолчанию применяется `stdin`). В случае успеха возвращает строку, а если доступные входные данные отсутствуют, возвращает `EOF`.

`(remove <file-name>)`

Удаляет файл `<file-name>`.

`(rename <old-file-name> <new-file-name>)`

Переименовывает файл `<old-file-name>` в `<new-file-name>`.

Память

`(conserve-mem on | off)`

Разрешает (`off`) или запрещает (`on`) хранить в памяти информацию, используемую для команд сохранения и структурированного вывода.

`(mem-used)`

Возвращает информацию о количестве байтов памяти, затребованной системой CLIPS от операционной системы.

`(mem-requests)`

Возвращает информацию о том, сколько раз система CLIPS запрашивала память от операционной системы.

`(release-mem)`

Освобождает всю свободную память, захваченную внутри системы CLIPS, передавая ее операционной системе. Возвращаемое значение показывает, какой объем памяти был освобожден.

Прочие функции

`(funcall <function-name-expression> <expression>*)`

Создает вызов функции из своих параметров, а затем выполняет этот вызов функции.

`(gensym)`

Возвращает следующий по порядку символ в форме `genX`, где *X* — целое число.

`(gensym*)`

Возвращает упорядоченный символ `genX` формы, где *X* — целое число. В отличие от функции `gensym`, функция `gensym*` производит уникальный символ, который в настоящее время больше не существует в пределах всей среды CLIPS.

`(get-function-restrictions <function-name-expression>)`

Возвращает строку ограничения, связанную с функцией CLIPS или с функцией, определяемой пользователем.

`(length <lexeme-or-multifield-expression>)`

Возвращает целочисленное значение, которое показывает количество полей в многозначном значении либо длину строки или символа.

```
(random [<start-integer-expression>
   <end-integer-expression>])
```

Возвращает "псевдослучайное" целочисленное значение (при желании могут быть указаны начальное и конечное целочисленные значения).

```
(seed <integer-expression>)
```

Устанавливает начальное число, используемое генератором случайных чисел для функции `random`.

```
(setgen <integer-expression>)
```

Устанавливает начальное значение индекса последовательности, используемого функциями `gensym` и `gensym*`.

```
(sort <comparison-function-name> <expression>*)
```

Сортирует список полей, указанных параметром `<expression>*`; для определения того, следует ли поменять местами два рассматриваемых поля, используется функция сравнения `<comparison-function-name>`.

```
(time)
```

Возвращает значение с плавающей запятой, представляющее количество секунд, истекших с начала отсчета времени в системе.

```
(timer <expression>*)
```

Возвращает информацию о количестве секунд, затраченных на вычисление указанного ряда выражений.

Функции для работы с многозначными величинами

```
(create$ <expression>*)
```

Соединяет друг с другом от нуля или больше выражений для создания многозначного значения.

```
(delete$ <multifield-expression>
   <begin-integer-expression>
   <end-integer-expression>)
```

Удаляет все поля в указанном ряду (начинающегося со значения `<begin-integer-expression>` и заканчивающегося значением `<end-integer-expression>`) из многозначного значения `<multifield-expression>` и возвращает полученный результат.

```
(delete-member$ <multifield-expression> <expression>+)
```

Удаляет указанные значения, входящие в состав многозначного значения, и возвращает модифицированное многозначное значение.

```
(explode$ <string-expression>)
```

Возвращает многозначное значение, созданное из полей, содержащихся в строке.

```
(first$ <multifield-expression>)
```

Возвращает первое поле многозначного значения <multifield-expression>.

```
(implode$ <multifield-expression>)
```

Возвращает строку, содержащую поля из многозначного значения.

```
(insert$ <multifield-expression> <integer-expression>
   <single-or-multifield-expression>+)
```

Вставляет все содержимое выражения <single-or-multifield-expression> в многозначное значение <multifield-expression> перед тем полем указанного многозначного значения, которое задано параметром <integer-expression>.

```
(length$ <multifield-expression>)
```

Возвращает информацию о количестве полей в многозначном значении.

```
(member$ <single-field-expression>
   <multifield-expression>)
```

Возвращает данные о позиции первого параметра (однозначного значения) во втором параметре (многозначном значении) или значение FALSE, если первый параметр не содержится во втором параметре.

```
(nth$ <integer-expression> <multifield-expression>)
```

Возвращает поле многозначного значения <multifield-expression>, номер которого задан параметром <integer-expression>.

```
(replace$ <multifield-expression>
   <begin-integer-expression> <end-integer-expression>
   <single-or-multifield-expression>+)
```

Заменяет поля из указанного ряда (от <begin-integer-expression> до <end-integer-expression>) в многозначном значении <multifield-expression> всеми значениями <single-or-multifield-expression> и возвращает полученный результат.

```
(replace-member$ <multifield-expression>
   <substitute-expression> <search-expression>+)
```

Заменяет конкретные поля, содержащиеся в многозначном значении, и возвращает модифицированное многозначное значение.

```
(rest$ <multifield-expression>)
```

Возвращает многозначное значение, содержащее все поля многозначного значения <multifield-expression>, кроме первого.

```
(subseq$ <multifield-expression>
   <begin-integer-expression> <end-integer-expression>)
```

Извлекает поля из указанного ряда (от <begin-integer-expression> до <end-integer-expression>) в многозначном значении <multifield-expression> и возвращает их в виде многозначного значения.

```
(subsetp <expression>)
```

Возвращает символ TRUE, если первый параметр является подмножеством второго параметра; в противном случае возвращает символ FALSE.

Предикат

```
(and <expression>+)
```

Возвращает символ TRUE, если вычисление каждого выражения <expression>, входящего в состав параметров, приводит к получению значения TRUE; в противном случае возвращает символ FALSE.

```
(eq <expression> <expression>+)
```

Возвращает символ TRUE, если первый параметр имеет такой же тип и такое же значение, как и каждый из последующих параметров; в противном случае возвращает символ FALSE.

```
(evenp <expression>)
```

Возвращает символ TRUE, если параметр <expression> представляет собой четное целое число; в противном случае возвращает символ FALSE.

```
(floatp <expression>)
```

Возвращает символ TRUE, если параметр <expression> представляет собой число с плавающей точкой; в противном случае возвращает символ FALSE.

```
(integerp <expression>)
```

Возвращает символ TRUE, если параметр <expression> представляет собой целое число; в противном случае возвращает символ FALSE.

(lexemep <expression>)

Возвращает символ TRUE, если параметр <expression> представляет собой строку или символ; в противном случае возвращает символ FALSE.

(multifieldp <expression>)

Возвращает символ TRUE, если параметр <expression> представляет собой многозначное значение; в противном случае возвращает символ FALSE.

(neq <expression> <expression>+)

Возвращает символ TRUE, если первый параметр не имеет такого же типа и такого же значения, как и все без исключения последующие параметры; в противном случае возвращает символ FALSE.

(not <expression>)

Возвращает символ TRUE, если вычисление выражения <expression>, заданного в качестве параметра, приводит к получению символа FALSE; в противном случае возвращает символ FALSE.

(numberp <expression>)

Возвращает символ TRUE, если параметр <expression> представляет собой число с плавающей точкой или целое число; в противном случае возвращает символ FALSE.

(oddp <expression>)

Возвращает символ TRUE, если параметр <expression> представляет собой нечетное целое число; в противном случае возвращает символ FALSE.

(or <expression>+)

Возвращает символ TRUE, если вычисление любого из выражений <expression>, приведенных в виде параметров, приводит к получению значения TRUE; в противном случае возвращает символ FALSE.

(pointerp <expression>)

Возвращает символ TRUE, если параметр <expression> представляет собой внешний адрес; в противном случае возвращает символ FALSE.

(stringp <expression>)

Возвращает символ TRUE, если параметр <expression> представляет собой строку; в противном случае возвращает символ FALSE.

(symbolp <expression>)

Возвращает символ TRUE, если параметр <expression> представляет собой символ; в противном случае возвращает символ FALSE.

```
(= <numeric-expression> <numeric-expression>+)
```

Возвращает символ TRUE, если первый параметр имеет такое же числовое значение, как и все последующие параметры; в противном случае возвращает символ FALSE.

```
(<> <numeric-expression> <numeric-expression>+)
```

Возвращает символ TRUE, если первый параметр не имеет такого же числового значения, как и все без исключения последующие параметры; в противном случае возвращает символ FALSE.

```
(> <numeric-expression> <numeric-expression>+)
```

Возвращает символ TRUE, если для всех параметров параметр n – 1 больше параметра n; в противном случае возвращает символ FALSE.

```
(>= <numeric-expression> <numeric-expression>+)
```

Возвращает символ TRUE, если для всех параметров параметр n – 1 больше или равен параметру n; в противном случае возвращает символ FALSE.

```
(< <numeric-expression> <numeric-expression>+)
```

Возвращает символ TRUE, если для всех параметров параметр n – 1 меньше параметра n; в противном случае возвращает символ FALSE.

```
(<= <numeric-expression> <numeric-expression>+)
```

Возвращает символ TRUE, если для всех параметров параметр n – 1 меньше или равен параметру n; в противном случае возвращает символ FALSE.

Процедурные программные средства

```
(bind <variable> <value>)
```

Связывает переменную с указанным значением.

```
(break)
```

Эта команда завершает выполнение той конструкции while, loop-for-count, progn$, do-for-instance, do-for-all-instances или той функции delayed-do-for-all-instances, в которой она непосредственно содержится.

```
(if <predicate-expression> then <expression>+
   [else <expression>+])
```

Вычисляет выражения, содержащиеся в части else функции, если вычисление выражения <predicate-expression> приводит к получению значения FALSE; в противном случае вычисляет выражения, содержащиеся в части then функции.

```
(loop-for-count <range-spec> [do] <expression>*)
<range-spec> ::= <end-index> |
   (<loop-variable> <end-index>) |
   (<loop-variable> <start-index> <end-index>)
```

Вычисляет выражения <expression>* такое количество раз, какое указано параметром <range-spec>. Если параметр <start-index> не задан, то предполагается, что он равен единице. Если значение <start-index> больше, чем <end-index>, то тело цикла не выполняется. Целочисленное значение, соответствующее текущей итерации, можно определить с помощью переменной <loop-variable>, если она задана.

```
(progn <expression>*)
```

Вычисляет все параметры <expression> и возвращает значение последнего вычисленного параметра.

```
(progn$ <list-spec> <expression>*)
<list-spec> ::= <multifield-expression> |
   (<list-variable> <multifield-expression>)
```

Вычисляет выражение <expression>* для каждого поля, содержащегося в спецификации списка <list-spec>. Значение поля в текущей итерации может быть проверено с помощью переменной <list-variable>, если она задана. Кроме того, целочисленное значение индекса текущей итерации может быть проверено с помощью переменной <list-variable>-index.

```
(return [<expression>])
```

Завершает выполнение конструкции deffunction, родового метода функции, обработчика сообщений или правой части конструкции defrule, выполняющихся в настоящее время. Если параметр, заданный выражением <expression>, отсутствует, то отсутствует и возвращаемое значение. А если параметр <expression> задан, то результат вычисления этого выражения рассматривается как возвращаемое значение конструкции deffunction, метода или обработчика сообщений.

```
(switch <test-expression> <case-statement>*
   [<default-statement>])
<case-statement> ::=
   (case <comparison-expression> then <expression>*)
<default-statement> ::= (default <expression>*)
```

Результат вычисления выражения <test-expression> сравнивается с результатом вычисления выражения <comparison-expression> в каждой конструкции case. Если обнаруживается конструкция case с совпадающим значением, вычисляется выражение <expression>* в найденной конструкции case и вы-

полняется возврат из функции. Если совпадение не обнаруживается и задана конструкция <default-statement>, то выполняется выражение <expression>*, соответствующее этой ветви, применяемой по умолчанию.

```
(while <predicate-expression> [do] <expression>*)
```

Проверяется результат вычисления выражения <predicate-expression>. Если этот результат отличен от значения FALSE, вычисляется выражение <expression>* и снова происходит переход на этап проверки, т.е. цикл повторяется.

Профилирование

```
(get-profile-percent-threshold)
```

Возвращает текущее значение максимально допустимой продолжительности профилирования в процентах.

```
(profile constructs | user-functions | off)
```

Разрешает или запрещает профилирование конструкций и определяемых пользователем функций.

```
(profile-info)
```

Отображает собранную до настоящего времени информацию профилирования конструкций и определяемых пользователем функций.

```
(profile-reset)
```

Сбрасывает (переводит в исходное состояние) всю собранную до настоящего времени информацию профилирования конструкций и определяемых пользователем функций.

```
(set-profile-percent-threshold
   <number-in-range-of-0-to-100>)
```

Устанавливает минимальную процентную долю времени, которая должна быть затрачена на выполнение конструкции или определяемой пользователем функции для того, чтобы собранную о ней информацию профилирования можно было вывести на внешнее устройство с помощью команды profile-info.

Развертывание последовательности

```
(expand$ <multifield-expression>*)
```

При использовании в вызове какой-либо функции функция expand$ развертывает свои параметры в виде отдельных параметров функции. Операция $ — это сокращенное обозначение вызова функции expand$.

```
(get-sequence-operator-recognition)
```

Возвращает символ TRUE или FALSE, которое показывает, применяется распознавание операции определения последовательности в вызове функции или нет. В первом случае обнаруженное многозначное значение развертывается и передается в функцию в виде отдельных параметров, а во втором передается в виде одного параметра.

```
(set-sequence-operator-recognition
   <boolean-expression>)
```

Устанавливает принцип распознавания операции определения последовательности (см. описание функции get-sequence-operator-recognition).

Строки

```
(build <string-or-symbol-expression>)
```

Вычисляет выражение, заданное строкой, как если бы это выражение было введено в приглашении к вводу команд. Допускается вычисление только выражений, представляющих собой конструкции.

```
(check-syntax <string-expression>)
```

Позволяет выполнить проверку текстового представления конструкции или вызова функции на наличие синтаксических и семантических ошибок. Если в результате проверки не вырабатываются какие-либо сообщения об ошибках или предупреждающие сообщения, возвращает символ FALSE.

```
(eval <string-or-symbol-expression>)
```

Вычисляет выражение, заданное строкой, как если бы это выражение было введено в приглашении к вводу команд. Допускается вычисление только выражений, представляющих собой вызовы функций.

```
(lowcase <string-or-symbol-expression>)
```

Возвращает свой параметр, в котором все прописные буквы заменены строчными буквами.

```
(str-cat <expression>*)
```

Возвращает результат конкатенации всех своих параметров в одну строку.

```
(str-compare <string-or-symbol-expression>
   <string-or-symbol-expression>)
```

Возвращает нуль, если оба параметра равны, положительное целое число, если первый параметр лексикографически больше второго, и отрицательное целое число, если первый параметр лексикографически меньше второго.

```
(str-index <lexeme-expression> <lexeme-expression>)
```

Возвращает целочисленное обозначение позиции строки, заданной первым параметром, в строке, заданной вторым параметром, если первая строка является подстрокой второй строки; в противном случае возвращает символ FALSE.

```
(str-length <string-or-symbol-expression>)
```

Возвращает длину строки в символах.

```
(string-to-field <string-or-symbol-expression>)
```

Преобразовывает строку в поле.

```
(sub-string <begin-integer-expression>
   <end-integer-expression> <string-expression>)
```

Возвращает подстроку строки, заданной выражением <string-expression>. В подстроку входит указанный участок строки (от позиции <begin-integer-expression> до позиции <end-integer-expression>).

```
(sym-cat <expression>*)
```

Возвращает результат конкатенации всех своих параметров в один символ.

```
(upcase <string-or-symbol-expression>)
```

Возвращает свой параметр, в котором все строчные буквы заменены прописными буквами.

Обработка текста

```
(fetch <file-name>)
```

Загружает указанный файл справки во внутреннюю поисковую таблицу.

```
(print-region <logical-name> <lookup-file> <topic-field>*)
```

Отыскивает указанный элемент в конкретном файле, который был предварительно загружен в поисковую таблицу, и выводит содержимое этого элемента под указанным логическим именем.

```
(toss <file-name>)
```

Выгружает указанный файл из внутренней поисковой таблицы.

Тригонометрические функции

```
(acos <numeric-expression>)
```

Возвращает арккосинус своего параметра (в радианах).

```
(acosh <numeric-expression>)
```

Возвращает гиперболический арккосинус своего параметра (в радианах).

```
(acot <numeric-expression>)
```

Возвращает арккотангенс своего параметра (в радианах).

```
(acoth <numeric-expression>)
```

Возвращает гиперболический арккотангенс своего параметра (в радианах).

```
(acsc <numeric-expression>)
```

Возвращает арккосеканс своего параметра (в радианах).

```
(acsch <numeric-expression>)
```

Возвращает гиперболический арккосеканс своего параметра (в радианах).

```
(asec <numeric-expression>)
```

Возвращает арксеканс своего параметра (в радианах).

```
(asech <numeric-expression>)
```

Возвращает гиперболический арксеканс своего параметра (в радианах).

```
(asin <numeric-expression>)
```

Возвращает арксинус своего параметра (в радианах).

```
(asinh <numeric-expression>)
```

Возвращает гиперболический арксинус своего параметра (в радианах).

```
(atan <numeric-expression>)
```

Возвращает арктангенс своего параметра (в радианах).

```
(atanh <numeric-expression>)
```

Возвращает гиперболический арктангенс своего параметра (в радианах).

```
(cos <numeric-expression>)
```

Возвращает косинус своего параметра (в радианах).

```
(cosh <numeric-expression>)
```

Возвращает гиперболический косинус своего параметра (в радианах).

```
(cot <numeric-expression>)
```

Возвращает котангенс своего параметра (в радианах).

```
(coth <numeric-expression>)
```

Возвращает гиперболический котангенс своего параметра (в радианах).

`(csc <numeric-expression>)`

Возвращает косеканс своего параметра (в радианах).

`(csch <numeric-expression>)`

Возвращает гиперболический косеканс своего параметра (в радианах).

`(sec <numeric-expression>)`

Возвращает секанс своего параметра (в радианах).

`(sech <numeric-expression>)`

Возвращает гиперболический секанс своего параметра (в радианах).

`(sin <numeric-expression>)`

Возвращает синус своего параметра (в радианах).

`(sinh <numeric-expression>)`

Возвращает гиперболический синус своего параметра (в радианах).

`(tan <numeric-expression>)`

Возвращает тангенс своего параметра (в радианах).

`(tanh <numeric-expression>)`

Возвращает гиперболический тангенс своего параметра (в радианах).

Приложение E

Определение языка в нормальной форме Бэкуса–Наура

Программа CLIPS

```
<CLIPS-program> ::= <construct>*
<construct>     ::= <deffacts-construct> |
                    <deftemplate-construct> |
                    <defglobal-construct> |
                    <defrule-construct> |
                    <deffunction-construct> |
                    <defgeneric-construct> |
                    <defmethod-construct> |
                    <defclass-construct> |
                    <definstance-construct> |
                    <defmessage-handler-construct> |
                    <defmodule-construct>
```

Конструкция `deffacts`

```
<deffacts-construct> ::= (deffacts <name> [<comment>]
   <RHS-pattern>*)
```

Конструкция `deftemplate`

```
<deftemplate-construct> ::= (deftemplate <name>
                             [<comment>]
                             <slot-definition>*)
```

```
<slot-definition> ::= <single-slot-definition> |
                      <multislot-definition>
<single-slot-definition> ::= (slot <name>
                                <slot-attribute>*)
<multislot-definition> ::= (multislot <name>
                             <template-attribute>*)
<template-attribute> ::= <default-attribute> |
                         <constraint-attribute>
<default-attribute> ::= (default ?DERIVE | ?NONE |
                         <expression>*) |
   (default-dynamic <expression>*)
```

Спецификация факта

```
<RHS-pattern> ::= <ordered-RHS-pattern> |
                  <template-RHS-pattern>
<ordered-RHS-pattern> ::= (<symbol> <RHS-field>+)
<template-RHS-pattern> ::= (<deftemplate-name>
                            <RHS-slot>*)
<RHS-slot> ::= <single-field-RHS-slot> |
               <multifield-RHS-slot>
<single-field-RHS-slot> ::= (<slot-name> <RHS-field>)
<multifield-RHS-slot> ::= (<slot-name> <RHS-field>*)
<RHS-field> ::= <variable> |
                <constant> |
                <function-call>
```

Конструкция `defrule`

```
<defrule-construct> ::= (defrule <name> [<comment>]
                          [<declaration>]
                          <conditional-element>*
                          =>
                          <expression>*)
<declaration> ::= (declare <rule-property>+)
<rule-property> ::= (salience <integer-expression>) |
                    (auto-focus <boolean-symbol>)
<boolean-symbol> ::= TRUE | FALSE
<conditional-element> ::= <pattern-CE> |
                          <assigned-pattern-CE> |
                          <not-CE> |
                          <and-CE> |
```

```
                          <or-CE> |
                          <logical-CE> |
                          <test-CE> |
                          <exists-CE> |
                          <forall-CE>
<pattern-CE> ::= <ordered-pattern-CE> |
                 <template-pattern-CE> |
                 <object-pattern-CE)
<assigned-pattern-CE>
         ::= <single-field-variable> <- <pattern-CE>
<not-CE> ::= (not <conditional-element>)
<and-CE> ::= (and <conditional-element>+)
<or-CE> ::= (or <conditional-element>+)
<logical-CE> ::= (logical <conditional-element>+)
<test-CE> ::= (test <function-call>)
<exists-CE> ::= (exists <conditional-element>+)
<forall-CE> ::= (forall <conditional-element>
                        <conditional-element>+)
<ordered-pattern-CE> ::= (<symbol> <constraint>+)
<template-pattern-CE> ::= (<deftemplate-name <LHS-slot>*)
<object-pattern-CE> ::= (object <attribute-constraint>*)
<attribute-constraint> ::= (is-a <constraint>) |
                           (name <constraint>) |
                           (<slot-name> <constraint>*)
<LHS-slot> ::= <single-field-LHS-slot> |
               <multifield-LHS-slot>
<single-field-LHS-slot> ::= (<slot-name> <constraint>)
<multifield-LHS-slot> ::= (<slot-name> <constraint>*)
<constraint> ::= ? | $? | <connected-constraint>
<connected-constraint>
   ::= <single-constraint> |
       <single-constraint> & <connected-constraint> |
       <single-constraint> | <connected-constraint>
<single-constraint> ::= <term> | ~<term>
<term> ::= <constant> |
           <single-field-variable> |
           <multifield-variable> |
           :<function-call> |
           =<function-call>
```

Конструкция `defglobal`

```
<defglobal-construct>
   ::= (defglobal [<defmodule-name>]
                  <global-assignment>*)
<global-assignment>
   ::= <global-variable> = <expression>
<global-variable> ::= ?*<symbol>*
```

Конструкция `deffunction`

```
<deffunction-construct>
   ::= (deffunction <name> [<comment>]
       (<regular-parameter>* [<wildcard-parameter>])
        <expression>*)
<regular-parameter> ::= <single-field-variable>
<wildcard-parameter> ::= <multifield-variable>
```

Конструкция `defgeneric`

```
<defgeneric-construct> ::= (defgeneric <name> [<comment>])
```

Конструкция `defmethod`

```
<defmethod-construct>
   ::= (defmethod <name> [<index>] [<comment>]
       (<parameter-restriction>*
        [<wildcard-parameter-restriction>])
       <expression>*)
<parameter-restriction>
   ::= <single-field-variable> |
       (<single-field-variable> <type>* [<query>])
<wildcard-parameter-restriction>
   ::= <multifield-variable> |
       (<multifield-variable> <type>* [<query>])
<type> ::= <class-name>
<query> ::= <global-variable> | <function-call>
```

Конструкция `defclass`

```
<defclass-construct> ::= (defclass <name> [<comment>]
                         (is-a <superclass-name>+)
                         [<role>]
```

```
                          [<pattern-match-role>]
                          <slot>*
                          <handler-documentation>*)
<role> ::= (role concrete | abstract)
<pattern-match-role>
   ::= (pattern-match reactive | non-reactive)
<slot> ::= (slot <name> <facet>*) |
           (single-slot <name> <facet>*) |
           (multislot <name> <facet>*)
<facet> ::= <default-facet> | <storage-facet> |
            <access-facet> | <propagation-facet> |
            <source-facet> | <pattern-match-facet> |
            <visibility-facet> |
            <create-accessor-facet> |
            <override-message-facet> |
            <constraint-attribute>
<default-facet>
   ::= (default ?DERIVE | ?NONE | <expression>*) |
       (default-dynamic <expression>*)
<storage-facet> ::= (storage local | shared)
<access-facet> ::= (access read-write |
                    read-only |
                    initialize-only)
<propagation-facet>
   ::= (propagation inherit | no-inherit)
<source-facet> ::= (source exclusive | composite)
<pattern-match-facet>
   ::= (pattern-match reactive | non-reactive)
<visibility-facet> ::= (visibility private | public)
<create-accessor-facet>
   ::= (create-accessor ?NONE | read |
        write | read-write)
<override-message-facet>
   ::= (override-message ?DEFAULT | <message-name>)
<handler-documentation>
   ::= (message-handler <name> [<handler-type>])
<handler-type> ::= primary | around | before | after
```

Конструкция defmessage-handler

```
<defmessage-handler-construct>
   ::= (defmessage-handler <class-name>
       <message-name> [<handler-type>] [<comment>]
       (<parameter>* [<wildcard-parameter>])
       <action>*)
<handler-type> ::= around | before | primary | after
<parameter> ::= <single-field-variable>
<wildcard-parameter> ::= <multifield-variable>
```

Конструкция definstance

```
<definstances-construct> ::= (definstances <name>
                              [active] [<comment>]
                              <instance-template>*)
<instance-template> ::= (<instance-definition>)
<instance-definition>
   ::= <instance-name-expression> of
       <class-name-expression> <slot-override>*
<slot-override>
   ::= (<slot-name-expression> <expression>*)
```

Конструкция defmodule

```
<defmodule-construct>
   ::= (defmodule <name> [<comment>])
       <port-specification>*)
<port-specification>
   ::= (export <port-item>) |
       (import <module-name> <port-item>)
<port-item> ::= ?ALL |
                ?NONE |
                <port-construct> ?ALL |
                <port-construct> ?NONE |
                <port-construct> <construct-name>+
<port-construct> ::= deftemplate | defclass |
                     defglobal | deffunction |
                     defgeneric
```

Атрибуты ограничения

```
<constraint-attribute>
   ::= <type-attribute> |
       <allowed-constant-attribute> |
       <range-attribute> |
       <cardinality-attribute>
<type-attribute> ::= (type <type-specification>)
<type-specification> ::= <allowed-type>+ | ?VARIABLE
<allowed-type> ::= SYMBOL | STRING | LEXEME |
                   INTEGER | FLOAT | NUMBER |
                   INSTANCE-NAME |
                   INSTANCE-ADDRESS |
                   INSTANCE | EXTERNAL-ADDRESS |
                   FACT-ADDRESS
<allowed-constant-attribute>
   ::= (allowed-symbols <symbol-list>) |
       (allowed-strings <string-list>) |
       (allowed-lexemes <lexeme-list> |
       (allowed-integers <integer-list>) |
       (allowed-floats <float-list>) |
       (allowed-numbers <number-list>) |
       (allowed-instance-names <instance-list>) |
       (allowed-values <value-list>)
<symbol-list> ::= <symbol>+ | ?VARIABLE
<string-list> ::= <string>+ | ?VARIABLE
<lexeme-list> ::= <lexeme>+ | ?VARIABLE
<integer-list> ::= <integer>+ | ?VARIABLE
<float-list> ::= <float>+ | ?VARIABLE
<number-list> ::= <number>+ | ?VARIABLE
<instance-name-list> ::= <instance-name>+ | ?VARIABLE
<value-list> ::= <constant>+ | ?VARIABLE
<range-attribute> ::= (range <range-specification>
                      <range-specification>)
<range-specification> ::= <number> | ?VARIABLE
<cardinality-attribute>
   ::= (cardinality <cardinality-specification>
       <cardinality-specification>)
<cardinality-specification> ::= <integer> | ?VARIABLE
```

Переменные и выражения

```
<single-field-variable> ::= ?<variable-symbol>
<multifield-variable> ::= $?<variable-symbol>
<global-variable> ::= ?*<symbol>*
<variable> ::= <single-field-variable> |
               <multifield-variable> |
               <global-variable>
<function-call> ::= (<function-name> <expression>*) |
                    (<special-function-name>
                    <special-function-arguments>)
<special-function-name>
   ::= имя функции, такой как assert или make-instance,
       которая не выполняет синтаксического анализа своих
       параметров предусмотренным по умолчанию способом,
       который используется стандартными функциями.
<special-function-arguments>
   ::= В специальных функциях <special-function-name>
       используется нестандартный способ синтаксического
       анализа параметров. Для ознакомления с синтаксисом
       параметров специальных функций,
       <special-function-arguments>, обращайтесь к
       документации по каждой специальной функции.
       Например, в вызове функции (assert (value 3))
       параметр (value 3) - не вызов функции value с
       параметром 3, а факт, который должен быть внесен
       в список фактов.
<expression> ::= <constant> |
                 <variable> |
                 <function-call>
```

Типы данных

```
<symbol>
   ::= действительный символ, как указано в разделе 7.4
<string>
   ::= допустимая строка, как определено в разделе 7.4
<float>
   ::= число с~плавающей точкой, имеющее допустимый
       формат, как определено в разделе 7.4
```

```
<integer>
   ::= целое число, имеющее допустимый формат, как
       указано в разделе 7.4
<instance-name>
   ::= действительное имя экземпляра, как указано
       в разделе 7.4
<number> ::= <float> | <integer>
<lexeme> ::= <symbol> | <string>
<constant> ::= <number> | <lexeme>
<comment> ::= <string>
<variable-symbol>
   ::= терм <symbol>, начинающийся с алфавитного знака
<function-name> ::= <symbol>
<name> ::= <symbol>
<...-name>
   ::= терм <symbol>, в котором многоточие указывает,
       что представляет собой символ. Например,
       <deftemplate-name> - символ, который представляет
       имя конструкции deftemplate.
```

Приложение Ж

Программные ресурсы

В настоящем приложении приведены сведения о современных ресурсах Web, касающихся той тематики, которая обсуждается в каждой главе книги. Но материал приложения может также использоваться во время классных занятий, поскольку с его помощью можно дополнить содержание книги, поручая студентам исследовать программное обеспечение, а также подготавливать презентации, рассматриваемые на лекциях, и демонстрационные версии. Безусловно, никто не оспаривает достоинств обычных занятий, проводимых в аудиториях с применением учебника, но студенты получают дополнительные преимущества в обучении, открывая новые знания путем самостоятельного поиска.

Многие из коммерческих программных продуктов могут быть загружены в виде пробных версий, а в состав таких версий входят примеры, предназначенные для ознакомления с возможностями программного продукта. Но лучше всего поручить студентам самостоятельно подготавливать примеры и сравнивать результаты. В частности, целесообразно провести сравнение эффективности решения задачи коммивояжера, рассматриваемой в главе 1, с помощью стандартного жадного алгоритма (обсуждаемого в курсах по структурам данных), искусственных нейронных сетей многих типов (включая определение наиболее эффективного типа), генетических алгоритмов, эволюционного алгоритма муравейника (Ant Colony Evolutionary Algorithm) и других методов, как показано в данном приложении.

Студентам особенно интересно наблюдать за тем, как изменяются характеристики методов, основанных на искусственном интеллекте, по сравнению с методами, базирующимися на стандартных алгоритмах, по мере возрастания масштабов задачи от нескольких до сотен городов. Еще больший смысл такое сопоставление методов приобретает, когда данный пример соотносится с ситуацией, наблюдаемой в реальном мире, например, при рассмотрении маршрутов, проходящих через разные города, по которым летают самолеты компании Southwest Airlines. Отме-

тим, что в Web можно легко найти информацию обо всех городах, обслуживаемых компанией Southwest Airlines, а также о географическом местонахождении этих городов, определяемом по широте и долготе, и воспользоваться этой информацией в качестве набора данных.

Возможности трудоустройства

В наши дни быстрее всего развиваются такие пять областей применения искусственного интеллекта и экспертных систем, как оборона, медицина, бизнес, промышленность и видеоигры, поэтому, кроме многочисленных поисковых узлов общего назначения, существуют узлы, посвященные решению проблем трудоустройства, которые специально предназначены для привлечения к работе людей, владеющих компьютером и способных применять свои знания в этих областях. При подготовке специалистов подобного профиля рассматриваются многие темы, рассматриваемые в настоящей книге. Чтобы приобрести дополнительный опыт, загрузите программные демонстрационные версии и выполните примеры, описанные в разделах данного приложения, посвященных каждой главе, как описано ниже.

- Аналитики данных (главы 1–12).
- Системные проектировщики (главы 1–12).
- Разработчики (главы 1–12).
- Эксперты в предметной области. (Безусловно, изучение данной книги не позволит стать экспертом в какой-то конкретной предметной области, но с ее помощью можно приобрести квалификацию инженера по знаниям. В частности, практические рекомендации по проведению собеседований с экспертами в предметной области, что является основной задачей для инженеров по знаниям, приведены в главе 6.)
- Специалисты по методам поиска (главы 1 и 2).
- Специалисты по системам, основанным на правилах (конечные автоматы, деревья решений и продукционные системы; главы 1–3, 6–12).
- Специалисты по теории игр и деревьям игры (для обозначения последних применяются также термины “конечные автоматы” и “деревья решений”; глава 1)
- Специалисты по методам моделирования искусственной жизни и стайного поведения (глава 1).
- Специалисты по методам планирования (глава 2).
- Специалисты по нечеткой логике (главы 4 и 5).

- Специалисты по искусственным нейронным сетям, генетическим алгоритмам и сетям доверия (главы 1 и 4).

Ниже приведен список, в котором можно найти ссылки на некоторые узлы, посвященные трудоустройству. Дополнительные возможности предоставляет также посещение начальных страниц компаний, которые публикуют свои объявления в оперативных коммерческих журналах, подобных журналу *PCAI.com* и другим журналам, перечисленным в этом приложении. Кроме того, в настоящее время в оперативном режиме можно получить доступ ко всем научным и коммерческим журналам, которые обычно публикуют списки вакансий в конце выпуска. Электронная подписка на такие журналы, позволяющая получить к ним бесплатный оперативный доступ, должна быть оформлена в любом учебном заведении по данному профилю. Кроме того, объявления о приеме на работу часто публикуются в оперативных источниках, перечисленных в разделе "Группы новостей" данного приложения. Фактически заинтересованный в этом специалист может реализовать проект по созданию агента с искусственным интеллектом, который просматривает избранные группы новостей в поиске объявлений о приеме на работе и доставляет пользователю полученную информацию.

Но рекомендуем соблюдать осторожность: выставляя на узле или отправляя свое резюме, не включайте личную информацию, такую как номер карточки социального обеспечения, разглашение которой может создать предпосылки совершения правонарушений с использованием ваших идентификационных данных. В связи с распространением онлайновых технологий значимость этой проблемы постоянно возрастает. Не следует передавать какую-либо личную информацию, которая может использоваться для совершения правонарушений на основе ваших идентификационных данных, даже если вы ведете конфиденциальную переписку с возможным работодателем по электронной почте (независимо от того, насколько привлекательными являются его предложения).

Не следует беспокоиться, если ваши характеристики не соответствуют всем квалификационным требованиям. Пятый закон успеха гласит, что приз присуждается любому, кто добрался до финиша, если он окажется единственным. Это означает, что зачастую достаточно лишь соответствовать большинству квалификационных требований и быть готовым принять практически любую зарплату, на которую решит потратиться компания. Безусловно, многим работодателям хотелось бы привлечь к работе специалиста, имеющего десятилетний опыт использования двадцати различных языков программирования, но чаще всего бюджет компании не позволяет выплачивать зарплату, соответствующую требованиям такого работника, поэтому самые лучшие шансы будут у вас. Получение работы можно сравнить с заключением брака. Вы вступаете в брак с наилучшим кандидатом, которого вам удается найти к этому времени, а о его недостатках узнаете потом.

Ссылки на узлы, посвященные трудоустройству

Ссылки на узлы, посвященные трудоустройству, приведены ниже.

- http://www.aaai.org/Magazine/Jobs.
- http://www.mary-margaret.com/.
- http://www.blizzard.com/jobopp/.
- http://www.aktor-kt.com/jobs.htm.
- http://www.cdacindia.com/html/aai/aaidx.asp.
- http://www.lplizard.com/lounge/jobs_programmer.htm.
- http://corporate.infogrames.com/corp_hrmain.php?action=jobdetails&jobID=218&locationID=7.
- http://www.insomniacgames.com/html/about/jobs.html.
- http://www.alifemedical.com/careers.html.
- http://www.hirehealth.com/ci/servlet/com.ci.jobseeker.JobDetails;jsessionid=79FDF0992C80EB299B6B5778CBB6EB1D?JOB_ID=48179.
- http://www.shrinershq.org/cgi-bin/classifieds/classifieds.cgi.
- http://www.genesciences.com/DNAjobsNews/12June04.htm.
- http://www.business.com/search/rslt_default.asp?query=medical+field&bdct=&bdcf=&vt=all&type=jobs&search=Next+Search.
- http://www.aegiss.com/html/jobs.html.
- http://www.gamedev.net/directory/careers/.
- http://www.capcom.com/jobs/job.xpml?jobid=400016.
- http://www.gamerecruiter.com/.
- http://www.3drealms.com/gethired.html.

Оперативные энциклопедии

В оперативных энциклопедиях приведены краткие и понятные пояснения терминов. В некоторых из этих пояснений можно найти примеры и дополнительные ссылки. Настоятельно рекомендуется при изучении данной книги ознакомиться с определениями представленных в ней новых терминов, отыскивая в дополнительных источниках объяснения и примеры (это предложение не относится к студентам-отличникам и к тем, кто считает, что эта книга идеальна сама по себе).

- Wikipedia. Превосходная крупная энциклопедия с открытым информационным наполнением на многих языках, в которой приведены четкие описания многих терминов и даны ссылки на источники, предназначенные для дополнительного чтения (`http://en.wikipedia.org/wiki/Main_Page`).
- Webopedia. Хорошая общая оперативная энциклопедия, посвященная компьютерным наукам (`http://www.webopedia.com/`).
- Platonic Realms. Весьма доступная для восприятия студента оперативная математическая энциклопедия, приведенная вместе с другими ресурсами; настоятельно рекомендуем (`http://www.mathacademy.com/pr/`).
- Mathworld. Математическая энциклопедия, предоставляемая компанией Wolfram, которая выпустила программу Mathematica (`http://mathworld.wolfram.com/`).
- The Internet Encyclopedia of Philosophy. Хорошая оперативная энциклопедия по философии и логике, которая особенно должна помочь при изучении глав 2–5 (`http://www.iep.utm.edu/`).
- Stanford Encyclopedia of Philosophy. Общие определения и объяснения в области философии и логики; с этой оперативной энциклопедией можно ознакомиться по адресу `http://plato.stanford.edu/`.
- Microsoft Multi-University/Research Laboratory. Интересные оперативные лекции по многим темам, прочитанные известными специалистами под эгидой межуниверситетской научно-исследовательской лаборатории Microsoft (`http://murl.microsoft.com/ContentMap.asp`).
- Google. Специалисты узла Google утверждают, что наибольший объем ресурсов по логике в мире находится по следующему адресу: `http://www.uni-bonn.de/logic/world.html`.

Группы новостей

Многочисленные сообщения, публикации с ответами на часто задаваемые вопросы, информационные ресурсы, материалы конференций и всевозможные новости об искусственном интеллекте, экспертных системах и тысячах других тем можно найти в группах новостей. Кроме того, в группах новостей и научных журналах часто можно встретить списки вакансий в тех областях, которым они посвящены, таких как экспертные системы и искусственный интеллект. В качестве примера можно указать, что вакансии для специалистов по экспертным системам в Web часто публикуются в группе новостей `http://mailgate.supereva.it/comp/comp.ai.shells/`, а для специалистов по искусственному интеллекту — в ведущей по отношению к ней группе новостей

http://mailgate.supereva.it/comp/comp.ai/. Но если вы действительно захотите зарегистрироваться в этих группах новостей, то должны знать не только слово “pizza”, а гораздо больше итальянских слов, и разбираться в географии Италии лучше, чем было бы достаточно для того, чтобы просто рассказать, где придумали сосиски пепперони!

Ответы на общие вопросы о языках экспертных систем, таких как CLIPS, можно получить, передав их в группу новостей comp.ai.shells или отправив на официальный узел CLIPS, который указан в приложении Г. Для доступа к группам новостей, посвященным искусственному интеллекту, предусмотрены три способа. Один из способов состоит в том, чтобы узнать имя сервера новостей от провайдера служб Интернета (Internet Service Provider — ISP) и ввести его в своем браузере. Для этого в браузере Internet Explorer нужно выбрать меню Tools, затем подменю Mail and News и, наконец, находящееся внизу подменю Read Mail. Откроется новое окно и появится ссылка, в которой указано Set up a newsgroups account. После этого необходимо следовать указаниям программы-мастера Internet Connection Wizard.

Более простой способ состоит в том, чтобы перейти по адресу http://www.google.com и щелкнуть на выделенной красным цветом ссылке Groups, находящейся над окном поиска, в котором вводятся ключевые слова для поиска. Появится список общих типов групп новостей. Щелкните на ссылке comp. и ознакомьтесь с именами некоторых групп новостей, которые могут представлять для вас интерес, учитывая то, что наличие звездочки указывает на существование многочисленных подгрупп:

- comp.job.* (1 группа).
- comp.jobs.* (6 групп).
- comp.jobsoffered.

В других группах новостей, посвященных конкретным языкам, могут также быть развернуты подгруппы с предложениями о найме на работу. В наши дни при предоставлении какой-либо персональной идентификационной информации, которая может использоваться для совершения правонарушений со стороны злонамеренных лиц, приходится соблюдать исключительную осторожность. После перехода к общему множеству групп новостей comp. можно щелкнуть на ссылке comp.ai, чтобы ознакомиться со всеми группами новостей, посвященными искусственному интеллекту.

А для того чтобы воспользоваться третьим способом, достаточно просто набрать http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&group=comp.ai в поле адреса браузера, и вы сразу же перейдете к требуемой группе новостей. На экране появится примерно такая информация:

- comp.ai.
- comp.ai.jair.announce.

- comp.ai.alife.
- comp.ai.jair.papers.
- comp.ai.doc-analysis.misc.
- comp.ai.nat-lang.
- comp.ai.analysis.ocr.
- comp.ai.neural-nets.
- comp.ai.edu.
- comp.ai.nlang-know-rep.
- comp.ai.fuzzy.
- comp.ai.philospohy.
- comp.ai.games.
- comp.ai.shells.
- comp.ai.gentic.
- comp.ai.vision.

Следует отметить, что слово jair представляет собой сокращение от *Journal of AI Research*. Так называется реферативный журнал, предоставляемый в оперативном режиме. Превосходная особенность групп новостей состоит в том, что они позволяют публиковать свои вопросы и отвечать на чужие. Имеется также много групп новостей, в которых приведены перечни вакансий по государствам, штатам и даже некоторым городам. Для поиска этих групп новостей может применяться ключевое слово job.

Одним из наилучших способов, позволяющих приступить к изучению конкретной темы, такой как искусственный интеллект, нейронные сети, генетические алгоритмы и т.д., а также кратко ознакомиться со многими ресурсами, является получение списка часто задаваемых вопросов с ответами (Frequently Asked Questions — FAQ). Документы FAQ могут служить хорошей отправной точкой для любого новичка в какой-то конкретной области. Эти документы составлены добровольцами, обладающими знаниями в соответствующей области, и предоставляются бесплатно. Безусловно, в этих документах часто упоминаются коммерческие программные продукты, однако обсуждение не только подобного программного обеспечения, но и всех других программ проводится с максимальной объективностью. Хотя и можно подписаться на любую группу новостей, которая выглядит достаточно многообещающей, и только потом прочитать предоставляемые в ней документы FAQ, лучше всего поступить прямо противоположно. К документам FAQ для всех этих тысяч групп новостей можно получить доступ на странице, поддерживающей режим поиска. Эта страница находится по адресу http://www.faqs.org/faqs/ и позволяет проводить поиск документов

FAQ по ключевым словам, а также просматривать списки этих документов по категориям либо по алфавиту. Это очень удобно, поскольку, например, все документы FAQ, относящиеся к группе новостей по искусственному интеллекту и ее подгруппам, можно найти по адресу `http://www.faqs.org/faqs/ai-faq/`. Большой список из всех документов FAQ, относящихся к компьютерным наукам, находится по адресу `http://www.faqs.org/faqs/by-newsgroup/comp/`.

Оперативные научные и коммерческие журналы

Научные и коммерческие журналы, не только издаваемые на бумаге, но и предоставляемые в оперативном режиме, позволяют ознакомиться с последними новостями и статьями по конкретной тематике, а также являются хорошими источниками для поиска вакансий или сообщений о конференциях, практикумах и семинарах. Основное различие между научными и коммерческими журналами состоит в том, что научные журналы главным образом поддерживаются их подписчиками, а коммерческие зарабатывают основную часть денег с помощью рекламы. Научные журналы не изменяют изначально принятым ими принципам свободы от коммерции, чтобы никто не мог сомневаться в непредвзятости их публикаций. Еще одной характерной особенностью научных журналов является то, что публикуемые в них статьи оценивают несколько рецензентов, поэтому настоящим научным журналом можно считать только тот, статьи которого рецензируются, а не тот, который содержит в своем названии слово “Journal”.

В коммерческих журналах нередко встречаются статьи, написанные авторами, которые работают в компаниях, публикующих рекламу в тех же журналах. С другой стороны, и в научных журналах часто встречаются статьи, написанные членами редакционной коллегии или сотрудниками тех организаций, в которых работают члены коллектива журнала. Особую осторожность следует проявлять при чтении статей, в которых имеется такая сноска, что “исследование отчасти поддерживалось компанией *XYZ Group*”, поскольку авторы этих статей смогут рассчитывать на дальнейшее финансирование, только если будут оценивать перспективы получения дальнейших результатов от своей работы оптимистически, а не пессимистически. Кроме того, и для компании *XYZ Group* должно быть также продемонстрировано, что ее гранты и контракты позволили выполнить значительный объем работы, судя по тому, что опубликовано такое количество статей или получено дополнительное финансирование от правительственных агентств.

Но наиболее тщательной проверке на объективность должны подвергаться статьи, опубликованные по материалам конференций, поскольку на конференциях иногда принимают к рассмотрению даже такие работы, которые предназначены лишь для оправдания командировочных расходов, а другие публикации пред-

ставляют собой просто незавершенную продукцию, содержат в себе материалы, собранные студентами при подготовке дипломов, или служат выражением благодарности компании *XYZ Group*. Именно поэтому в библиографии, прилагаемой к данной книге, не упоминаются какие-либо труды конференций.

Перечень наиболее интересных журналов Web-узлов приведен ниже.

- *AI Case-Based Reasoning Journal*. Журнал, посвященный тематике рассуждений на основе прецедентов (`http://www.ai-cbr.org/theindex.htm`).
- *BusinessWeek Online*. Журнал, посвященный экономике и предпринимательству (`http://www.businessweek.com`). Безусловно, этот оперативный коммерческий журнал не относится к категории научных журналов, но имеет хорошую поисковую машину по таким ключевым словам, как `artificial intelligence` (искусственный интеллект), и применяется для публикации статей о новостях высоких технологий в бизнесе, управлении государством и военном деле, поскольку в этих трех областях тратится много денег. Имеет также смысл регулярно просматривать журнал *BusinessWeek* для ознакомления с коммерческими приложениями новейших компьютерных технологий. Именно в этом журнале можно узнать о том, что компания Wal-Mart использует анализ скрытых закономерностей в данных по всем своим трем тысячам магазинов, что позволяет предсказывать сбыт каждого товара по каждому из магазинов со сверхъестественной точностью, а также о том, какое широкое распространение получили в наши дни методы искусственного интеллекта (`http://www.businessweek.com/bw50/content/mar2003/a3826072.htm`).
- *American Association for Artificial Intelligence* (`http://www.aaai.org/`).
- *Expert Systems International Journal* (`http://www.blackwellpublishing.com/journal.asp?ref`).
- *Expert Systems Journal* (`http://www.aaai.org/AITopics/html/expert.html`).
- *Fuzzy Optimization and Decision Making*. Научный журнал по моделированию и вычислениям в условиях неопределенности (`http://www.kluweronline.com/issn/1568-4539`).
- *IEEE Intelligent Systems Journal* (`http://www.computer.org/intelligent/`).
- *IEEE Transactions on Fuzzy Systems* (`http://www.ieee.org/portal/index.jsp?pageID=corp_level1&path=pubs/transactions&file=tfs.xml&xsl=generic.xsl`).

- *IEEE Transactions on Neural Networks* (`http://www.ieee.org/portal/index.jsp?pageID=corp_level1&path=pubs/transactions&file=tnn.xml&xsl=generic.xsl`).
- *Journal of Artificial Intelligence Research* (`http://www.cs.washington.edu/research/jair/home.html`).
- *Neural Computation. Публикуется издательством MIT Press* (`http://mitpress.mit.edu/catalog/item/default.asp?sid=B395B969-A0A5-4F6E-A566-AF7B2EDCE494&ttype=4&tid=31`).
- *Neural Computing & Applications. Публикуется издательством Springer-Verlag* (`http://springerlink.metapress.com`). Введите указанный адрес издательства и щелкните на ссылке Browse Publications. После этого щелкните на ссылке Jump to N, а затем на Neural Computing and Applications.
- *North American Fuzzy Information Processing Society* (`http://morden.csee.usf.edu/Nafipsf/`). На этом узле представлены ссылки на многочисленные информационные ресурсы, предоставляемые в оперативном режиме, которые посвящены нечетким методам.
- *PCAI.com* (`http://PCAI.com`). Превосходный оперативный коммерческий журнал с хорошей поисковой машиной. Журнал *PCAI* публикуется с 1980-х годов и содержит обширный репозитарий информации об искусственном интеллекте со ссылками на программное обеспечение, статьи и другие ресурсы, сгруппированные следующим образом: C++ (Язык C++), Blackboard Technology (Технология классной доски), Client/Server (Технология "клиент/сервер"), Dylan (Язык Dylan), Creative Thinking (Творческое мышление), Data Mining (Анализ скрытых закономерностей в данных), Forth (Язык Forth), Distributed Computing (Распределенные вычисления), Expert Systems (Экспертные системы), LISP (Язык LISP), Fuzzy Logic (Нечеткая логика), General AI Sites (Основные узлы по искусственному интеллекту), Logo (Язык Logo), Genetic Algorithms (Генетические алгоритмы), OPS (Командный интерпретатор OPS), Intelligent Agents (Интеллектуальные агенты), Intelligent Applications (Интеллектуальные приложения), Prolog (Язык Prolog), Internet (Интернет), Logic Programming (Логическое программирование), Scheme (Язык Scheme), Machine Learning (Машинное обучение), Modeling and Simulation (Моделирование и эмуляция), Smalltalk (Язык Smalltalk), Multimedia (Мультимедиа), Natural Language Processing (Обработка естественного языка), Neural Networks (Нейронные сети), Object Oriented Development (Объектно-ориентированная разработка), Optical Character Recognition (Оптическое распознавание знаков), Robotics (Робототехника), Speech Recognition (Распознавание речи), Virtual Reality (Виртуальная реальность).

- *Generation5.com* (`http://www.generation5.org/`). Хороший и разносторонний узел по искусственному интеллекту, на котором можно найти новости и информацию, касающиеся всех аспектов применения искусственного интеллекта, включая игры. Включает много ссылок на ресурсы, которые подразделяются на следующие основные области искусственного интеллекта: Aerospace and Military (Космос и вооруженные силы), Agents (Агенты), Artificial Intelligence (Искусственный интеллект), Artificial Life (Искусственная жизнь), Biometrics (Биометрия), Constraint Satisfaction Programming (Программирование в ограничениях), Creativity (Artificial) (Творчество на базе искусственного интеллекта), Fractals (Фракталы), Chaos (Хаос), Complexity (Сложность), Non-linear Dynamics (Нелинейная динамика), Gaming (Деловые игры), Genetic Algorithms (Генетические алгоритмы), Gesture Recognition (Распознавание мимики), Home Automation (Автоматизация жилья), Image Analysis/Recognition (Анализ и распознавание изображений), Knowledge-based Systems (Системы, основанные на знаниях), LISP Programming Language (Язык программирования LISP), Natural Language Processing (Обработка естественного языка), Neural Networks (Нейронные сети), Neuroscience (Неврология), Pattern Recognition (Распознавание образов), Personal (Приложения для личного использования), Philosophy (Философия), Prolog (Язык Prolog), Robotics (Робототехника), Furby (Игра Furby), LEGO Mindstorms (Мозговые штурмы на основе LEGO), Scheme Programming Language (Язык программирования Scheme), Speech Recognition and Synthesis (Распознавание и синтез речи), Commercial (Коммерческие приложения), VoiceXML (Язык VoiceXML).
- *SourceForge.Net* (`http://sourceforge.net/`). Важный источник программного обеспечения и других ресурсов. Относится к категории крупнейших Web-узлов. На нем базируется свыше 90 тысяч проектов с открытым исходным кодом, включая ресурсы по всем темам, обсуждаемым в данной книге, и многим другим.
- *AI@HOME* (`http://www.aiathome.com/`). Хороший оперативный узел по искусственному интеллекту, на котором публикуются все программы с открытым исходным кодом по нейронным сетям, эволюционным алгоритмам, метаобучению и распределенным вычислениям; метаобучение — это использование алгоритмов обучения для создания новых алгоритмов обучения.
- *Dr. Dobbs Portal AI Links* (`http://www.ddj.com/documents/s=7730/ddj0212ai/0212ai001.html{\#}egyptian_cu`). Большая коллекция статей, учебников, ссылок и программного обеспечения, предоставляемая коммерческим журналом общего назначения, посвященным многим языкам программирования, *Dr. Dobbs Journal*.

- *Adaptive Behavior* (`http://www.isab.org/journal/`). Международный научный журнал по исследованию адаптивного поведения животных и таких автономных искусственных систем, как роботы. В число рассматриваемых тем входят восприятие и управление движениями, обучение, эволюция, выбор действий и поведенческие последовательности, мотивация и эмоции, распознавание характеристик окружающей среды, коллективное и социальное поведение, навигация, коммуникация и обработка сигналов.
- *Artificial Life* (`http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=41`). Крупный узел, посвященный исследованиям в области искусственного интеллекта, на котором представлено много коммерческих приложений. Кинематографическая промышленность приносит доход в 20 миллиардов долларов в год, а видеоигры являются источником еще большего дохода. Кроме того, искусственный интеллект открывает перспективу создания многих приложений военного назначения. На этом узле публикуются материалы исследований в области научных, инженерных, философских и социальных проблем, связанных с синтезом поведения, аналогичного поведению живых существ с применением компьютеров, механических устройств, химических молекул и других альтернативных средств, начиная с самых элементарных форм поведения. Кроме того, на нем представлены исследования в области происхождения жизни, самосборки, роста и развития, эволюционной и экологической динамики, поведения животных и роботов, социальной организации и эволюции культуры. Достижения в области создания искусственной жизни находят важное применение в видеоиграх и кинофильмах, таких как *Lord of the Rings* (Властелин колец) и *I, Robot* (Я, робот), в которых искусственные формы жизни используются для создания изображений персонажей, участвующих в битвах, с помощью чего формируются грандиозные батальные сцены, включающие тысячи персонажей.

Художники-аниматоры, работающие на компьютерах в традиционном стиле, и кинорежиссеры не могут справиться без привлечения средств искусственного интеллекта с задачей координации действий тысяч персонажей, созданных с помощью компьютерной анимации. В связи с этим была проведена большая работа по обучению искусственных форм жизни навыкам борьбы и сложным стратегиям ведения сражений, выбранным режиссером как наиболее интересные. Внедрение этих новых технологий привело к революционным изменениям в процессе съемки фильмов по сравнению с тем применявшимся когда-то съемочным процессом, в котором режиссер руководил действиями отдельных актеров или анимационных персонажей. В фильме *Star Wars 2: Attack of the Clones* (Звездные войны 2: атака клонов) для имитации толпы, следящей за гонками, кинематографисты использовали цветные фигурки, а теперь у них появилась возможность создавать

огромные, динамические массовки с использованием искусственных интеллектуальных форм жизни. Горькая правда состоит в том, что фильм *Matrix* может оказаться верным изображением перспектив человечества, если не считать того, что в нем реальные люди находились в ловушке, созданной в виртуальной среде; он показал, как постепенно все шире распространяются технологии, не требующие участия людей.

- *Computational Linguistics* (`http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=10`). На этом узле представлены материалы по проектированию и анализу систем обработки естественного языка; узел посвящен искусственному интеллекту, когнитологии, анализу речевой деятельности, психологии восприятия языка и вопросам производительности.
- *Computer-Mediated Communication Magazine* (`http://www.december.com/cmc/mag/meta/`). Оперативный коммерческий журнал, в котором публикуются последние новости, распределенные по таким рубрикам: информационная технология, управления знаниями, электронная коммерция, участники разработок, предстоящие мероприятия, технологические новинки, государственная политика, культура, практические рекомендации, а также исследования и приложения, относящиеся к сфере человеческого общения и взаимодействия в оперативной среде. Предоставляет полный обзор современных событий и достижений в области технологии.
- *Decision Support Systems and Electronic Commerce* (`http://www.elsevier.com/wps/find/journaldescription.cws_home/505540/description#description`). Научный журнал, посвященный описанию методов реализации и оценки систем поддержки принятия решений, применяемых в искусственном интеллекте, когнитологии, кооперативной работе с компьютерной поддержкой, управлении базами данных, теории принятия решений, экономике, лингвистике, управленческой науке, математическом моделировании, психологии управления операциями, системах управления пользовательским интерфейсом и в других областях.
- *Electronic Journals and Periodicals in Psychology* (`http://psych.hanover.edu/Krantz/journal.html`). Этот узел представляет собой не оперативный научный журнал, а обширный список ссылок. Создатели узла стремятся предоставить ссылки на все электронные научные журналы, публикации трудов конференций и другие периодические издания, посвященные психологии. Одной из великолепных особенностей узла является система предупреждения, с помощью которой его подписчикам по электронной почте рассылается информация об интересующих его новинках. В значительной части искусственный интеллект обязан свои-

ми достижениями именно психологии, поскольку люди в течение десятков тысяч лет своей эволюции изобрели некоторые очень удачные (а также довольно неудачные) методы решения задач. Тем, кто хочет серьезно заняться подготовкой в области искусственного интеллекта, необходимо изучать работы по психологии, биологии, неврологии, философии и многим другим областям для расширения своих познаний (само это слово также является термином из психологии).

- *AI Events* (`http://www.aievents.co.uk/`). Этот узел — также не научный журнал, а очень полезная поисковая база данных по предстоящим конференциям, семинарам, летним школам и аналогичным мероприятиям, посвященным искусственному интеллекту; участие в подобном событии — превосходный способ провести отпуск.
- *Evolutionary Computation* (`http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=25`). Узел научного журнала по вычислительным системам, создаваемым на основании изучения живой природы. На этом узле особое внимание уделено эволюционным алгоритмам (Evolutionary Algorithm — EA), генетическим алгоритмам (Genetic Algorithm — GA), стратегиям развития (Evolution Strategy — ES), эволюционному программированию (Evolutionary Programming — EP), генетическому программированию (Genetic Programming — GP), системам классификаторов (Classifier System — CS) и другим естественным вычислительным методам, созданным на основании изучения биологических систем.
- *International Journal of Human-Computer Studies/Knowledge Acquisition* (`http://repgrid.com/IJHCS/`). Материалы этого узла особенно полезны для изучения главы 6; рассматривается широкий перечень тем, перечисленных ниже.
  - Интеллектуальные пользовательские интерфейсы.
  - Взаимодействие с помощью естественного языка.
  - Учет личностных факторов при проектировании мультимедийных систем.
  - Учет личностных и социальных факторов при проектировании средств виртуальной реальности.
  - Учет личностных и социальных факторов при проектировании средств World Wide Web.
  - Учет личностных и социальных факторов при разработке программного обеспечения.
  - Организация совместной работы с помощью компьютеров.

  - Речевое взаимодействие.
  - Графическое взаимодействие.
  - Приобретение знаний.
  - Системы, основанные на знаниях.
  - Гипертекстовые и гипермедийные представления.
  - Пользовательское моделирование.
  - Эмпирические исследования поведения пользователей.
  - Психология программирования.
  - Теория систем и основы взаимодействия “человек–компьютер”.
  - Системы управления пользовательским интерфейсом.
  - Информация и системы поддержки принятия решений..
  - Инженерия требований.
  - Новаторские проекты и приложения интерактивных систем.
- *Journal of Artificial Intelligence Research* (`http://www.cs.washington.edu/research/jair/home.html`). Весьма разносторонний оперативный научный журнал, в котором рассматриваются все темы искусственного интеллекта.
- *Journal of Cognitive Neuroscience* (`http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=12`). На этом узле представлены материалы по взаимодействию деятельности мозга и поведения, результаты изучения функционирования мозга и феноменов, наблюдаемых при этом в мозгу, а также сведения по неврологии, нейропсихологии, когнитивной психологии, нейробиологии, лингвистике, компьютерным наукам и философии.
- *Journal of Constructivist Psychology* (`http://www.tandf.co.uk/journals/titles/10720537.asp`). Узел посвящен конструктивизму — одному из успешных направлений науки об образовании и о том, как происходит обучение людей.
- *Journal of Mind and Behavior* (`http://kramer.ume.maine.edu/~jmb/welcome.html`). Теоретический научный журнал, в котором предпринимаются попытки установить связь между мышлением и поведением.
- *Noetica: A Cognitive Science Forum* (`http://www.drc.ntu.edu.sg/users/mgeorg/enter.epl`). Оперативный реферируемый научный журнал по вопросам когнитологии, машинного обучения и многим другим темам, относящимся к искусственному интеллекту.

- *PRESENCE: Teleoperators and Virtual Environments*. Великолепный научный журнал, в котором публикуются сведения о важных приложениях в области телехирургии и о приложениях военного назначения, таких как беспилотные летательные аппараты, используемые для проведения воздушной разведки без участия пилота. Беспилотные летательные аппараты широко применяются в вооруженных силах, поскольку они дешевле по сравнению с пилотируемыми самолетами, сложнее поддаются обнаружению, а также способны летать над целью и оставаться дольше, чем воздушные суда с экипажем. А самое значительное их преимущество состоит в том, что единственный оператор, работающий в режиме дистанционного присутствия, может управлять многочисленными беспилотными летательными аппаратами. При этом, если один из таких аппаратов будет сбит, в прессе не поднимется шумиха, как произошло в известном инциденте с Фрэнсисом Гэри Пауэрсом в 1962 году.

Предполагается, что к 2010 году на вооружение в войска поступит интеллектуальный аналог беспилотного летательного аппарата — беспилотный летательный аппарат военного назначения (Unmanned Combat Air Vehicle — UCAV) X-45 (`http://www.fas.org/man/dod-101/sys/ac/ucav.htm`). Это должен быть самолет класса “бомбардировщик-разведчик”, способный летать на заранее известное расстояние по циклическому маршруту без пилота на борту. Судя по тому, что показывает телевидение США, ведется разработка многих интеллектуальных систем оружия стоимостью в миллиарды долларов. Такие программы, демонстрируемые по кабельному телевидению, как *Mail Call* на *History Channel*, *Tactics to Practice* на *Discovery Channel* и *Future Fighting Machines* на канале *G4TechTV*, показывают, что на военные системы, создаваемые на базе искусственного интеллекта, затрачиваются миллионы долларов.

Действительно, впечатляет наблюдение за тем, насколько действия летательного аппарата X-45 напоминают полет самолета с искусственным интеллектом под управлением вымышленной программы с искусственным интеллектом Skynet в серии фильмов *Terminator* (в этой серии военизированный искусственный интеллект пытается отвоевать мир у людей). Та же тема повторяется в кинофильме *I, Robot*, выпущенном в 2004 году, и восходит к классическому фильму 1970 года *Colossus: The Forbin Project*, в котором американские и российские суперкомпьютеры объединяются, чтобы не дать людям уничтожить мир.

- *Public E-print Archive* (`http://www.ecs.soton.ac.uk/~harnad/genpub.html`). Статьи Стивена Харнада (Stevan Harnad) по когнитивной психологии, в которых охвачены многие интересные темы, такие как индукция, познание, машинное обучение, робототехника и многие другие; эти статьи вполне доступны для восприятия.

- *PSYCHE* (`http://psyche.cs.monash.edu.au/`). Междисциплинарный научный журнал, посвященный исследованиям феномена сознания, в котором публикуется много интересных статей.
- *Shufflebrain* (`http://www.indiana.edu/~pietsch/home.html`). Исследования на тему того, благодаря чему физический объект, мозг, обладает способностью воплощать в себе разум, а также многих других интересных вопросов, в том числе перечисленным в разделе узла Popular Science Menu.
- *The Journal of Computer-Mediated Communication* (`http://www.ascusc.org/jcmc/`). Этот узел не посвящен непосредственно искусственному интеллекту, но на нем представлены материалы из многих областей, которые могут привести к новым разработкам и новым достижениям в области искусственного интеллекта.
- *The Journal of Mind and Behavior* (`http://kramer.ume.maine.edu/~jmb/welcome.html`). Данный узел в основном предназначен для психологов, но может подсказать, в каких областях могут быть развернуты новые направления исследований по искусственному интеллекту.

Глава 1

Ресурсы искусственного интеллекта

Основным источником финансирования разработок в области искусственного интеллекта всегда было агентство DARPA (Defense Advanced Research Projects Agency — Управление перспективных исследовательских программ) Министерства обороны США. Кроме того, агентство DARPA предоставило финансирование, на основе которого началась разработка сети ARPANET, которая в 1990-х годах была преобразована в коммерческую сеть Интернет. По утверждениям представителей DARPA, применение методов искусственного интеллекта в планировании снабженческой деятельности при проведении операции “Буря в пустыне” во время войны в Персидском заливе в 1990-х годах во много раз окупило все затраты на исследования по искусственному интеллекту с 1950-х годов (`http://www.au.af.mil/au/aul/school/acsc/ai02.htm`).

- *American Association for Artificial Intelligence*. Хорошая отправная точка для получения знаний об искусственном интеллекте и экспертных системах (`http://www.aaai.org/AITopics/html/welcome.html`). Для ознакомления со многими статьями по экспертным системам и с другими ресурсами перейдите также по ссылке на следующий адрес: `http://www.aaai.org/AITopics/html/expert.html#good`.
- *Out of Control: The New Biology of Machines*. Превосходная книга Кевина Келли (Kevin Kelly) о том, как искусственный интеллект влияет на общество

и экономику, которую можно читать в оперативном режиме; вполне заслуживает того, чтобы ее прочесть (`http://www.kk.org/outofcontrol/`).

- Metaxiotis K. and Psarras, J. *Expert Systems in Business: Applications and future directions for the operations researcher*, Industrial Management & Data Systems, 2003, Vol. 103, No. 5, p. 361–368. Статья посвящена вопросам применения экспертных систем в бизнесе.
- Hugh McKellar, *Artificial intelligence: Past and future*, KMWorld Magazine, Vol. 12, Issue 4, 2004. В этой статье рассматриваются перспективы развития искусственного интеллекта и приведен прогноз, согласно которому к 2007 году товарооборот на мировом рынке в пяти основных областях применения искусственного интеллекта (экспертные системы, сети доверия, системы поддержки принятия решений, нейронные сети и агенты) достигнет 21 миллиарда долларов. Этот бесплатный оперативный коммерческий журнал издается с целью предоставления административным кругам информации о новейших тенденциях в управлении знаниями. Статьи этого журнала, весьма доступные для восприятия, можно найти по адресу `http://www.kmworld.com/publications/magazine/index.cfm?action=readarticle&article_id=1504&publication_id=1`.
- Alexander Nareyek, *AI in Computer Games*. ACMQueue.com. ACM Queue Vol. 1, No.10, February 2004. Качественное обсуждение проблем практического использования конечных автоматов, деревьев решений и других методов искусственного интеллекта в играх. Способствует лучшему пониманию всех проблем искусственного интеллекта в целом (`http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=117&page=1`).
- *AI on the Web* (`http://www.cs.berkeley.edu/~russell/ai.html`). Огромная коллекция ссылок на все области искусственного интеллекта, которая включает больше чем 850 ссылок, составленная Стюартом Расселлом (Stuart Russell). На данном узле можно найти не только эти, но и другие ресурсы, взятые из книги *Artificial Intelligence: A Modern Approach (Second Edition) by Stuart Russell and Peter Norvig*, 2002[1]. Настоятельно рекомендуется как хороший вводный учебник по искусственному интеллекту с большим количеством других вспомогательных материалов, которые можно найти по адресу `http://aima.cs.berkeley.edu/`.
- *Google* (`http://directory.google.com/Top/Computers/Artificial_Intelligence/`). Каталог ссылок *Directory of AI* узла *Google*, предназначенный для просмотра и поиска.

[1] *Стюарт Рассел, Питер Норвиг (2005).* Искусственный интеллект: современный подход, 2-е изд., *Издательский дом "Вильямс"*.

- *Yahoo* (`http://dir.yahoo.com/Science/Computer_Science/Artificial_Intelligence/`). Каталог ссылок *Directory of AI* узла *Yahoo*, предназначенный для просмотра и поиска.
- *The AI Center* (`http://www.ai-center.com/sitemap.html`). Хороший стартовый портал, с которого можно начинать освоение тематики искусственного интеллекта. В разделе Links этого узла приведены многочисленные ссылки на информационные ресурсы (`http://www.ai-center.com/links/`).
- *Dr. Mark Humphrey's Homepage* (`http://www.compapp.dcu.ie/%7Ehumphrys/index.html`). Узел, на котором приведено очаровательное описание, позволяющее узнать, есть ли в вашей родословной особы королевской крови! Но упоминание об этом узле приведено в настоящем приложении потому, что на его начальной странице можно найти многочисленные ссылки на материалы по обучению, искусственному интеллекту и на многие другие ресурсы.
- *AI Links* (`http://www.compapp.dcu.ie/%7Ehumphrys/ai.links.html`). Огромный список ссылок, составленный доктором Хэмфри, охватывающий многие общие темы по искусственному интеллекту, по которым и сгруппированы ссылки.
- *Teaching Links* (`http://www.compapp.dcu.ie/%7Ehumphrys/teaching.html`). Узел доктора Хэмфри с гигантским объемом оперативных ресурсов по искусственному интеллекту. С этим узлом действительно стоит ознакомиться преподавателям, которые только начинают вести курс по искусственному интеллекту, или исследователям, стремящимся найти новые идеи для реализации.
- *Search Engines Collection* (`http://www.compapp.dcu.ie/%7Ehumphrys/computers.internet.html`). Предлагаемая доктором Хэмфри превосходная коллекция машин поиска, с помощью которых можно легко провести поиск не только на данном узле, но и во всем мире.
- *Royal Descent Search* (`http://humphrysfamilytree.com/famous.descents.html`). Поддерживаемый доктором Хэмфри действительно эффективный узел поиска королевских корней. Например, с его помощью можно просмотреть генеалогическое дерево Уолта Диснея, который ведет свое происхождение от короля Англии Генрих I по прозвищу Боклерк, от фр. "имеющий хорошее образование" (1070–1135). После этого становится ясно, почему символ Диснейленда — замок (`http://members.aol.com/dwidad/disney.html#wd`).
- *Virtual Environments* (`http://www.planet9.com/demos_downloads.html`). На этом узле находится больше 40 очень точных трехмерных мо-

делей городов, созданных компанией Planet 9 Studios. Эти наборы данных могут использоваться в самых разных целях, включая окончательную проверку автономного робота, способного успешно двигаться по городским улицам. Изображения, которые создаются с помощью этих наборов данных, весьма напоминают трехмерный город, где происходит действие в серии фильмов *Matrix*.

- *Online Robotics and Cameras* (`http://ford.ieor.berkeley.edu/ir/`). Оперативный архив 50 проектов по телемеханике и робототехнике.
- *Hyperlinked Papers On AI* (`http://www.cs.bham.ac.uk/~wbl/ftp/biblio/gp-bibliography.ref`). Огромная коллекция ссылок на работы по искусственному интеллекту, составленная Стюартом Рейнолдсом (Stuart Reynolds).
- *Reinforcement Learning: An Introduction* (`http://www-anw.cs.umass.edu/%7Erich/book/the-book.html`). Книга Ричарда С. Саттона (Richard S. Sutton) и Эндрю Г. Барто (Andrew G. Barto), предоставляемая в оперативном режиме (эта книга была также выпущена издательством MIT Press в 1998 году и имеется в продаже), в который речь идет о том, как проводить обучение программных агентов и роботов, чтобы они могли реагировать на воздействия окружающей среды. В книге обсуждаются многие методы, включая марковский процесс принятия решений, планирования и обучение, а также приведены некоторые сложные практические примеры; безусловно, с этой книгой следует ознакомиться.
- *Complexity Papers Online* (`http://www.calresco.org/papers.htm`). Огромная коллекция статей по многим темам, включая искусственный интеллект, теорию сложности, хаос и сотни других, в том числе классических статей первых основателей такого научного направления, как искусственный интеллект.
- *EvoWeb* (`http://evonet.lri.fr/`). Обзор всех европейских Web-узлов, посвященных любым направлениям эволюционных вычислений, включая генетические алгоритмы, робототехнику и интеллектуальные системы; представлены многочисленные статьи по программному обеспечению, книги и ссылки. Кроме того, на этом узле можно ознакомиться с последними новостями и технологическими достижениями. В Европейском союзе принята программа предоставления правительственных грантов небольшим деловым предприятиям для создания Web-узлов, способных обеспечить глобальный маркетинг выпускаемых ими товаров и предоставляемых услуг. Создание узла *EvoWeb* свидетельствует о стремлении представителей руководства Европейского союза использовать наиболее развитые технологии искусственного интеллекта, такие как эволюционные вычисления, на что

указывает применение термина “evolutionary” (или сокращенно “evo”) в названии узла.

- В Европе был также создан ряд других важных Web-узлов, предназначенных для стимуляции развития конкретных областей искусственного интеллекта. Все эти узлы связаны с указанным выше главным узлом EvoWeb.

EvoBIO. Web-узел по биоинформатике, которой посвящен проблемам разработки алгоритмов на основе эволюционных вычислений, позволяющих решать важные задачи в молекулярной биологии, геномике и генетике.

EvoELEC. Европейский Web-узел по эволюционной электронике, на котором рассматриваются способные к эволюции цифровые и аналоговые аппаратные средства, созданные по образцам живой природы вычислительные средства, эволюционирующие машины и эволюционная электроника.

EvoGP. Web-узел по генетическому программированию, на котором рассматриваются трудные задачи проектирования, распознавания образов и управления, а также приложения, применяемые в финансовом анализе скрытых закономерностей в данных, обработке сигналов и изображений, биоинформатике, техническом проектировании, музыке и искусстве.

EvoIASP. Европейский Web-узел по анализу изображений и обработке сигналов. Доказано, что эволюционные алгоритмы (Evolutionary Algorithms — EA) являются эффективным инструментальным средством автоматического проектирования и оптимизации систем. Опубликованы сотни статей, посвященных вопросам применения эволюционных алгоритмов для анализа изображений и обработки сигналов.

EvoROB. Европейский Web-узел по эволюционной робототехники, в основном посвященный применению эволюционных алгоритмов для автоматического проектирования автономных роботов. На нем рассматривается восходящий подход к робототехническому обучению и синтезу контроллеров путем взаимодействия со средой, а не в результате проектирования людьми.

EvoSTIM. Европейский Web-узел, посвященный планированию и составлению расписаний с использованием эволюционных алгоритмов, показавших себя очень успешными.

- Tom Lloyd, *When swarm intelligence beats brainpower* (`http://money.telegraph.co.uk/money/main.jhtml?xml=%2Fmoney%2F2001%2F06%2F06%2Fcnantz06.xml`). Статья об инструментальных средствах преодоления сложности, таких как средства коллективного интеллекта, позволяющих находить решения задач, которые являются настолько сложными (наподобие задачи коммивояжера), что для их решения в масштабах, превосходящих определенные размеры задачи, не существует эффективных алгоритмов. После того как количество городов, рассматриваемых в условиях задачи, становится слишком большим, чрезвычайно замедляются

даже вычисления с использованием Grid-технологии, поскольку количество возможных маршрутов для N городов определяется значением $(N-1)!$, т.е. стремительно возрастает. В методах коллективного интеллекта применяются алгоритмы, подобные алгоритму колонии муравьев (Evolutionary Ant, который еще не раз будет упоминаться в данном приложении), для получения качественного решения за приемлемое время для больших значений N. Данная статья представляет собой отличный обзор состояния того, как технологии коллективного интеллекта успешно используются в бизнесе для решения сложных задач.

- *SwarmWiki* (`http://wiki.swarm.org/wiki/Main_Page`). Важный портал, на котором представлены информация, программное обеспечение и многочисленные ссылки, полностью посвященные коллективному интеллекту и его применению для решения задач к искусственного интеллекта.
- Предоставляемые межуниверситетской научно-исследовательской лабораторией Microsoft (Microsoft Multi-University Research Laboratory) содержательные лекции известных специалистов, с которыми можно ознакомиться в оперативном режиме (`http://murl.microsoft.com/ContentMap.asp`).
- Предоставляемая в оперативном режиме превосходная серия слайдов для преподавания предметов по искусственному интеллекту и экспертным системам с точки зрения PROLOG, подготовленная Элисон Коуси (Alison Cawsey), которая охватывает большинство тем из теоретического раздела настоящей книги. Несмотря на конкуренцию со стороны более новых языков (таких как Python), PROLOG и Lisp все еще остаются стандартными языками, применяемыми в учебниках по искусственному интеллекту (`http://www.cee.hw.ac.uk/~alison/ai3/`).
- Огромная коллекция слайдов по многим темам искусственного интеллекта, подготовленная Эроном Сломеном (Aaron Sloman), вполне заслуживающая ознакомления (`http://www.cs.bham.ac.uk/%7Eaxs/misc/talks/`).
- Нейронные сети применяются не только в исследовательских целях, но и на практике. В частности, с их помощью были сэкономлены миллионы долларов благодаря раскрытию случаев мошенничества с дебетовыми кредитными карточками в интересах организаций по обслуживанию кредитных союзов (Credit Union Service Organizations — CUSO). Представители крупнейшей в США организации по обслуживанию кредитных союзов PSCU Financial Services объявили, что с использованием нейронных сетей в 2002 году было расследовано 74 тысячи случаев возможных мошенничеств с карточками и возвращено 12,8 миллиона долларов, или 99,9% восстанови-

мых потерь (http://www.cuna.org/newsnow/archive/list.php?date=041003).

- Конкретным инструментальным средством, используемым организацией PSCU Financial Services, является инструментальное средство нейронных сетей Falcon, с помощью которого мошенничества с кредитными карточками обнаруживаются в реальном времени (http://www.pscufs.com/falcon.htm). Если вам когда-либо приходилось стоять у расчетного узла, удивляясь, почему для списания денег с кредитной карточки требуется так много времени, то, по-видимому, в течение этого периода ожидания происходила проверка с помощью нейронной сети в целях обнаружения подозрительных действий. При совершении необычно крупных покупок может быть даже отказано в заключении торговой сделки; в таких случаях приходится звонить в компанию, обслуживающую дебетовую кредитную карточку, и доказывать, что сделку действительно совершаете вы. (К счастью, представители этих компаний не звонят домой, чтобы узнать о подробностях сделки у супруга или супруги, поскольку иначе у многих были бы серьезные неприятности.)
- Фактически организация Credit Union National Association требует от своих клиентов (обществ взаимного кредитования), чтобы они использовали технологию нейронных сетей для сокращения случаев мошенничества и снижения издержек. С другими деловыми приложениями можно ознакомиться на оперативном узле *BusinessWeek*, оснащенном хорошей машиной поиска, которая позволяет легко находить информацию о новейших приложениях искусственного интеллекта (http://www.businessweek.com/bw50/content/mar2003/a3826072.htm).
- Компания CorMac Technologies обладает всесторонним набором коммерческих инструментальных средств искусственного интеллекта и позволяет ознакомиться с примерами успешного применения искусственного интеллекта, включая следующие:
  - диагностика онкозаболеваний с использованием нейронной сети;
  - диагностика онкозаболеваний с использованием индуктивного извлечения правил;
  - анализ ДНК митохондрий с помощью индуктивного извлечения правил;
  - анализ ДНК митохондрий с помощью идентификации кластеров.

  Web-узел этой организации находится по адресу http://www.cormactech.com. Пробную версию программного обеспечения искусственной нейронной сети можно получить по адресу: http://

cormactech.com/neunet/index.html, а доступ к другим программным продуктам предоставляется на начальной странице этого узла. Многочисленные законченные примеры со всеми наборами данных находятся по адресу (http://cormactech.com/neunet/download.html#DATA).

- Многочисленные интересные демонстрационные версии и проекты по искусственному интеллекту (http://www.cs.wisc.edu/~dyer/cs540/demos.html).
- Ссылки по тематике искусственной жизни (http://www.alcyone.com/max/links/alife.html).
- Простое объяснение нюансов нечеткой логики, предоставляемое в оперативном режиме, под названием *Fuzzy logic for Just Plain Folks* (http://www.fuzzy-logic.com/).
- Предоставляемые в оперативном режиме книга по нечеткой логике и программное обеспечение нечеткой экспертной системы FLOPS (http://members.aol.com/wsiler/).
- Преимущества генетического программирования (http://murl.microsoft.com/LectureDetails.asp?785).
- *Genetics and Evolutionary Algorithm Archive*. Огромный информационный ресурс со ссылками на узлы, посвященные генетическим и эволюционным алгоритмам и приложениям (http://www.aic.nrl.navy.mil/galist/).
- *GAUL*™ (Genetic Algorithm Utility Library). Ведущая библиотека программ с открытым исходным кодом, посвященная эволюционным вычислениям (http://sourceforge.net/forum/forum.php?forum_id=386667).
- Компания PMSI поддерживает превосходный узел со множеством учебников и инструментальных средств, которые могут быть загружены бесплатно вместе с рабочими примерами (http://www.pmsi.fr/home-gb.htm). Ниже перечислены приложения по генетическим алгоритмам, которые можно найти по адресу http://www.pmsi.fr/gafxmpa.htm.
  - FOOD. Обучение роботов поиску пищи.
  - GAFUNC. Различные задачи функциональной оптимизации.
  - TSPSA. Нейронная сеть, предназначенная для решения задачи коммивояжера, действующая по принципу моделирования отжига.
- Перечисленные ниже демонстрационные версии приложений, действующих на основе нейронных сетей, можно найти по адресу http://www.pmsi.fr/sxcxmpa.htm.

  - Parabola. Практическое использование нейронных сетей.
  - FUNCTION. Приложение, иллюстрирующее способность нейронных сетей к обучению в целях моделирования функций.
  - OCR. Простой пример приложения, предназначенного для оптического распознавания символов.
  - VEHICLE.EXE. Приложение, предназначенное для управления транспортным средством, моделирования поведения и инверсии модели, при использовании которого система управления транспортным средством (или роботом) пытается найти путь к месту назначения из некоторой произвольной начальной точки.

- Средства обеспечения эволюционного роста нейронной сети с помощью генетических алгоритмов и моделируемого отжига, а также много интересных примеров предоставляются по адресу `http://www.pmsi.fr/grnxmpa.htm`.
- Средства обеспечения индуктивного роста дерева решений с использованием классических методов ID3 и C4.5, позволяющие автоматически формировать деревья решений, приведены по адресу `http://www.pmsi.fr/padxmpa.htm`.
- Эволюционное программное обеспечение моделирования поведения колонии муравьев из книги *Marco Dorigo and Thomas Stuetzle, Ant Colony Optimization, MIT Press, 2004*. Одно из приложений этого программного обеспечения применяется для решения важной задачи коммивояжера. Это программное обеспечение можно получить по адресу `http://iridia.ulb.ac.be/~mdorigo/ACO/aco-code/public-software.html`.
- *The History of Computer Science* (`http://www.eingang.org/Lecture/toc.html`). Многочисленные ссылки на информационные ресурсы, с помощью которых можно ознакомиться с тем, как в течение многих лет происходило развитие компьютерных наук; в этих ресурсах отражены многие темы и описаны такие важные исторические личности, как Ален Тьюринг.
- *The Turing Test Page* (`http://cogsci.ucsd.edu/%7Easaygin/tt/ttest.html#people`). Хороший обзорный узел, посвященный Алену Тьюрингу, с описанием его биографии. На этом узле представлены ссылки на информационные ресурсы, посвященные тесту Тьюринга, указана дополнительная литература, приведено описание истории борьбы за главный приз Лебнера в 100 тысяч долларов, учрежденный доктором Хью Лебнером в 1991 году для присуждения автору первой компьютерной программы, которая пройдет неограниченный тест Тьюринга. На этом узле предоставляется возможность пообщаться с некоторыми программами и ознакомиться

со ссылками на работы специалистов, занимающихся интересными исследованиями по философии разума, интеллектуальным роботам и по многим другим темам, подходящим для презентации в учебном процессе и выполнения сложных курсовых заданий.

- *The Alan Turing Home Page* (`http://www.turing.org.uk/turing/`). На этом узле можно получить подробные сведения о жизни и деятельности Алена Тьюринга. На нем представлены основные вехи жизни ученого, полная библиография работ Тьюринга, доступная для ознакомления в оперативном режиме биография Тьюринга, архив, фотографии и профессиональные статьи.
- *The Turing Test Is Not a Trick* (`http://www.ecs.soton.ac.uk/%7Eharnad/Papers/Harnad/harnad92.turing.html`). На этом узле можно ознакомиться с аргументами в пользу того, что тест Тьюринга, позволяющий отличить компьютер от человека, — это действительно научный критерий, а способ времяпрепровождения, как утверждают некоторые оппоненты. С другой стороны, можно ли назвать лишь забавой современные состязания, подобное такому телевизионному игровому шоу, как *Jeopardy* или *Survivor*, в котором люди выигрывают свыше миллиона долларов? Утверждения такого рода похожи на те, согласно которым бейсбол высшей лиги — это только развлечение, притом что некоторые бейсболисты зарабатывают по 10 миллионов долларов в год.
- *The 2003 Loebner Prize Winner*. Программа Jabberwock, хотя и не прошла полную неограниченную версию теста Тьюринга, принесла в 2003 году своему создателю 2 тысячи долларов. Безусловно, это не 100 тысяч долларов, но много ли найдется программ, приносящих такие призы (`http://www.surrey.ac.uk/dwrc/loebner/`).
- *Botspot* (`http://www.botspot.com`). На этом Web-узле можно ознакомиться с некоторыми из наиболее широко применяемых чаттерботов. *Чаттерботы* — это программы, предназначенные для ведения с ними беседы, как и с людьми. Некоторые из них создавались с серьезными намерениями и предназначались для прохождения теста Тьюринга, а другие служат для всевозможных развлечений! Сложность таких программ постоянно возрастает, и хотя с их помощью еще не удалось выиграть приз Лебнера, иногда у знакомых с ними рядовых сотрудников компаний складывается впечатление, что именно с помощью таких программ начальство формулирует свои вопросы о том, как продвигаются порученные им проекты. Конечным итогом развития этого направления является замена людей в телефонных центрах обработки заказов. Такая замена привела бы к еще большей экономии, чем глобальный аутсорсинг, поскольку даже при использовании аутсорсин-

га приходится платить людям заработную плату, пусть даже значительно меньшую.

- Проект LISA — это платформа для разработки интеллектуальных программных агентов на языке Lisp. Это — система на основе продукционных правил, реализованная с применением системы CLOS (Common Lisp Object System — общая объектная система Lisp), которая испытала серьезное влияние языков CLIPS и Jess. В наши дни интеллектуальные агенты используются во многих приложениях (`http://lisa.sourceforge.net/`).
- *New Directions Magazine* (`http://www.newscientist.com/hottopics/quantum/`). Журнал, в котором публикуются многочисленные статьи и ссылки по квантовым компьютерам и другим темам, оформленный как страница Web-узла журнала *New Scientist*. С журнала *New Directions Magazine* вполне можно начинать изучение любой темы, поскольку его статьи написана на вводном уровне. Кроме того, начальная страница этого журнала также хорошо подходит для ознакомления с новейшей информацией о последних достижениях в науке и вычислительной технике. На данном узле имеется также обширный раздел Jobs с предложениями о трудоустройстве, хотя большинство этих предложений касаются рабочих мест в Европе, поскольку журнал *New Scientist* публикуется в Англии.
- *Quantum Information Science* (`http://wtec.org/qis/Awschalom.PDF`). Краткий курс по квантовым методам, изданный в австрийском городе Инсбруке. Включает в себя множество современных презентаций, посвященных квантовым вычислениям, квантовой криптографии и многим другим темам. Некоторые презентации настолько великолепны, что авторам еще не приходилось видеть подобных, поэтому они вполне заслуживают внимания.
- *Committee on Standards for Artificial Intelligence Interfaces* (`http://www.igda.org/ai/`). Это — подгруппа международной ассоциации разработчиков игр, *International Game Developers Association*. Представители этой подгруппы формулируют свою задачу следующим образом: "... стимулирование разработки интерфейсов для основных функциональных средств искусственного интеллекта, обеспечивающих повторное применение кода и аутсорсинг, а также освобождение программистов от необходимости реализовывать программные средства искусственного интеллекта на низком уровне, чтобы они могли направить больше ресурсов на решение сложных задач искусственного интеллекта". С этого узла можно начинать свои исследования тем, кто собирается приобщиться к индустрии игр или ищет способы стандартизации разработок по искусственному интеллекту для своей компании, чтобы "повторно не изобретать колесо" после развертывания каждого нового проекта, требующего применения искусственного интеллекта.

- Grid-технология вычислений позволяет эксплуатировать в фоновом режиме неиспользуемые мощности многочисленных компьютеров, подключенных к локальной сети или Интернету. Эта технология применяется для эмуляции беспредельно мощного виртуального Grid-компьютера, способного проявлять невероятные возможности, ограничиваемые только пропускной способностью и количеством компьютеров; теоретически в сеть Grid можно включить все компьютеры в мире. Grid-технология активно используется такими компаниями, как IBM (`http://www-1.ibm.com/grid/`).
- В настоящее время появился новый класс провайдеров Grid, которые предоставляют возможность пользоваться вычислительными мощностями с производительностью настоящего суперкомпьютера, не взимают плату за машинное время и не требуют приобретения такого суперкомпьютера. В действительности сеть Grid может оказаться гораздо более быстродействующей, чем любой отдельно взятый суперкомпьютер; производительность зависит лишь от того, насколько успешно может быть распределена какая-то конкретная задача по всем соединенным в сеть компьютерам. В частности, для проведения биологических и медицинских исследований использовались такие сети Grid специального назначения, как BioGrid, поскольку при совместной эксплуатации программного обеспечения задачи анализа решаются гораздо более эффективно после их распределения по всем компьютерам, хотя в этом не всегда есть необходимость. Кроме того, Grid-технология может использоваться для исследования новых методов и алгоритмов искусственного интеллекта, которые до сих пор невозможно было осуществить из-за ограниченной мощи компьютеров. Ниже указаны некоторые источники информации и приведены ссылки.
  - Madhu Chetty and Rajkumar Buyya. *Weaving Computational Grids: How Analogous are they with Electrical Grids*, Computing in Science & Engineering, p. 61–71, August 2002.
  - Sergio Rajsbaum. *Distributed Computing Research Issues in Grid Computing*, ACM SIGACT News Distributed Computing Column 8, p. 50–70, July 2002.
- Дополнительные ссылки на информационные ресурсы, касающиеся вычислений с помощью Grid-технологии:
  - `http://www.gridcomputing.com`.
  - `http://www.eu-datagrid.org`.
  - `http://www.globus.org/`.
  - `http://www.bioinformaticsworld.info/feature3b.html`.
  - `http://www.gpds.org/`.

- http://www-1.ibm.com/grid.
- http://www.ncbiogrid.org/.
- http://www.sbml.org.
- http://www.biogrid.jp.
- http://biocomp.ece.utk.edu.
- http://www.biogrid.icm.edu.pl.
- http://www.grid.org.
- http://www.ncbi.nih.gov/BLAST/.
- http://gridcafe.web.cern.ch.
- http://www.nbirn.net.

Приложения экспертных систем

- Информация о приложениях экспертных систем, статьях, программном обеспечении и компаниях. Огромный список, предоставляемый оперативным коммерческим журналом PCAI (http://www.pcai.com/web/ai_info/expert_systems.html).
- Медицинские экспертные системы, огромный список (http://www.computer.privateweb.at/judith/name_3.htm).
- Искусственный интеллект и экспертные системы, большое количество ссылок (http://www.dmaier.net/teaching/cis386/links.htm).
- Экспертные системы сельскохозяйственного назначения, большое количество ссылок (http://potato.claes.sci.eg/Home/wes.htm).
- Слайды краткой ознакомительной презентации CLIPS, предоставляемые Питером Джексоном (Peter Jackson), автором книги *Introduction to Expert Systems*, в которой рассматриваются CLIPS и другие экспертные системы (http://www.geocities.com/jacksonpe/clips/clips.htm).
- Judith Lamont, *Innovative Applications Make Government More Responsive*, *KMWorld Magazine*, Vol. 12, Issue 6. Хорошее вводное описание, позволяющее ознакомиться с тем, как различные правительственные агентства используют экспертные системы для улучшения реагирования на вопросы общественности. *KMWorld Magazine* — это оперативный коммерческий журнал, в котором публикуются многие интересные статьи об управлении знаниями и применении интеллектуальных систем в бизнесе и государственной управлении. Вполне заслуживает ознакомления. На узле журнала *KMWorld Magazine* предусмотрена возможность просматривать статьи по интересующей тематике или находить нужные статьи с помощью машины поиска (http://www.kmworld.com/publications/magazine/index.cfm?action=readarticle&Article_ID=1541&Publication_ID=93).

- Оперативная медицинская диагностическая экспертная система (`http://easydiagnosis.com/`). Ее разработчики предоставляют возможность бесплатно пройти диагностику, с помощью которой можно, например, узнать, достаточно ли в вашей пище растительных волокон.
- Министерство труда США предоставляет предпринимателям целый ряд экспертных систем по трудовому праву, которые позволяют решать проблемы соблюдения правовых норм, следить за соблюдением техники безопасности на рабочем месте, уточнять права и обязанности. Эти экспертные системы объединены под общим названием *elaws Advisors* и предоставляются на узле `http://www.dol.gov/elaws/see_adv.asp?Subset=ID>0`.
- *eTools and Electronic Products for Compliance Assistance* (`http://www.osha.gov/dts/osta/oshasoft/eTools`). Экспертные системы, предоставляемые Управлением по гигиене труда и профессиональным заболеваниям (Occupational Safety and Health Administration — OSHA).
- *OSHA Emergency Action Plan* (`http://www.osha-slc.gov/SLTC/etools/evacuation/experts.html`). Правительственная программа, направленная на предоставление компаниям экспертных систем, предназначенных для определения того, нуждается ли компания в чрезвычайном плане действий, и в случае положительного ответа позволяющая создать такой план.
- *National Science Foundation — Environmental Monitoring and Measurement Advisor* (`http://www.emma-expertsystem.com/`). Консультационная система по мониторингу и измерению характеристик окружающей среды.
- *U.S. Geological Survey — Decision Support Systems for Trumpeter Swan Management* (`http://swan.msu.montana.edu/cygnet/`).
- *Seven "Online Consultants"* (`http://www.nre.vic.gov.au/web/root/Domino/Target10/T10Frame.nsf`). Семь "оперативных консультантов" — экспертные системы, предоставляемые Министерством природных ресурсов и окружающей среды (Department of Natural Resources and Environment) Австралии для молочных ферм.
- *BusinessLaw.gov* (`http://www.businesslaw.gov/tools/business-wizards.htm`). Узел, на котором предоставляется огромный объем информации и целый ряд экспертных систем для деловых предприятий.
- *Shyster* (`http://cs.anu.edu.au/software/shyster/`). Юридическая экспертная система, действующая на основе прецедентов, которая разработана Джеймсом Попплом (James Popple). Подробные сведения о проектировании, реализации, эксплуатации и тестировании системы Shyster приведены в книге James Popple, *A Pragmatic Legal Expert System, Applied Legal Philosophy Series*, Dartmouth, Aldershot, 1996.

- *Acquire*®. Командный интерпретатор системы приобретения знаний и экспертной системы. Испытательная версия этого программного обеспечения предоставляется на Web-узле `http://www.aiinc.ca`. Многочисленные примеры приведены по адресу `http://www.aiinc.ca/demos/index.html`.
- Система управления знаниями XpertRule Knowledge Builder и инструментальное средство для анализа скрытых закономерностей в данных XpertRule Miner предоставляются компанией Attar Software Limited. Список экспертных систем, созданных этой компанией и ее филиалом с помощью указанных программ, представлен по адресу `http://www.attar.com/deploy/demos.htm`.
- Информацию о приложениях, разработанных компанией Attar Software Limited (в том числе перечисленных ниже), можно найти по адресу `http://www.intellicrafters.com/cases.htm`.
  - *Rockwell Aerospace and NASA*. Экспертная система, разработанная для NASA компанией Rockwell International в целях получения консультаций по проведению работ, связанных с предотвращением и ликвидацией загрязнений.
  - *Channel 4 TVResource Optimization*. Система на основе генетических алгоритмов, разработанная с использованием программы XpertRule(r), которая предназначена для планирования последовательности рекламных пауз.
  - *Tokyo Nissan*. Интеллектуальная система Car Selection System (Система выбора автомобиля), разработанная с использованием программы XpertRule®.
  - *Australian Taxation Office*. Экспертная система по налогообложению.
  - *Work and Income New Zealand*. Программа XpertRule® Expert Calculator, которой пользуются 3000 сотрудников для решения вопросов подтверждения прав на получение пособий и льгот, а также расчета сумм пособий и льгот.
  - *Hosokawa MicronData*. Система предоставления консультаций по обработке порошкообразных и сыпучих веществ.
  - *GE Capital Global Consumer Finance*. Анализ скрытых закономерностей в данных, предназначенных для финансового использования.
  - *The Gas Research and Technology Centre*. Применение средств анализа скрытых закономерностей в данных для уменьшения стоимости бурения с продувкой газом в компании BG.

- *Department of Industry and Fisheries*. Экспертные системы, предоставляемые правительственным агентством Тасмании для помощи фермерам.
- *Department of Industry and Fisheries*. Система Threatened Fauna Adviser, предоставляемая правительством Тасмании лесохозяйственным компаниям и подрядчикам для получения консультаций, которые касаются фауны, находящейся под угрозой.
- Сеть магазинов Misselbrook and Weston. Система, основанная на знаниях, которая предназначена для обнаружения случаев мошенничества в магазинах.
- *Hibernian Life & Pensions* (Ирландия). Система анализа скрытых закономерностей в данных, основанная на знаниях, позволяющая обеспечить оптимальное перепроектирование бизнес-процесса в страховых и пенсионных компаниях.
- *United Distillers*. Система Resource Optimization, созданная с использованием программы XpertRule®, которая позволяет оптимизировать перевозки бочек виски, требуемых для производства качественных сортов напитков в компании United Distillers, Шотландия.
- *ICI and Carlsberg Tetley*. Система анализа скрытых закономерностей в данных, используемая на электростанции Thornton компании ICI и на пивоваренном заводе Tetley компании Carlsberg для уменьшения расхода энергии.
- *Elf-Atochem North America*. Экспертная система Rilsan® Advisor, предоставляющая консультации по характеристикам товаров для технического персонала, занимающегося сбытом и маркетингом.
- *The Leeds Building Society*. Система анализа скрытых закономерностей в данных, предназначенная для выявления задолженностей по закладным в компании Leeds Building Society (которая теперь входит в состав компании HBOS).
- *Swedish Marines*. Экспертная система, которая используется в Министерстве обороны Швеции для определения пригодности к службе морских пехотинцев.
- *Heureka*. Экспертная система, используемая в Шведском управлении по промышленному развитию, для оценки перспективности предложений по выпуску товаров.
- *VAT in Sweden*. Система предоставления консультаций и обучения, которая касается применения налога на добавленную стоимость (Value Added Tax — VAT) в Швеции.

  - *Meiji Mutual Life Insurance*. Система, основанная на знаниях, применяемая для выбора плана страхования в компании Meiji Mutual Life Insurance.
  - *Ebara Manufacturing*. Система, позволяющая японской компании координировать работу 3 тысяч ветровых установок и водяных насосов с учетом потребностей заказчиков.
  - *Traversum AB*. Система анализа скрытых закономерностей в данных, основанная на знаниях, которая предназначена для предоставления консультаций по акциям и ценным бумагам, которые находятся в обращении в Европе.
  - *Sun Direct Insurance*. Система, основанная на знаниях, которая предназначена для определения страховых котировок, применяемых при страховании домов и автомобилей.
- *LPA (Logic Programming Associates)*. На узле LPA предоставляются программные инструменты, основанные на расширенной версии языка PROLOG, называемой PROLOG++. Для приложений нечеткой логики и приложений PROLOG, эксплуатируемых в среде Windows, разработано еще одно инструментальное средство — Flint. Предоставляются также бесплатная версия PROLOG и интересные демонстрационные версии (`http://www.lpa.co.uk/pws_dem.htm`).
- На узле компании Exsys, Inc. имеется большой список современных экспертных систем, разработанных с помощью коммерческого инструментального средства создания экспертных систем этой компании. С этого узла можно также загрузить демонстрационные версии (`http://www.exsys.com/case2.html`).

Глава 2

- Ulf Nilsson and Jan Mahuszynski, *Logic Programming and PROLOG, 2nd Edition*, 2000. Книга, предоставляемая бесплатно для чтения в оперативном режиме, которую можно также загрузить с указанного узла. Язык PROLOG предназначен для обеспечения обратного логического вывода и наряду с языком LISP относится к числу классических языков искусственного интеллекта, которые все еще находят свое применение (`http://www.ida.liu.se/~ulfni/lpp/`). На узле Нилссона предоставляется доступ ко многим дополнительным информационным ресурсам, применяемым при обучении (`http://www.ida.liu.se/~ulfni/teaching.shtml`). А в указанной выше книге Нилссона приведено гораздо больше сведений о представлении знаний, логическом программировании, а также дано больше примеров, чем в главах 2 и 3 настоящей книги.

- Для представления знаний и формирования запросов разработаны специальные языки, такие как KQML (`http://www.cs.umbc.edu/kqml/`).
- На Web-узле Дейва Хеннея (Dave Hannay) предоставляется возможность ознакомиться с конечными автоматами и синтаксическими анализаторами других типов, применяемыми для распознавания лексем, а также приведены другие учебные материалы (`http://scoter3.union.edu/~hannayd/csc140/simulators/`).
- Дуг Ленат (Doug Lenat) опубликовал текст лекции по здравому смыслу и искусственному интеллекту, предоставляемый в оперативном режиме (`http://murl.microsoft.com/LectureDetails.asp?1032`). Дуг приобрел известность благодаря своему участию в разработке технологии OpenCyc — версии технологии Cyc с открытым исходным кодом (`http://www.cyc.com/cyc/technology/whatiscyc`). В состав этой технологии входят крупнейшая в мире и наиболее полная база общих знаний и машина формирования рассуждений на основе здравого смысла. Дуг стал также знаменитым благодаря новаторской работе в области автоматизированного совершения открытий в математике с применением программ AM (Automated Mathematician — автоматизированный математик) и Eurisko. Об этом уже шла речь в данной книге.
- *Formal Methods* (`http://archive.comlab.ox.ac.uk/comp/formal-methods.html`). Важный узел со многими информационными ресурсами, относящимися к логике.
- Программное обеспечение для формирования правил по методу обучения с помощью индукции. На этом узле компания CorMac Technologies, Inc. предоставляет программу VisiRex 2.0 для Windows, в том числе испытательную версию. Кроме того, предоставляется возможность воспользоваться двумя другими методами — классическим методом обратного распространения и нейронной сетью SFAM. В комплект программного обеспечения входят наглядные примеры (`http://cormactech.com/visirex/faq.html`).

Информационные ресурсы и программное обеспечение анализа скрытых закономерностей в данных

- Компания Complexica® предоставляет примеры успешного использования средств анализа скрытых закономерностей в данных, а также инструментальные средства и службы анализа скрытых закономерностей в данных (`http://internet.cybermesa.com/~rfrye/complexica/dm_em.htm`).

- Классическим программным обеспечением для индуктивного машинного обучения стала программа ID3 Росса Куинлана (Ross Quinlan), которая в дальнейшем была усовершенствована и выпущена под названием C4.5. Это программное обеспечение наряду с различными статьями и слайдами можно получить на личной Web-странице Росса по адресу `http://www.cse.unsw.edu.au/~quinlan/`. Новейшие инструментальные средства анализа скрытых закономерностей в данных (See3 и C5) обладают значительными преимуществами по сравнению с программой C4.5 и предоставляются на коммерческой основе компанией, которую также организовал Росс (`http://www.rulequest.com/`).
- Огромная коллекция программ для анализа скрытых закономерностей в данных, формирования деревьев решений, анализа данных с использованием нейронных сетей, а также программ по статистике и нечетким технологиям представлена по следующему адресу: `http://www.the-data-mine.com/bin/view/Software/WebIndex`.
- Крупная коллекция баз данных, определения теорий проблемных областей и генераторов данных, предназначенных для проверки алгоритмов машинного обучения находится по адресу (`http://www.ics.uci.edu/~mlearn/MLRepository.html`).
- Большая библиотека классов C++, с помощью которой могут быть реализованы общие алгоритмы машинного обучения, предназначенные для разработки инструментальных средств анализа скрытых закономерностей в данных и визуализации (`http://www.sgi.com/tech/mlc/`). Поскольку код библиотеки написан на языке C++, с ее помощью можно легко написать функции CLIPS для решения указанных задач, а затем повторно откомпилировать интерпретатор CLIPS для обеспечения оптимальной производительности. Полный исходный код системы CLIPS на языке C++ CLIPS находится на компакт-диске, прилагаемом к настоящей книге.
- Коллекция, состоящая из очень крупных наборов данных, относящихся к различным областям, которая предназначена для проверки инструментальных средств анализа скрытых закономерностей в данных. Эта коллекция может оказаться полезной для тех, кто разрабатывает собственную версию инструментальных средств интеллектуального анализа данных на языке CLIPS, как указано выше в данном приложении (`http://kdd.ics.uci.edu/`).
- *alphaWorks* (`http://www.alphaworks.ibm.com/Home/`). Программный продукт, предназначенный для создания средств анализа скрытых закономерностей в данных и многих других приложений.
- Великолепный и очень мощный программный инструмент для создания семантических сетей с гиперссылками, предоставляемый бесплатно (`http:`

//cmap.ihmc.us/). Может также применяться для представления обычных и экспертных знаний в визуальном формате. Удобное инструментальное средство для приобретения знаний (эта тема обсуждалась в главе 6). От пользователей во всем мире получено множество примеров. Прежде чем подписаться на список рассылки указанного узла, подготовьтесь к тому, что значительная часть электронной почты будет поступать на испанском языке.

- TAP — это база знаний, предназначенная для оказания помощи при создании сети Semantic Web (http://www.w3.org/2001/sw/), которая представляет собой усовершенствованную версию современной сети World Wide Web, обеспечивающую чтение данных с помощью компьютеров. На узле TAP имеются также другие многочисленные ссылки (http://www.iturls.com/English/TechHotspot/TH_SemanticWeb.asp).

  В настоящее время поиск в Web может осуществляться только по ключевым словам с использованием примитивных булевых операций. Безусловно, содержимое некоторых Web-узлов переводится на такие языки, доступные для чтения компьютером, как MusicXML, RuleXML, MathXML и т.д., основанные на языке XML, но база знаний TAP позволяет воспользоваться более значительным объемом информации о реальном мире и текущих событиях, представленной в семантическом контексте.

  TAP — это база поверхностных, но весьма обширных знаний, содержащая основную лексическую и таксономическую информацию по широкому перечню часто используемых тематических рубрик, таких как музыка, кинофильмы, творческие произведения и их авторы, спорт, автомобили, компании, домашние приборы, игрушки, товары для детей, достопримечательности, потребительская электроника и здравоохранение. Эта база знаний может быть загружена в целях проверки экспертных систем многих различных типов. Она по сути представляет собой поверхностную онтологию, которая предназначена для дополнения, а не замены базы знаний Cyc, имеющей более глубокую онтологию знаний о феноменах, включаемых в круг понятий здравого смысла, но не о текущих событиях. Поэтому, например, с помощью Cyc нельзя узнать, что представляет собой *Yo-Yo Ma*. На основе базы знаний TAP могут быть выполнены интересные студенческие проекты (http://tap.stanford.edu/tap/tapkb.html).

- Семейство систем представления знаний Classic предназначено для создания приложений, обладающих лишь ограниченной выразительной мощью, но обеспечивающих быстрое получение ответов на вопросы. С помощью семейства систем Classic могут быть созданы приложения, обладающие большинством характеристик, свойственных семантическим сетям. Classic дает возможность пользователям представлять описания, понятия, роли, индивидуумов, правила, фреймы из других систем представления знаний и объ-

екты объектно-ориентированных языков программирования. Но наиболее интересным свойством семейства систем Classic является то, что в нем концепции автоматически организуются в таксономии и по мере необходимости осуществляется автоматическое наследование объектов.

Семейство систем Classic позволяет также обнаруживать несовместимости в сообщаемой ей информации. (Поэтому программы из данного семейства, по-видимому, следует использовать для того, чтобы они отвечали на звонки представителей службы электронной торговли по телефону или на спам, поступающий по электронной почте, с сообщениями о том, как можно заработать 10 миллионов долларов, оказав помощь зарубежному дипломату, которому требуется положить большие деньги на банковский счет в вашей стране. С другой стороны, эти программы вряд ли стоит привлекать, чтобы они помогли супруге разобраться в ваших объяснениях по поводу того, почему вы не пришли домой вовремя вчера вечером.)

В состав семейства Classic входят три системы. Во-первых, версия на языке LISP, которая была разработана в исследовательских целях, во-вторых, версия на языке C и, в-третьих, версия NeoClassic, написанная на языке C++. Семейство систем Classic использовалось при создании программ настройки конфигураций PROSE и QUESTAR, с помощью которых была выполнена настройка конфигурации изделий AT&T и Lucent стоимостью более 4 миллиардов долларов. Для того чтобі ознакомиться с дополнительными сведениями о семействе систем Classic, обратитесь по адресу `http://www.bell-labs.com/project/classic/`.

- Цель проекта Rule Markup Initiative (сокращенное название RuleML) — разработка стандарта разметки правил для прямого и обратного логического вывода, позволяющего предусмотреть универсальный подход к формированию правил, пригодных для ввода во всевозможные экспертные системы. С этим проектом связан еще один проект, направленный на создание языка RuleXML. Кроме того, предпринимаются усилия по достижению совместимости стандартов RuleML и RuleXML. Узел RuleML находится по адресу `http://www.ruleml.org/`.

  Задача состоит в том, что должен быть создан единый универсальный формат для правил, позволяющий избежать необходимости разрабатывать код с использованием различных стилей, что позволило бы упростить обмен знаниями между различными экспертными системами. В настоящее время отсутствует стандартный способ передачи знаний из одной экспертной системы в другую, поскольку для разных инструментальных средств экспертных систем используется собственный синтаксис. Проект создания стандарта RuleML является очень важным и должен открыть весьма перспективное направление исследований с огромной отдачей в случае его успешной ре-

ализации. По существу, достигаемые при этом преимущества аналогичны повторному использованию программного обеспечения вместо проведения всех работ с нуля после перехода к созданию новой экспертной системы. Ближе всего к указанной цели позволяет подойти практическая реализация объектов на языке CLIPS, поскольку объекты вполне определены и могут передаваться между различными экспертными системами CLIPS. С другой стороны, хотя язык Jess создан на базе объектно-ориентированного языка Java, по состоянию на 2004 год Jess не поддерживал возможность применения объектов в правилах, иными словами, не поддерживал возможности языка COOL. Главным преимуществом языка Jess является то, что он может быть внедрен в базовый язык Java.

Кроме того, объекты содержат данные и методы, а также имеют вполне определенный интерфейс, поэтому позволяют проще создавать крупномасштабные экспертные системы. Сложности, связанные с созданием гибридной, объектно-ориентированной системы, основанной на правилах, были буквально грандиозными, но система CLIPS непрерывно совершенствовалась и подвергалась тщательному тестированию перед каждым выпуском, начиная с 1986 года. Безусловно, у нас, разработчиков системы, возникал сильный соблазн ввести новые средства, такие как обратный логический вывод, нечеткая логика, байесовские возможности и многие другие, но мы всегда стремились избегать таких соблазнов в пользу создания более простого и надежного инструментального средства, на которое могли бы опираться другие разработчики как на надежную программную основу. Теперь стало очевидно, что этот подход себя полностью оправдал, поскольку появилось большое количество языков, происходящих от CLIPS, но оснащенных дополнительными средствами, и их перечень постоянно увеличивается (а мы как разработчики не обязаны их поддерживать!).

Программные ресурсы, относящиеся к логике

Значительная часть работ в области искусственного интеллекта направлена на автоматизацию процессов формирования рассуждений, с помощью которых люди осуществляют логический вывод. В действительности одной из первых проблем, возникших перед искусственным интеллектом, было осуществление процессов формирования символических рассуждений, выполняемых математиками, а не создание нового класса машин для сложных математических расчетов. Безусловно, машины использовались уже тысячи лет для умножения мощи человеческих мускулов, а для арифметических расчетов применялись простые устройства, такие как абак, но теперь впервые в истории брошен вызов интеллектуальным способностям человеческого разума в части доказательства символических математических теорем.

Логические ошибки

Для специалистов в области экспертных систем, выявляющих у экспертов знания в проблемной области с помощью интервью, важно учитывать опыт, накопленный в результате изучения логических ошибок. Дело в том, что в одних ситуациях жертвой логических ошибок может стать даже сам эксперт, а в других допускает неправильное толкование сказанного экспертом специалист по экспертным системам, поэтому составляет формулировку, содержащую в себе логическую ошибку. Это означает, что знания эксперта необходимо подвергнуть проверке, прежде чем поместить их в базу знаний. Узел, на котором можно ознакомиться с подробным перечнем логических ошибок, указан в конце раздела данного приложения, относящегося к главе 6. (Если вы действительно хорошо освоите тематику логических ошибок, то всегда сможете оставить работу в области искусственного интеллекта и успешно заниматься юриспруденцией или политикой. Многие логические ошибки, хотя и нарушают принципы логики, служат в качестве очень убедительных доказательств, которым люди склонны верить, поэтому используются в качестве безотказного метода убеждения людей в том, что предъявляемые им аргументы являются правильными.)

- Качественное общее описание и примеры логических ошибок, возникающих в процессе дедуктивного вывода, можно найти по следующему адресу: `http://webpages.shepherd.edu/maustin/rhetoric/deductiv.htm`.
- Логические ошибки, возникающие в процессе индуктивного вывода (`http://webpages.shepherd.edu/maustin/rhetoric/inductiv.htm`).
- Список многих других логических ошибок (`http://webpages.shepherd.edu/maustin/rhetoric/fallacies.htm`).
- *Critical Thinking on the Web* (`http://www.austhink.org/critical/`). Узел, содержащий многочисленные ссылки на материалы по логике и оперативные учебники, а также очень разносторонний список логических ошибок.

Программное обеспечение, применяемое при изучении логики

Ханс ван Дитмарш (Hans van Ditmarsch), с которым можно связаться по адресу `hans@cs.otago.ac.nz`, представил на своей Web-странице большую подборку ссылок на программное обеспечение, доступное в Интернете, которое представляет широкое разнообразие тем, относящихся к логике. Это программное обеспечение может использоваться для проверки правильности ответов на логические задачи и ознакомления с действием средств автоматического формирования рассуждений. В общем в этом списке можно найти информа-

цию об образовательном программном обеспечении, которое относится к логике, а также другие ссылки (http:/www.cs.otago.ac.nz/staffpriv/hans/logiccourseware.html).

Программы для доказательства теорем

Для выполнения более сложных заданий по логике используется такое развитое программное обеспечение, как программы автоматического доказательства теорем, средства автоматизации формирования логических выводов или средства механизированного проведения рассуждений.

- *Mechanized Reasoning Home Page* (http://www-formal.stanford.edu/clt/ARS/ars-db.html).
- *World Wide Web Virtual Library: Formal Methods*. Огромная коллекция логических инструментальных средств и языковых верификаторов (http://vl.fmnet.info/).
- *Formal Methods Education Resources* (http://www.cs.indiana.edu/formal-methods-education/).
- *Logik Software fuer Unterrichtszwecke* (http://www.phil-fak.uni-duesseldorf.de/logik/software.html). Информация представлена на немецком языке.
- *Database of Existing Mechanized Reasoning Systems*. Крупный информационный ресурс с описаниями и ссылками (http://www-formal.stanford.edu/clt/ARS/systems.html).
- Программы и учебные пособия по логике, программы для доказательства теорем и языки (http://home.clara.net/ghrow/subjects/logic_software.html).
- *Newsletter on Philosophy and Computers*. Информационный бюллетень (http://www.apa.udel.edu/apa/publications/newsletters/computers.html).
- Весной 2000-го года был проведен очередной тест Тьюринга, который оказался весьма знаменательным, поскольку совпал с 50-летним юбилеем предсказания Тьюринга, сформулированном в 1950-м году, что через 50 лет компьютеры будут проходить его тест. Безусловно, людям все еще удается очень быстро узнавать, что с ними беседует компьютер, но любопытно отметить, что до сих пор бытующие представления о том, что сами люди являются компьютерами, имеют более длительную предысторию, чем тест Тьюринга. Но в целом приятно отметить, что не настало еще время, когда люди будут заменены компьютерами (http://www.apa.udel.edu/apa/publications/newsletters/v99n2/computers/chair.asp).

- *Linear Logic Prover* (`http://bach.cs.kobe-u.ac.jp/llprover/`). Наоюки Тамура (Naoyuki Tamura) предоставляет доступ в Web к своей программе линейного логического доказательства теорем. На его Web-узле содержится много ссылок на другие средства автоматического доказательства теорем, программы на многих языках, включая PROLOG, а также ссылки на программное обеспечение.

Проекты и исследования, посвященные логическому образованию

- *Association of Symbolic Logic: Committee on Logic Education*. Большое количество ссылок на огромный объем логического программного обеспечения и информационных ресурсов, применяемых для преподавания логики (`http://www.math.ufl.edu/~jal/asl/logic_education.html`).
- *Taller de Didactica de la Logica* (информация представлена на испанском языке). Список развернутых в Мексике Web-узлов, посвященных логическому образованию (`http://www.filosoficas.unam.mx/~Tdl/TDL.htm`).
- *Aracne* (информация представлена на испанском языке). Логические информационные ресурсы для Испании и Латинской Америки (`http://aracne.usal.es/`).
- Web-узел Дэвида Гри (David Gries), `http://www.cs.cornell.edu/gries/`. На этом узле представлены многочисленные ссылки на информационные ресурсы, посвященные логике и особенно новым направлениям в логике. Кроме того, приведен материал одной из многих книг Дэвида, *A Logical Approach to Discrete Math*, Springer-Verlag, 1993 (`http://www.cs.cornell.edu/gries/Logic/intro.html`).
- В Университете Бакнелла, США, ведется список рассылки, предназначенный для обсуждения проблем логического образования. Чтобы принять участие в его работе, отправьте письмо с сообщением `subscribe logic-l@bucknell.edu` по адресу `listserv@bucknell.edu`.
- Университет Карнеги–Меллона предоставляет доступ к оперативному программному обеспечению автоматизированного обучения в области причинно-следственных рассуждений. Программы и примеры Causality Lab, предоставляемые для бесплатной загрузки, учат студентов критическому мышлению, чтобы они ничего не принимали на веру без реальных свидетельств. Это особенно важно в наши дни, когда люди верят всему, что показывают и сообщают по телевидению, даже по прошествии многих лет после того, как было доказано, что это неправильно.

Тема причинно-следственных рассуждений связана с другой важной темой — поддержанием истинности в экспертных системах. В системе CLIPS предусмотрены такие средства поддержания истинности, что после доказательства ложности факта система отменяет все последующие операции добавления, модификации и удаления фактов и правил, выполненные на основании ложного факта, и состояние системы восстанавливается до того состояния, в котором она находилась перед добавлением ложного факта. Это позволяет постоянно поддерживать способность системы CLIPS к выводу действительных заключений, на что не способны большинство людей (`http://www.phil.cmu.edu/projects/csr/`).

- *The Self-Paced Logic Project*. Логический проект, рассчитанный на индивидуальную скорость обучения, в рамках которого предоставляется и разрабатывается программное обеспечение для создания и сопровождения банка контрольных вопросов для крупного курса логического обучения, рассчитанного на индивидуальные занятия. Банк контрольных вопросов создается программой автоматически в соответствии с поставленными целями и обновляется в каждом семестре. Программе не предоставляется содержание курса, но автор вопросов имеет в своем распоряжении весьма разностороннее учебное руководство, позволяющее ему определить, что, по его мнению, должен знать студент, изучающий логику. Список тем является настолько обширным, что студенты, которым пришлось принять участие в выполнении контрольных заданий, больше не задают друг другу вопросы: “Что именно я должен выучить, чтобы пройти экзамен?” (`http://www.sp.uconn.edu/~py102vc/selfpace.htm`).

Глава 3

- *Probability Web*. Крупная коллекция информационных ресурсов по теории вероятности (`http://www.mathcs.carleton.edu/probweb/probweb.html`).
- *Chance* (`http://www.dartmouth.edu/~chance/`). Узел, на котором представлены материалы, применяемые во время преподавания курса Chance — курса обучения количественным методам. Этот курс нацелен на то, чтобы студенты стали более информированными, критически настроенными читателями текущих новостей, в которых используются вероятностные и статистические данные. На этом узле представлена также превосходная книга по теории вероятности и статистике, написанная Гринстидом (Grinstead) и Снелл (Snell). На узле Chance представлены также компьютерные программы и ответы на упражнения с нечетными номерами.

- *Probability Computer Projects with Mathematica* (http://www.wku.edu/~neal/probability/prob.html). Вероятностные компьютерные проекты на основе программы Mathematica.
- Узел с многочисленными ссылками (http://www.maths.uq.oz.au/~pkp/probweb/probweb.html).
- Программное обеспечение по теории вероятности и статистике (http://forum.swarthmore.edu/probstat/probstat.software.html).
- Проект публикации в Интернете общедоступного программного обеспечения и предоставляемых в оперативном режиме материалов по теории вероятности и статистике (http://forum.swarthmore.edu/probstat/probstat.projects.html).
- Обзор многочисленных программных инструментов для принятия решений в условиях неопределенности, подготовленный Лабораторией систем принятия решений Питсбургского университета (http://www.sis.pitt.edu/~dsl/software.htm).

Глава 4

Статистика, теория вероятности и анализ скрытых закономерностей в данных

- Крупная общая коллекция программного обеспечения по теории вероятности, статистике и байесовским методам. Некоторые программы написаны для операционной системы DOS, но все еще работают (http://archives.math.utk.edu/software/msdos/probability/.html).
- Крупная коллекция наборов данных для проверки программного обеспечения анализа скрытых закономерностей в данных (http://lib.stat.cmu.edu/datasets/).
- Многочисленные наборы данных, которые могут применяться при изучении книг по статистике. Значительная часть из них, в том числе наборы данных работы с марковскими моделями, предназначены для использования с программой Excel (http://www.duxbury.com/cgi-brookscole/course_products_bc.pl?fid=M67&discipline_number=17).
- Широкий выбор программ, предназначенных для вероятностных расчетов (http://softsia.com/re.php?kw=Probability+Distribution+Calculator).

- Хороший выбор интересных видеофильмов, предоставляемых для загрузки, которые посвящены теории вероятности, статистике и приложениям к реальным задачам (http://www.dartmouth.edu/~chance/ChanceLecture/download.html).
- Огромный архив математических ресурсов. Программное обеспечение, документы, учебники, учебные пособия (включая слайды PowerPoint) и списки книг с относящимся к ним программным обеспечением (http://bayes.stat.washington.edu/almond/belief.html).
- Крупный список ссылок на математическое программное обеспечение всевозможного назначения (http://www.math.fsu.edu/Virtual/index.php?f=21).
- Огромный список международных ресурсов по статистике (http://gsociology.icaap.org/methods/statontheweb.html).
- Весьма разносторонний список ссылок на все виды статистических информационных ресурсов (http://gsociology.icaap.org/methods/statontheweb.html).
- Крупная коллекция предоставляемого бесплатно статистического программного обеспечения (http://members.aol.com/johnp71/javasta2.html).
- Узел, посвященный марковским цепям (http://www.saliu.com/Markov_Chains.html).

Байесовские ресурсы

- Начальная страница Кевина Патрика Мэрфи (Kevin Patrick Murphy) (http://www.ai.mit.edu/~murphyk/). Ссылки на важные источники информации о байесовских сетях, информационные ресурсы и результаты сравнения байесовских программных продуктов и другого программного обеспечения.
- *Bayesian Network Editor and Toolkit* (http://research.microsoft.com/adapt/MSBNx/). Предоставляемое бесплатно компанией Microsoft байесовское инструментальное средство анализа, предназначенное для создания, проверки и оценки байесовских сетей. Инструментарий MSBNx используется в таких программах Microsoft, как Office Assistant и Technical Support Troubleshooters, а также в фильтрах против спама.
- *Bayes Net Toolbox v5 for MATLAB* (http://www.ai.mit.edu/~murphyk/Software/BNT/bnt.html). Программное обеспечение, предоставляемое лабораторией искусственного интеллекта и компьютерных наук Массачусетсского технологического института (MIT), Кембридж.

- *Bayesian Knowledge Discoverer* (http://kmi.open.ac.uk/projects/bkd/). Программное обеспечение, способное к обучению байесовских сетей доверия. Доступна версия коммерческого программного обеспечения, предоставляемая студентам бесплатно.
- Инструментальные средства Java для байесовских сетей и искусственного интеллекта (http://bndev.sourceforge.net/).
- Очень сложное байесовское инструментальное средство для анализа скрытых закономерностей в данных и решения других задач, которое включает версию для бесплатного опробования (http://www.bayesia.com/).
- *Software for Manipulating Belief Networks* (http://archives.math.utk.edu/) Многочисленные информационные ресурсы, которые относятся к графическим моделям доверительных функций и близким темам, таким как байесовские сети, диаграммы влияния и вероятностные графические модели.
- *Bayesian and Dependency Networks Software* (http://www.kdnuggets.com/software/bayesian.html). Крупная коллекция байесовского программного обеспечения и информационных ресурсов.
- Крупная коллекция байесовского и марковского программного обеспечения и информационных ресурсов (http://www.cs.toronto.edu/~radford/fbm.software.html).
- *Bayesian networks and influence diagrams* (http://www.ia.uned.es/~fjdiez/bayes/). Великолепный узел с многочисленными ресурсами, относящимися к теории вероятности и байесовской теории, а также с большим количеством программного обеспечения.
- Коммерческое инструментальное средство для создания байесовских сетей доверия и влияния (http://norsys.com). Превосходный оперативный учебник по байесовским сетям с примерами из области автоматизированной диагностики, предсказания, управления финансовым риском, комплектования портфеля ценных бумаг, страхования, моделирования экосистем и сплавления данных от датчиков (http://norsys.com/tutorials/netica/nt_toc_A.htm).

Глава 5

- *Dempster-Shafer Theory Homepage of Glenn Shafer*. Начальная страница Гленна Шефера. Многочисленные информационные ресурсы, в том числе статьи, посвященные работам Гленна (http://www.glennshafer.com/). Особый интерес представляет очень короткое и ясное объяснение теории Демпстера–Шефера (http://www.glennshafer.com/assets/downloads/article48.pdf).

- *Free MatLab Software on Belief Functions and Pattern Recognition* (`http://www.hds.utc.fr/~tdenoeux/software.htm`). Предоставляемое бесплатно программное обеспечение различных моделей, относящихся к таким областям, как теория Демпстера–Шефера, нейронные сети и нечеткая логика.
- Качественные учебные материалы, предоставляемые Дейвом Маршаллом (Dave Marshall) в оперативном режиме, в том числе по теории вероятности, теореме Байеса, сетям доверия, коэффициентам достоверности, теории Демпстера–Шефера, байесовским сетям и нечеткой логике (`http://www.cs.cf.ac.uk/Dave/AI2/node84.html`).
- Качественные учебные материалы, предоставляемые в оперативном режиме Алленом Рамсеем (Allan Ramsay). На узле Аллена представлены ссылки на материалы, позволяющие понять, почему логика предикатов неприменима для теоретического исследования проблем реального мира, а также приведены рекомендации по использованию теории нечетких множеств, теории Демпстера–Шефера и других альтернативных подходов (`http://www.ccl.umist.ac.uk/teaching/material/5005/`).
- *Dempster-Shafer Methods for Object Classification*. Статья, сложная для восприятия, но позволяющая понять, как теория Демпстера–Шефера может использоваться для организации противоракетной обороны (`http://www.stormingmedia.us/28/2812/A281293.html`).
- *What is Dempster-Shafer's model?* Вводное описание метода Демпстера–Шефера, предоставляемое в оперативном режиме (`http://iridia.ulb.ac.be/~psmets/WhatIsDS.pdf`).
- Начальная страница организации International Fuzzy Systems Association (Международная ассоциация по нечетким системам). Представлено много информационных ресурсов по нечеткой логике (`http://www.pa.info.mie-u.ac.jp/~furu/ifsa/`).
- Магдебургский университет предоставляет большой объем программного обеспечения по нейронным сетям и нечетким множествам (`http://fuzzy.cs.uni-magdeburg.de/software.html`).
- Христиан Боргельт (Christian Borgelt) из Магдебургского университета предоставляет обширную коллекцию программного обеспечения, предоставляемого бесплатно по лицензии GNU по адресу (`http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html`).

Ресурсы, относящиеся к нечеткой логике

- Удобная страница со ссылками на многие информационные ресурсы по нечеткой логике, предоставляемая компанией Ortec Engineering, которая

выпускает аппаратные средства нечеткой логики (http://www.ortech-engr.com/fuzzy/reservoir.html#sharks).

- *Fuzzy Logic Laboratorium*. Организация, ежегодно проводящая семинары по нечеткой логике и математике (http://www.flll.uni-linz.ac.at/navigation/main_navigation/frame_research.html).
- *Fuzzy Sets & Systems* (http://www.elsevier.com/wps/find/journaldescription.cws_home/505545/description). Научный журнал, издаваемый компанией Elsevier Science Inc.
- Система CLIPS с поддержкой нечеткой логики; версия компании NRC (http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyClips/fuzzyCLIPSIndex.html).
- James Mathews, *An Introduction to Fuzzy Logic*. Краткий учебник по нечеткой логике и пример того, как реализовать средства нечеткой логики на языке C++, с кодом и документацией (http://www.generation5.org/content/1999/fuzzyintro.asp).

Глава 6

- Компания Microsoft предоставляет доступ к огромному каталогу, предназначенному для управления знаниями (Catalog on Knowledge Management), по адресу http://www.microsoft.com/windows/catalog/.
- KnowledgeBase.net 4.0 (http://www.knowledgebase.net/). Пример очень сложного инструментального средства, предназначенного для поддержки всех функций управления знаниями. Интересные материалы о современных тенденциях в мире информационных технологий.
- *Personal Construct Psychology* (http://repgrid.com/pcp/). Узел посвящен проблемам психологии конструирования личности — популярной теории, в соответствии с которой люди играют по отношению к самим себе роль ученых, непрерывно выдвигающих гипотезы и теории об окружающем мире. Психология конструирования личности использовалась в качестве основы для приобретения знаний с помощью личных устойчивых решеток. На этом узле представлено также много ресурсов и ссылок.
- Узел IKMS (Information and Knowledge Management Society). На этом узле представлено много материалов конференций, статей, практических примеров и программных ресурсов (http://www.ikms.org.sg/resources/index.html).
- Крупный международный портал по управлению знаниями, на котором представлены многочисленные документы и предусмотрены удобные средства поиска (http://www.kmtool.net/).

- *Brint Institute: The Knowledge Creating Company* (`http://www.kmbook.com/`). Узел, на котором представлено много превосходных статей по широкому спектру тем, относящихся к управлению знаниями. Статьи сопровождаются превосходными аннотациями, которые позволяют безошибочно выбирать только те материалы, которые действительно представляют для вас интерес.
- *Course on Information Management in Organizations: models and platforms*. Узел, который сопровождает Чун Вей Чу (Chun Wei Choo), содержит много информационных ресурсов, относящихся к различным практическим примерам и видам средств управления знаниями (`http://choo.fis.utoronto.ca/FIS/Courses/LIS2102/LIS2102.slides.html`).
- Achour S.L., et. al., *A UMLS-based Knowledge Acquisition Tool for Rule-based Clinical Decision Support System Development* (`http://www.pubmedcentral.gov/articlerender.fcgi?tool=pmcentrez&artid=130080`).
- Инструментальное средство приобретения знаний и командный интерпретатор экспертной системы Acquire®. Разумеется, это инструментальное средство уже упоминалось ранее, в разделе "Приложения экспертных систем", но в данном разделе оно должно быть указано снова, в связи с предоставляемыми им возможностями приобретения знаний. Безусловно, это инструментальное средство не способно заменить профессионального инженера по знаниям, но может вполне найти для себя применение в определенных областях, где знания структурированы как деревья решений, поэтому могут быть выявлены путем получения ответов на вопросы и автоматического формирования правил. Испытательная версия этого программного обеспечения предоставляется на следующем Web-узле: `http://www.aiinc.ca`.
- Чтобы не допустить ошибки в правиле, инженер по знаниям должен внимательно выслушать эксперта в проблемной области, выявляя знания, необходимые для составления правил. С другой стороны, наличие логической ошибки в правилах, применяемых для создания адвокатской экспертной системы, вполне допустимо, поскольку речи, произносимые в суде, не обязательно должны быть логически безупречными, ведь они должны лишь показать присяжным то, на чьей стороне выступает лучший адвокат. Основной особенностью ложных выводов является то, что люди склонны им верить, а наилучшие софизмы звучат даже более убедительно, чем истинные утверждения. Специалисты собрали целую коллекцию ложных доводов, которые показали свою применимость в качестве средств убеждения, поскольку часто позволяют доказать, что вы правы, а ваш противник неправ, даже если вы неправы, а ваш противник прав. Как и искусство владения ложными доводами, в качестве средства убеждения применяется также риторика, по-

этому, прежде чем вступать в брак, не мешает овладеть этим искусством. На указанном здесь узле опубликован краткий список логических доводов. Среди них встречаются очень хорошие методы, которые не только помогут адвокату или недобросовестному супругу, но и позволят убедить преподавателя, что вы заслуживаете отличной оценки, или начальника, что вам нужно повысить зарплату (`http://www.iep.utm.edu/f/fallacies.htm`).

Глава 7

Язык CLIPS используется для преподавания предмета "Экспертные системы" в сотнях университетов во всем мире. Проще всего можно узнать, как организованы типичные курсы, а также ознакомиться с подборками учебных материалов, воспользовавшись машиной поиска. Ниже в качестве примера перечислено несколько узлов.

- Начальная страница разработчиков CLIPS. Ссылки на многочисленные информационные ресурсы (`http://www.ghgcorp.com/clips/CLIPS.html`).
- `http://www.cs.unc.edu/Admin/Courses/descriptions/275.html`.
- `http://www.cs.wpi.edu/~dcb/courses/CS538/`.

Приложение 3

Литература

1. Adami C. (1998) *Knowledge Introduction to Artificial Life*, Springer-Verlag, p. 5.
2. Becerra-Fernandez I., Gonzalez A., and Sabherwal R. (2003) *Knowledge Management*, Prentice-Hall.
3. Begley R.J., Riege M., Rosenblum J., and Tseng D. (2003) *Adding Intelligence to Medical Devices. An overview of decision support and expert system technology in the medical device industry*, `http://www.devicelink.com/mddi/archive/00/03/014.html`. Качественное описание проблематики разработки средств приобретения знаний и итоговые сведения по системам принятия решений в виде удобных таблиц.
4. Bentley P.J. et al. (2002) *Creative Evolutionary Systems*, Academic Press.
5. Bergadano F. (1996) *Inductive Logic Programming*, The MIT Press.
6. Berlinski D. (2000) *The Advent of the Algorithm*, Harcourt, Inc., p. xvix.
7. Bertino E. et al. (2000) *Intelligent Database Systems*, Addison-Wesley International.
8. Brachman R., McGuinness D., Patel-Schneider P., Borgida A., and Resnick L. (1991) Living with CLASSIC: When and How to Use a KL-ONE-Like Language, *Principles of Semantic Networks*, Morgan Kaufman, p. 401–456, May.
9. Bramer M.A. (Ed.) (1999) *Knowledge Discovery and Data Mining*, IEEE Press.
10. Brewka G. (1997) *Principles of Knowledge Representation*, CSLI Publications.
11. Brown J. (2000) *Minds, Machines, and the Multiverse*, Simon & Schuster.
12. Brownston L. et al. (1985) *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley.
13. Castillo E. et al. (Ed.) (1997) *Expert Systems and Probabilistic Network Models*, Springer-Verlag New York, Inc.

14. Chandler D. (2001) *Semiotics: The Basics*, Rutledge Press. Сокращенная версия, доступная в оперативном режиме, предоставляется по адресу: `http://www.aber.ac.uk/media/Documents/S4B/`.
15. Chen G. and Pham T.T. (2001*) Introduction to Fuzzy Sets, Fuzzy Logic, and Fuzzy Control Systems*, CRC Press.
16. Constantin V.A. (1995) *Fuzzy Logic and Neuro Fuzzy Applications Explained*, Prentice Hall.
17. Conway S. and Sligar C. (2002) *Unlocking Knowledge Assets: Solutions from Microsoft*, Microsoft Press.
18. Cotterill R. (1998) *Enchanted Looms*, Cambridge University Press, p. 360.
19. Das B. (1999) *Representing Uncertainties Using Bayesian Networks*, `http://www.dsto.defence.gov.au/corporate/reports/DSTO-TR-0918.pdf`. Очень подробный отчет о создании ответственного приложения байесовских сетей, предназначенного для моделирования и формирования рассуждений о неопределенностях в реалистичном сценарии вооруженного столкновения с участием военно-морских сил. Намного более сложный пример по сравнению с примером системы Prospector, приведенным в настоящей книге.
20. de Silva C.W. (Ed.) (2000) *Intelligent Machines: Myths and Realities*, CRC Press, p. 13.
21. Debenham J. (1998) *Knowledge Engineering: Unifying Knowledge base and Database Design*, Springer-Verlag.
22. Dempster A. P. (1967) Upper and Lower Probabilities Induced by Multivalued Mappings, *Annals of Math. Stat.*, 38, p. 325–329.
23. Dorigo M. and Thomas Stutzle (2004) *Ant Colony Optimization*, The MIT Press. Следует отметить, что к данной книге прилагается компакт-диск с программным обеспечением для решения задачи коммивояжера с использованием эволюционного алгоритма функционирования колонии муравьев.
24. Fetzer J.H. (2001) *Computers and Cognition: Why Minds Are Not Machines*, Kluwer Academic Publishers, p. 25–182.
25. Firebaugh M.W. (1988) *Artificial Intelligence: A Knowledge-Based Approach*, Boyd & Fraser Publishing Co.
26. Fogel G.B. and Corne D.W. (Ed.) (2003) *Evolutionary Computation in Bioinformatics*, Morgan Kaufmann Publisher.
27. Forgy C.L. (1979) *On the Efficient Implementation of Production Systems*, Ph.D. thesis, Carnegie-Mellon University.
28. Forgy C.L. (1985) Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence*, 19, p. 17–37.

29. Foster I. (2003) The Grid: Computing without Bounds, *Scientific American*, Volume 288, Number 4, p. 78–85, April.

30. Friedman-Hill E. (2003) *Jess in Action: Rule-Based Systems in Java*, Manning Publications.

31. Gardenfors P. (2004) *Belief Revision*, Cambridge Tracts in Theoretical Computer Science.

32. Gates B. (2000) *Business @ the Speed of Thought: Succeeding in the Digital Economy*, Warner Business Books. Выдержка из этой книги, касающаяся применения байесовской логики в программных продуктах Microsoft, приведена по адресу `http://www.microsoft.com/billgates/speedofthought/additional/badnews.asp`.

33. Giarratano J.C. (1993) *CLIPS User's Guide*, NASA, Version 6.2 of CLIPS.

34. Giarratano J.C. (2004) `http://www.pcai.com/Paid/Issues/PCAI-Online-Issues/17.4_OL/New_Folder/So&9i2/17.4_PA/PCAI-17.4-Paid-pp.18-Art1.htm`.

35. Giarratano J.C., et al. (1990) Future Impacts of Artificial Neural Systems on Industry, *ISA Transactions*, p. 9–14, Jan.

36. Giarratano J.C., et al. (1990) The State of the Art for Current and Future Expert System Tools, *ISA Transactions*, p. 17–25, Jan.

37. Giarratano J.C., et al. (1991) An Intelligent SQL Tutor, *1991 Conference on Intelligent Computer-Aided Training (ICAT '91)*, p. 309–316.

38. Giarratano J.C., et al. (1991) Neural Network Techniques in Manufacturing and Automation Systems // Leondes C.T. (Ed.) *Control and Dynamic Systems*, Vol. 49, Academic Press, p. 37–98.

39. Giarratano J.C., et al., (1991) Fuzzy Logic Control for Camera Tracking System, *Fifth Annual Workshop on Space Operations Applications and Research (SOAR '91)*, p. 94–99.

40. Gonzalez A.J. and Dankel D.D. (1993) *The Engineering of Knowledge-based Systems: Theory and Practice*, Prentice-Hall. Превосходное дополнение к настоящей книге. Тематика аналогична, но изложение — более подробное. В одном из приложений рассматривается даже более старая версия CLIPS для DOS. Однако рекомендуется новейшая версия CLIPS, описанная в настоящей книге. Широкий выбор интересных задач — от очень простых до очень сложных.

41. Grinstead C.M. and Snell J.L. (1997) *Introduction to Probability*, American Mathematical Society, 2nd edition. *Примечание.* Не только программное обеспечение, прилагаемое к книге, но и сама книга предоставляются бесплатно

по лицензии GNU по адресу `http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book-5-17-03.pdf`.

42. Gruber T.R. (1993) A Translation Approach to Portable Ontology Specification, *Knowledge Acquisition*, 5, p. 199–220.
43. Grzymala-Busse J.W. (1991) *Managing Uncertainty in Expert Systems*, Kluwer Academic Publishers.
44. Harris J. (2000) *An Introduction to Fuzzy Logic Applications*, Kluwer Academic Publishers.
45. Hecht-Nielsen R. (1990) *Neurocomputing*, Addison-Wesley Publishing Co., p. 147.
46. Helmreich S. (1998) *Silicon Second Nature*, University of California Press, p. 180–202.
47. Hopgood A.A. (2001) *Intelligent Systems for Engineers and Scientists*, CRC Press.
48. Hrycej T. (1997) *Neurocontrol*, John Wiley & Sons, Inc.
49. Huth M. and Ryan M. (2004) *Logic in Computer Science, 2nd Edition*, Cambridge University Press.
50. Ibrahim A.M. (2003) *Fuzzy Logic for Embedded Systems Applications*, Newnes. К книге прилагается также компакт-диск с большим количеством ссылок.
51. Jackson P. (1999) *Introduction to Expert Systems*, Addison-Wesley Publishing Co.
52. Jacquette D. (2001) *Symbolic Logic*, Wadsworth.
53. Jang J.S.R., Sun C.T., Mizutani E. (1996) *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*, Prentice-Hall.
54. Johnson S. (2001) *Emergence: The connected lives of ants, brains, cities, and software*, Scribner.
55. Kaelbling L.P., Littman M.L., and Cassandra A.R. (1998) Planning and Acting in Partially Observable Stochastic Domains, *Artificial Intelligence*, 101 p. 99–134.
56. Kahane H. and Tidman P. (2003) *Logic and Philosophy: A Modern Introduction*, 9th edition, Wadsworth.
57. Kantardzic M. (2003) *Data Mining*, IEEE Press.
58. Kasabov N. (2002) *Evolving Connectionist Systems*, Springer-Verlag, p. 7–29.
59. Kennedy J. and Eberhart R.C. (2001) *Swarm Intelligence*, Morgan-Kaufmann Publishers.
60. Klir G.J. and Weirman M.J. (1998) *Uncertainty-Based Information*, Physica-Verlag.

61. Korb K.B. and Nicholson A.E. (2004) *Bayesian Artificial Intelligence*, CRC Press. Книга включает очень обширное приложение, доступное в оперативном режиме, в котором проводится сравнение различных байесовских инструментальных средств: `http://www.csse.monash.edu.au/bai/book/appendix_b.pdf`.
62. Kosko B. (1996) *Fuzzy Engineering*, Prentice Hall.
63. Lajoie S.P. (Ed.) (2000) *Computers As Cognitive Tools, Volume Two: No More Walls*, Lawrence Erlbaum Associates, Publishers.
64. Lakemeyer G. and Nebel B. (Eds.) (2003) *Exploring Artificial Intelligence in the New Millennium*, Morgan Kaufmann Publishers.
65. Lakoff G. and Nunez R.E. (2000) *Where Mathematics Comes From*, Basic Books.
66. Leake D. et al. (Eds.) (1996) *Case-Based Reasoning*, AAAI Press/MIT Press.
67. Leondes C.T. (Ed.) (1998) *Fuzzy Logic and Expert Systems Applications*, Academic Press.
68. Liu P. (2004) *Fuzzy Neural Network Theory and Application (Machine Perception and Artificial Intelligence)*, World Scientific Publishing Company.
69. Luger G.F. (2002) *Artificial Intelligence*, Fourth Edition, Addison-Wesley.
70. Malhotra Y. (2002) Expert Systems for Knowledge Management: crossing the chasm between information processing and sense making, *Expert Systems with Applications journal*, (20) p. 7–16. Публикация доступна в оперативном режиме по адресу: `http://www.brint.org/expertsystems.pdf`.
71. Mendel J.M. (2000) *Uncertain Rule-Based Fuzzy Logic Systems: Introduction and New Directions*, Prentice-Hall.
72. Mendel J.M. (2001) *Introduction to Rule-Based Fuzzy Logic Systems*, IEEE Press.
73. Merrit D. (2004) Building a Custom Rule Engine with PROLOG, *Dr. Dobb's Journal*, p. 40–45, March.
74. Neapolitan R.E. (2003) *Learning Bayesian Networks*, Prentice-Hall.
75. Nguyen H.T. and Walker E.A. (1999) *A First Course in Fuzzy Logic*, CRC Press.
76. Novak J.D. (1998) Learning, Creating, and Using Knowledge: Concept Maps as Facilitative tools in Schools and Corporations, Lawrence Erlbaum and Associates.
77. Pearl J. (2000) *Causality: Models, Reasoning, and Inference*, Cambridge University Press.
78. Pohl R. (2004) *Cognitive Illusions: A Handbook On Fallacies And Biases In Thinking, Judgement And Memory*, Taylor & Francis.
79. Ross S.M. (2002) *Introduction to Probability Models, Eighth Edition*, p. 188–191, Academic Press.

80. Ross T.J. (2004) *Fuzzy Logic with Engineering Applications*, John Wiley & Sons.
81. Ruan D. (1997) *Intelligent Hybrid Systems: Fuzzy Logic, Neural Networks, and Genetic Algorithms*, Kluwer Academic Publishers.
82. Ruan D. and Kerre E.E. (2000) *Fuzzy If-Then Rules in Computational Intelligence*, Kluwer Academic Publishers.
83. Russell S. and Norvig P. (2003) *Artificial Intelligence*, Second Edition, by Pearson Education[1].
84. Saratchandran P., et al. (1996) *Parallel Implementations of Backpropagation Neural Networks on Transputers: A Study of Training Set Parallelism, Progress in Neural Processing 3*, World Scientific Pub. Co, July.
85. Satinover J. (2001) *The Quantum Brain*, John Wiley.
86. Siler W. and Buckley J.J. (2004) *Fuzzy Expert Systems: Theory and Applications*, Wiley-Interscience.
87. Sipper M. (2002) *Machine Nature: The Coming Age of Bio-Inspired Computing*, McGraw-Hill.
88. Sowa J.F. (2000) *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks-Cole.
89. Stebbing L.S. (1930) *A Modern Introduction to Logic*, Methuen and Co.
90. Swingler K. (1996) *Applying Neural Networks: A Practical Guide*, London: Academic Press.
91. Tiwana A. (2003) *Knowledge Management Toolkit: Orchestrating IT, Strategy, and Knowledge Platforms*, 2nd Edition, Prentice-Hall. Книга и компакт-диск с большим количеством практических примеров, программных ресурсов и ссылок.
92. Wagman M. (1999) *The Human Mind According To Artificial Intelligence*, Praeger Publishers, p. 76.
93. Wentworth J.A., Knaus R., and Aougab H. (1994) *Verification, Validation and Evaluation of Expert Systems*, `http://www.tfhrc.gov/advanc/vve/cover.htm`. Хорошая книга с описанием 327 правил, предоставляемая в оперативном режиме, которая отражает опыт, приобретенный Федеральным управлением шоссейных дорог США во время эксплуатации экспертной системы PAMEX.
94. Werbos P. (1994) *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting: Adaptive and Learning Systems for Signal Processing*, Wiley-Interscience.

[1] *Стюарт Рассел, Питер Норвиг (2005)*. Искусственный интеллект: современный подход, 2-е изд., *Издательский дом "Вильямс"*.

Предметный указатель

A

α-отсечение, *445*

ANS
Artificial Neural System, *103*

AV
Attribute-Value, *150*

B

Bel
belief, *424*

BNF
Backus–Naur Form, *78*

BPA
Basic Probability Assignment, *417*

C

CE
Conditional Element, *575*

CF
Certainty Factor, *404*; *950*

CLIPS
C Language Integrated Production system, *553*

COOL
CLIPS Object-Oriented Language, *20*

D

DARPA
Defense Advanced Research Projects Agency, *120*

Dbt
dubiety, *427*

F

FAQ
Frequently Asked Questions, *506*

FSM
Finite State Machine, *142*

G

GM
General Motors, *138*

GPA
Grade Point Average, *402*

GPS
General Problem Solver, *97*

H

HGL
High Guard Line, *987*

HMM
Hidden Markov Model, *82*

HRL
High Red Line, *987*

I

IFF
Identification Friend or Foe, *419*

Igr
ignorance, *427*

IM
Information Management, *506*

IP
Information Processing, *506*

IPA
Intellectual Property Agreement, *508*

IPO
Initial Public Offering, *517*

IS
Information System, *506*
IT
Information Technology, *506*

K

KM
Knowledge Management, *506*
KR
Knowledge Representation, *128*

L

L-логика, *467*
LEX
lexicographic, *396*
LGL
Low Guard Line, *987*
LHS
Left-Hand Side, *71*; *92*; *576*
LISP
LISt Processing, *92*
L_n
n-значная логика, *467*
LRL
Low Red Line, *987*

M

max-произведение, *456*
MB
Measure of Belief, *405*
MD
Measure of Disbelief, *405*
MDP
Markov Decision Process, *286*
MEA
Means-Ends Analysis, *396*
MGU
Most General Unifier, *261*
MTBF
Mean Time Between Failures, *530*

O

OAV
Object–Attribute–Value, *149*; *950*

P

$\prod$-функция, *443*
Pls
plausibility, *424*

R

Rete-алгоритм, *56*; *83*
сопоставления с шаблонами, *737*
RHS
Right-Hand Side, *74*; *92*; *576*
RuleML
Rule Markup Language, *549*

S

S-функция, *440*
SEC
Securities and Exchange Commission, *517*
SMART
Self-Monitoring, Analysis And Reporting Technology, *531*
Spt
support, *424*
SQL
Structured Query Language, *103*
supp
support, *445*
SUV
Sport Utility Vehicle, *138*

U

UML
Unified Modeling Language, *95*
UNK
unknown, *407*

V

V & V
Verification and Validation, *535*

W

WFF
Well-Formed Formula, *235*

X

XML
eXtensible Markup Language, *122*; *158*
XOR
Exclusive-OR, *107*

А

Аббревиатура V&V, *285*
Абдукция, *212*; *275*
Абстракция
  данных, *66*
  знаний, *66*
Автомат, *200*
  конечный, *142*
Агент, *287*
Агентство DARPA, *120*
Адрес
  внешний, *556*; *558*
  факта, *556*; *559*; *596*
  экземпляра, *556*; *559*
Аксиома, *175*; *236*
  равенства, *241*
Активизация, *51*; *576*
Алгоритм, *41*; *85*
  марковский, *81*
  унификации, *261*
Альфа-память, *743*
Анализ
  синтаксический, *142*
  скрытых закономерностей в данных, *32*; *104*; *128*
  целей и средств, *396*
Аналогия, *43*; *271*
Англоязычное издание, *22*
Антецедент, *71*; *214*
Артефакт, *397*
Архивирование документов, *496*
Архитектура классной доски, *493*
Атом, *92*
Атрибут, *148*
  access, *890*
  allowed-floats, *698*; *852*
  allowed-instance-names, *698*; *852*
  allowed-integers, *698*; *852*
  allowed-lexemes, *698*; *852*
  allowed-numbers, *698*; *852*
  allowed-strings, *698*; *852*
  allowed-symbols, *698*; *852*
  allowed-values, *698*; *852*
  auto-focus, *734*
  cardinality, *699*; *852*
  create-accessor, *890*
  default, *700*; *852*
  default-dynamic, *702*; *852*
  export, *723*
  import, *724*
  pattern-match, *877*
  propagation, *864*
  range, *699*; *852*
  role, *865*
  source, *863*
  storage, *917*
  type, *694*; *852*
  visibility, *886*
  допустимого значения, *698*
  слота, *853*
  сопоставления с шаблонами, *877*
Аттестация, *285*; *548*
  экспертной системы, *522*

Б

База данных
  интеллектуальная, *32*
  предикативная, *153*
База знаний, *46*; *552*
База фактов
  глобальная, *69*
Байесовское принятие решений, *328*
Барьер, *461*
Безразличие, *343*
Беспорядочное изменение, *304*
Бета-память, *745*
Биоинформатика, *120*

В

Вал, *377*
Вектор вероятностей, *337*
Верификация, *285*; *391*
Вероятность, *308*; *354*
  абсолютная, *322*
  апостериорная, *315*; *327*
  априорная, *308*
  байесовская, *297*
  безусловная, *322*
  внутренняя, *323*
  индивидуальная, *344*
  классическая, *308*
  маргинальная, *324*
  нечеткая, *485*

обратная, *326*; *327*
обычная, *403*
субъективная, *317*
условная, *321*
экспериментальная, *315*
эмпирическая, *315*
эпистемистическая, *404*
Вертикальная черта, *91*; *140*; *263*
Верхний уровень, *76*
Вершина, *194*
Включение множества, *448*
Возбуждение, *74*
Возведение множества в степень, *450*
Возможность, *482*
Возобновление нормальной работы, *708*
Временна́я избыточность, *83*; *738*
Временны́е ограничения, *44*
Вулканическая пробка, *377*
Выбор
источника знаний, *542*
способа представления знаний, *544*
стратегии реализации, *545*
Вывод
логический, *37*; *114*; *129*
логический обратный, *72*; *263*; *266*; *553*
логический правдоподобный, *281*
логический прямой, *72*; *263*; *266*; *553*
немонотонный, *278*
правил методом индукции, *42*
противоположный, *290*
Вызов
перегруженного метода, *811*; *817*
функции, *561*
Выигрыш, *331*; *509*
ожидаемый, *332*
Выработка
рекомендаций, *548*
случайных чисел, *839*
Выравнивание
влево, *646*
вправо, *646*
Выражение
наиболее общее, *261*
парадоксальное, *371*
предикативное, *151*
родительское, *244*
символическое, *92*
ссылочно прозрачное, *91*
условное, *229*
условное двустороннее, *229*
хорновское, *97*
Высказывание, *177*; *341*
атомарное, *144*
контингентное, *178*
нечеткое, *485*
составное, *177*
Высота, *446*

Г

Гарантия, *46*
Геномика, *120*
Гидроэлектрическая силовая установка, *495*
Гипотеза, *45*; *301*; *342*
верхнего уровня, *365*
нулевая, *301*
Грамматика, *140*
нечеткая, *463*
Граница
верхняя, *424*
нижняя, *424*
Границы незнания, *42*
Граф, *195*
ациклический, *195*
ориентированный, *145*; *195*
связный, *195*

Д

Данные, *266*
архивные, *129*
Двойная кавычка, *558*
Дедукция, *211*
Действие, *74*; *75*; *327*
оппортунистическое, *708*
Действия в реальном времени, *335*
Дерево, *194*
AND–OR, *206*
бинарное, *194*
вывода, *141*
вырожденное, *195*
генеалогическое, *195*; *679*
ориентированное, *194*

решений, *195*; *955*
решений самообучающееся, *197*
синтаксического анализа, *141*
событий, *312*
Диаграмма
Венна, *169*; *218*
дерева опровержения резолюции, *247*
конечного автомата, *200*
Дизъюнкт, *242*
Дилемма
деструктивная простая, *290*
деструктивная сложная, *290*
конструктивная сложная, *290*
Доверие, *424*
Доказательство, *130*; *214*
дедуктивное, *297*
действительное, *174*
индуктивное, *298*
логическое, *214*
непротиворечивое, *305*
Дополнение, *172*
множества, *447*
Дополнительные материалы, *22*
Достоверность, *371*
Дуга, *145*; *194*

Ж

Жизнь, *30*

З

Зависимость, *675*; *677*
Загрузка фактов, *835*
Задача, *539*
анализа результатов проверки, *546*
детерминированная, *206*
коммивояжера, *204*
перехода от нечеткого представления к четкому, *482*
правильно сформулированная, *206*
с обезьяной и бананами, *201*
слабо структурированная, *62*; *205*
формальной проверки, *546*
Заключение, *169*
дедуктивное, *305*
индуктивное, *305*
ложно отрицательное, *301*
ложно положительное, *301*
Закон
аддитивный, *321*
двойного отрицания, *228*
де Моргана, *228*
дизъюнктивного добавления, *228*
дизъюнктивного логического вывода, *228*
контрапозиции, *227*
конъюнктивного доказательства, *228*
конъюнкции, *228*
логического вывода, *227*
мультипликативный, *323*
отделения, *221*
упрощения, *228*
успеха второй, *510*
успеха минус второй, *524*
успеха минус первый, *520*
успеха минус третий, *534*
успеха нулевой, *513*
успеха первый, *513*
успеха третий, *510*
успеха четвертый, *507*
Замес, *478*
Заполнитель, *162*
Запрос, *154*
Запуск, *74*; *111*; *576*
Затененный обработчик, *945*
Знак, *511*
Знания
апостериорные, *132*
априорные, *132*
глубокие, *42*
декларативные, *133*
категорические, *158*
неявные, *133*
основанные на здравом смысле, *164*; *490*
поверхностные, *42*
подсознательные, *133*
полученные на основе здравого смысла, *212*
практические, *65*; *248*
причинные, *42*
процедурные, *133*
эвристические, *42*
экспертные, *136*

Значение, *148*
?NONE, *891*
composite, *863*
default, *163*
exclusive, *863*
inherit, *864*
initialize-only, *891*
local, *917*; *943*
no-inherit, *864*
non-reactive, *877*
private, *886*
public, *886*
reactive, *877*
read, *891*
read-only, *891*
read-write, *891*
shared, *917*
visible, *943*
write, *891*
возвращаемое, *561*; *632*
истинностное, *472*
истинностное антецедента, *473*
многозначное, *559*
пороговое, *107*; *408*
экзистенциальное, *175*
Значимость, *576*; *704*

И

Игра
Life, *770*
Sticks, *648*
в палочки, *648*
Идентификатор факта, *566*
Идентификация
источника знаний, *542*
элементов знаний, *543*
Иерархия, *194*
значимостей, *711*
Извлечение, *568*
ИИ
искусственный интеллект, *27*
Импликация, *178*
двусторонняя, *242*
материальная, *178*
Импорт, *723*
Имя
логическое, *641*
отношения, *561*
экземпляра, *556*; *559*
Индекс факта, *566*
Индукция, *149*; *211*; *304*
недействительная, *304*
обратная, *332*
Инженер
по знаниям, *40*; *529*
по онтологии, *121*
Инженерия знаний, *40*
Инкапсуляция, *886*
Инкрементное наращивание, *46*
Интеллект, *31*
искусственный, *27*
искусственный сильный, *27*
искусственный слабый, *27*
коллективный, *118*
Интенсификация, *452*
Интервал, *435*
проявления свидетельства, *423*; *426*
Интерпретатор
команд верхнего уровня, *76*
командный, *55*; *68*
командный экспертной системы, *89*
Интерпретация, *238*
Интрузивная брекчия, *373*
Интуиция, *212*
Исключение, *333*
Испытания, *309*
Истинностная таблица, *217*
Источник экспертных знаний, *521*
Исчисление, *176*
высказываний, *176*
нечетких ограничений, *476*
пропозициональное, *176*
утверждений, *176*

К

Кадр, *279*
Кардинальность, *425*
Карманный компьютер, *496*
Качество, *530*
Квадрат
магический, *272*
третьего порядка стандартный, *272*
Квантор, *181*; *216*; *436*

Always, *490*
Usually, *490*
всеобщности, *181*
нечеткий, *437*
общий, *216*
существования, *184*
частный, *216*
Клавиатура, *643*
Клавиша ввода, *560*
Класс, *95*; *147*
USER, *861*
абстрактный, *865*
конкретный, *865*
примитивный, *867*
Ключевое слово
inherit, *889*
slot, *563*
active, *857*
build, *198*
crlf, *586*
defrule, *575*
inherit, *943*
is-a, *872*
multislot, *563*
name, *874*
salience, *704*
Когнитивный процессор, *51*
Когнитология, *47*
Количественный показатель, *216*
Команда, *560*
active-duplicate-instance, *922*
active-message-modify-instance, *922*
active-modify-instance, *922*
agenda, *577*
apropos, *841*
assert, *564*
batch, *837*
bload-instances, *942*
browse-classes, *867*
bsave-instances, *942*
clear, *584*
clear-focus-stack, *730*
dribble-off, *839*
dribble-on, *839*
duplicate, *569*
duplicate-instance, *922*
exit, *560*
facts, *565*; *567*
focus, *728*
halt, *729*
instances, *854*
list-defclasses, *866*
list-deffacts, *582*
list-deffunctions, *799*
list-defgenerics, *830*
list-defglobals, *804*
list-definstances, *858*
list-defmessage-handlers, *889*
list-defmethods, *829*
list-defrules, *581*
list-deftemplates, *582*
list-focus-stack, *729*
load, *590*
load-instances, *942*
make-instance, *853*
matches, *752*
message-duplicate-instance, *922*
message-modify-instance, *921*
modify, *568*
modify-instance, *920*
override-next-handler, *903*
ppdefclass, *869*
ppdeffacts, *582*
ppdeffunction, *799*
ppdefgeneric, *830*
ppdefglobal, *804*
ppdefinstances, *858*
ppdefmessage-handler, *889*
ppdefmethod, *829*
ppdefrule, *582*
ppdeftemplate, *582*
preview-generic, *818*
preview-send, *914*
printout, *585*
refresh, *579*
remove-break, *590*
reset, *572*
restore-instances, *942*
retract, *568*
return, *733*
run, *576*
save, *592*

save-instances, *942*
send, *854*
set-break, *589*
set-fact-duplication, *951*
show-breaks, *590*
show-defglobals, *804*
system, *836*
undefclass, *869*
undeffacts, *583*
undeffunction, *799*
undefgeneric, *830*
undefglobal, *804*
undefinstances, *858*
undefmessage-handler, *889*
undefmethod, *829*
undefrule, *583*
undeftemplate, *583*
unwatch, *571*; *856*
watch, *570*; *828*
Комбинаторный взрыв, *105*
Комбинация
взвешенная, *368*
логическая, *368*
Компенсация колебаний видеокамеры, *496*
Компилятор, *142*
Композиция, *455*
дизъюнктивная, *486*
конъюнктивная, *486*
условная, *486*
Компонент экспертной системы, *552*
Компьютерная лингвистика, *459*
Конец списка операторов, *155*
Конкретность, *395*
Коннекционизм, *103*
Консеквент, *74*; *214*
Консенсус, *422*
Конструкция, *562*
defclass, *852*
deffacts, *571*
deffunction, *788*
defgeneric, *811*
defglobal, *802*
definstances, *857*
defmessage-handler, *882*
defmethod, *811*
defmodule, *717*
deftemplate, *562*
deftemplate подразумеваемая, *563*
deftemplate явная, *563*
Контекст, *372*
Континуум, *446*
Контур, *195*
Конфликт
между правилами, *520*
свидетельств, *430*
Концентрация, *451*
Коэффициент
весовой, *106*; *195*
достоверности, *365*; *399*; *403*; *404*; *950*
ослабления, *410*
правдоподобия эффективный, *360*
Культура, *523*
конечного пользователя, *523*
корпоративная, *523*
Купол, *377*

Л

Лексема, *142*; *556*
Литерал, *242*
Логика, *127*; *169*
бесконечнозначная, *467*
двухзначная, *434*; *466*
дедуктивная, *214*
неформальная, *127*
нечеткая, *295*; *297*; *358*; *432*; *466*
нечеткая FL, *485*
предикатов, *181*
предикатов первого порядка, *181*
пропозициональная, *181*
символическая, *128*; *175*
трехзначная, *466*
унарная, *467*
формальная, *127*; *128*; *172*
Логическое имя устройства, *585*
Локализация неисправностей, *708*

М

Масса, *417*
комбинированная, *422*
Матрица, *257*; *373*
переходов обычная, *340*
состояний, *337*

установившихся состояний, *339*
Машина
логического вывода, *35*; *52*; *69*; *527*; *552*
марковская скрытая, *143*
обратного логического вывода, *978*
Медно-порфировая руда, *373*
Мера
доверия, *405*; *424*
доверия к свидетельству, *417*
недоверия, *405*
Метазнания, *137*
Метаметодология, *535*
Метаобъяснение, *46*
Метаправило, *46*; *280*
Метасимвол, *237*
Метаязык, *139*
Метод
выработки и проверки, *212*; *274*
максимума, *477*
моментов, *477*
планирования, формирования и проверки, *274*
формирования и проверки, *274*
Методология разработки программного обеспечения, *505*
Метрика, *530*
Микроусовершенствование, *537*
Мир, *280*
Множество, *91*; *170*
адекватное, *180*
аксиом полное, *240*
выражений унифицируемое, *261*
меры нуль, *172*
нечеткое, *434*
нормализованное, *446*
одноэлементное, *180*; *415*
пустое, *172*
степенное, *190*
термов, *460*
универсальное, *171*
четкое, *434*
Моделирование процессуальных действий, *496*
Модель, *238*; *251*; *363*
каскадная, *535*
каскадная инкрементная, *537*
линейная, *538*
марковская, *297*
марковская скрытая, *82*; *287*
программирования, *54*
процесса, *535*
разработки на основе кодирования и исправления, *537*
спиральная, *538*
Модуль MAIN, *725*
Модус силлогизма, *216*
Монотонность, *278*
Мудрость, *138*
Мягкие вычисления, *433*

Н

Наведение видеокамеры, *495*
Назначение, *529*
Наименьшая верхняя грань, *491*
Наследование, *96*; *148*
единичное, *931*
множественное, *862*; *931*
Начало списка операторов, *155*
Неадекватность, *301*
Незнание, *418*; *427*
общее, *372*
Нейрон, *74*
Ненадежность, *304*
Неоднозначность, *301*
Неопределенность, *295*; *296*; *354*
воспроизводимая, *390*
Неполнота, *301*
Непротиворечивость, *240*
Нечеткий спецификатор, *436*; *485*
Новизна фактов, *396*
Нормализация, *428*; *453*
Нумерация законов успеха, *524*

О

Область
знаний, *36*
значений, *90*
определения, *90*; *234*
предметная, *29*; *36*
проблемная, *29*
Обнаружение неисправностей, *708*
Обобщение правил, *394*

Обобщенное дельта-правило, *112*
Обозначение
<<, *797*
>>, *797*
DFN, *797*
ED, *797*
GNC, *828*
MTH, *828*
Оболочка, *900*
Обоснование, *424*; *430*; *443*
логическое, *675*
Обработчик сообщений, *854*
after, *897*
around, *900*
before, *896*
get-, *855*
primary, *883*
put-, *855*
put-first-name, *922*
put-last-name, *922*
create, *915*
delete, *856*; *915*
direct-duplicate, *923*
init, *915*
message-duplicate, *923*
message-modify, *922*
собственный, *882*
Образ элементов, *90*
Обратная косая черта, *558*
Обучение, *104*
индуктивное, *114*
машинное, *46*
Объединение, *172*
множеств, *448*
резко выраженное, *450*
Объект, *40*; *145*
данных, *92*
отдельный, *148*
Ограничение
and, *628*
not, *627*
or, *627*
встроенное, *760*
запроса, *824*
нечеткое, *474*
поля, *626*
соединительное, *626*
эластичное, *474*
Ограничение поля
для возвращаемого значения, *656*
предикативное, *654*
Онтология, *122*; *137*
Операнд, *631*
Оператор отсечения, *152*
Операция
sup, *491*
дизъюнкции отрицаний, *180*
исключительного ИЛИ, *107*
композиции, *455*
конъюнкции отрицаний, *181*
логики Лукашевича, *467*
ортогональной суммы, *422*
переключения, *558*
прикладная, *92*
соединения, *448*
соприкосновения, *448*
Определение
базовой структуры знаний, *543*
конфигурации, *58*
стратегии приобретения знаний, *543*
требований высокого уровня, *541*
требований к системе, *543*
этапности решения задач, *541*
Опровержение, *240*; *247*
по методу резолюции, *247*
Осуществление процедур проверки, *547*
Отделитель имени модуля, *718*
Отметка временна́я, *74*; *396*
Отношение, *453*
нечеткое, *453*; *455*
ограничивающее, *490*
Отображение, *90*; *454*
Отсечение, *333*
Отступ, *868*
Оценка
активности фондовой биржи, *495*
важности источника знаний, *542*
вероятности, *486*
возможности, *486*
доступности источника знаний, *542*
истинностная, *486*
качества экспертной системы, *532*

осуществимости, *541*
результатов, *547*; *548*
Ошибка, *301*
в цепи логического вывода, *528*
второго рода, *301*
логическая, *130*
машины логического вывода, *527*
определения границ незнания, *528*
первого рода, *301*
семантическая, *524*
синтаксическая, *527*
систематическая, *304*
случайная, *304*
эксперта, *521*
Ошибочное обращение посылки и заключения, *227*

П

Память
ассоциативная, *109*
долговременная, *50*
кратковременная, *51*
отказоустойчивая, *109*
продукционная, *71*
рабочая, *69*
Пара, *150*
атрибут–значение, *150*
Парадокс
ворона, *298*
игры, *520*
лжеца, *177*
Параметр, *631*
local, *836*
visible, *836*
с подстановочным символом, *798*
функции, *561*
Первый момент инерции, *481*
Перебор с возвратами, *99*; *157*
Перегрузка функции, *831*
Переменная, *557*; *593*
бинарная, *454*
глобальная, *802*; *843*
лингвистическая, *459*
логическая, *176*
локальная, *802*
многозначная, *757*
области определения, *181*
свободная, *234*
связанная, *234*; *594*
случайная, *482*
Пересечение, *171*
множеств, *448*
резко выраженное, *450*
Переход, *199*
Персептрон, *111*
Петля, *195*
План, *274*
рабочий, *541*
Планирование, *541*
процессов передачи управления, *543*
расписаний движения автобусов, *496*
Пластичность, *110*
Побочный эффект, *632*
Повышение безопасности ядерных реакторов, *497*
Погрешность измерения, *301*
Подготовка
отчета, *548*
отчетов по результатам проверки, *547*
пользовательского интерфейса, *544*
пользовательской документации, *546*
распечаток программ, *546*
рекомендаций, *547*
руководств по инсталляции и эксплуатации, *546*
спецификаций проекта, *545*
Поддержание достоверности на основе предположений, *280*
Подкласс, *96*; *148*; *861*
Подмножество, *170*; *448*
нечеткое второго типа, *447*
нечеткое первого типа, *446*
нечеткое типа n, *447*
строгое, *171*; *448*
Подстановочный символ
однозначный, *600*
многозначный, *608*; *757*
Подход
декларативный, *94*
к программированию, *54*
непроцедурный, *94*
объектно-ориентированный, *94*

Подцель, *45*; *98*; *152*
Познание, *50*
Поиск
  в глубину, *153*
  в ширину, *153*
Поле, *556*
  числовое, *556*
Полиморфизм, *884*
Полнота, *240*
Полуразрешимость, *239*
Полуширина, *442*
Пользовательский интерфейс, *69*
Порода
  вулканическая, *373*
  порфировая, *372*
Порядок языка логики, *241*
Последовательность знаков
  <==, *579*; *857*
  ==>, *579*; *857*
Последствия, *344*
Постулат, *236*
Посылка, *169*
  большая, *215*
  меньшая, *215*
Правдоподобие, *342*; *424*
  достаточности, *345*
Правдоподобные отношения, *369*
Правило, *39*
  sum-areas, *638*
  sum-rectangles, *638*
  активизированное, *74*
  избыточное, *398*
  квантификации, *486*
  комбинирования Демпстера, *422*
  композиции, *485*
  композиции max-min, *473*
  композиционное, *491*
  логики, *129*
  модификации, *485*
  модус поненс, *221*
  модус поненс обобщенное, *491*
  модус толленс, *227*
  недостающее, *399*
  обеспечивающее полноту опровержения, *247*
  оценки, *486*
  подстановки, *78*; *230*
  преобразования, *485*
  продукционное, *50*; *52*
  реализованное, *74*
  семантическое, *463*
  синтаксическое, *462*
  универсальной конкретизации, *233*
  цепное, *228*
Правильность, *301*
Правильный вопрос, *268*
Предварительная функциональная компоновка, *541*
Предел
  допустимого диапазона верхний, *987*
  допустимого диапазона нижний, *987*
  рабочего диапазона верхний, *987*
  рабочего диапазона нижний, *987*
Предикат, *215*
  нечеткий, *483*
Предложение, *140*
  закрытое, *177*
  открытое, *177*
Представление
  данных, *128*
  знаний, *128*
Префикс
  get-, *855*
  HND, *887*
  MSG, *887*
  put-, *855*
  мета, *137*
Прецедент, *131*
Приведение к абсурду, *244*
Приглашение CLIPS, *559*
Приз Лебнера, *28*
Приложение управления производственными процессами, *496*
Пример сеанса, *560*
Принцип
  безразличия, *418*
  разделяй и властвуй, *45*
  расширения, *465*
  следования, *490*
Принятие антецедента, *221*

Приобретение
знаний, *371*
знаний автоматизированное, *198*
Приоритет
метода, *817*
неявный, *393*
явный, *393*
Присваивание
вероятностей в безнадежном положении, *329*
вероятности основное, *417*
основное, *417*
Причинная ассоциация, *277*
Пробел, *557*
Проблема
доставки, *93*; *517*
окружения, *279*
поддержания достоверности, *280*
Проверка, *546*
готовности к испытаниям, *546*
условий останова, *75*
Проверка на растекание, *478*
Проверка ограничений
динамическая, *697*
статическая, *696*
Прогноз, *46*
Программа
GPS, *97*
Logic Theorist, *96*
последовательная, *85*
процедурная, *85*
текущего контроля, *988*
Программирование
алгоритмическое, *54*
декларативное, *100*
императивное, *86*
логическое, *57*; *169*
на основе индукции, *102*
непроцедурное, *86*
объектно-ориентированное, *95*; *552*
основанное на правилах, *551*
процедурное, *551*
функциональное, *89*
Программотехника, *529*
Проект
ответственный, *521*
программы нисходящий, *87*
Проектирование
баз данных, *128*
восходящее, *90*
нисходящее, *90*
программного обеспечения, *496*
Проекция, *90*; *456*
цилиндрическая, *458*
Произведение
max-min, *456*
max-min матричное, *456*
декартово, *454*
множеств, *449*
ограниченное, *450*
Производство полупроводников, *496*
Пространство
абстрактное, *374*
выборочное, *312*
задач, *201*
состояний, *199*
Протеомика, *120*
Противоречивость правил, *393*
Противоречие, *178*
Протокол, *839*
Прототип, *166*
Процедура
if-added, *164*
if-needed, *163*
if-removal, *164*
именования, *92*
принятия решений, *217*
Процедурное вложение, *163*
Процесс
в марковской цепи, *340*
принятия решений марковский, *285*
согласования, *737*
стохастический, *336*
Путь, *194*

Р

Равенство множеств, *447*
Разграничитель, *556*
Разность
множеств, *189*
множеств симметрическая, *189*
ограниченная, *450*

Разработка
двукратная, *537*
кода, *546*
подробной функциональной компоновки, *543*
пользовательского интерфейса, *545*
структур управления подробная, *544*
структуры проекта, *545*
Разрешение конфликтов, *51*; *75*
Разрешимость, *238*
Разъяснение, *490*
Рамки различения, *417*
Распознавание, *496*
Распорядительное следование, *490*
Распределение
вероятностей, *482*
возможностей, *482*
Распространение
встречное, *107*
обратное, *107*
Рассуждения, *134*
аутоэпистемические, *212*; *281*
восходящие, *266*
временные, *335*
гипотетические, *41*
глубокие, *250*
косвенные, *227*
на основе здравого смысла, *281*; *466*
на основе прецедентов, *131*
на основе свидетельств, *414*
немонотонные, *212*
нестрогие, *300*; *390*
нечеткие, *468*
нисходящие, *266*
отменяемые, *282*
по аналогии, *212*; *271*
по умолчанию, *280*
поверхностные, *249*
приближенные, *295*; *468*; *476*
применяемые по умолчанию, *212*
строгие, *297*
Растворение, *452*
Ребро, *145*; *194*
Регламентация внутренней структуры фактов, *544*
Регулирование
расхода топлива в автомобиле, *497*
скорости автомобили, *496*
Режим верхнего уровня, *560*
Резольвента, *244*
Резолюция, *241*
Релаксация, *74*; *578*
Реляционное соединение, *456*
Решатель задач
универсальный, *97*
Решетка, *195*
AND–OR, *207*
AND–OR–NOT, *208*
решений, *195*
Ряд степеней доверия, *423*

C

Свидетельство, *263*; *342*
определенное, *370*
полное, *349*
простое, *354*
сложное, *354*
хрупкое, *528*
частичное, *349*
Свойство, *148*
Связка, *216*
NAND, *180*
NOR, *180*
Связывание, *594*
с шаблоном, *596*
Связь, *145*; *194*
A-KIND-OF, *147*
AKO, *147*
ARE, *148*
CAUSE, *148*
IS-A, *147*; *148*
сильная, *510*
слабая, *510*
Сейсмические исследования, *329*
Секционирование базы знаний, *717*
Семантика, *128*; *139*
Семиотика, *47*; *511*
Сеть
ассоциативная, *145*
логического вывода, *100*; *363*; *373*
нейронная, *103*; *496*
пропозициональная, *144*

с обратным распространением, *112*
семантическая, *144*
семантическая секционированная, *374*
соединений, *745*
шаблонов, *741*
Сигнатура, *811*
Силлогизм, *169*; *172*
категорический, *215*
Символ, *556*
eof, *644*
exit, *560*
MAIN::, *583*
nil, *93*
начальный, *139*
нетерминальный, *139*
терминальный, *139*
Синапс, *111*
Синтаксис, *139*
Система
Semantic Web, *158*
Web семантическая, *158*
автоматизированная, *29*
детерминированная, *310*
интеллектуальная, *29*; *390*
информационная, *506*
кондиционирования воздуха, *495*
логическая, *235*
монотонная, *278*
недетерминированная, *310*
нейронная искусственная, *103*; *106*
немонотонная, *278*
обозначений бесквантерная, *242*
опознавания свой–чужой, *419*
основанная на знаниях, *35*; *40*
прогнозирования, *496*
продукционная, *52*; *77*
совершения открытий, *33*
счисления единичная, *240*
формирования рассуждений автоматизированная, *169*
экспертная, *33*; *35*; *100*
экспертная нечеткая, *466*
экспертная, основанная на знаниях, *35*
Система обозначений, *554*
Скобка
квадратная, *554*
круглая, *560*
угловая, *139*
фигурная, *91*
Сколемовская константа, *256*
Скрытый слой, *113*
Слот, *162*; *561*
многозначный, *563*
однозначный, *563*
Смысл, *463*
контекстно-зависимый, *282*
Событие, *309*; *312*; *327*
достоверное, *314*
невозможное, *314*
сложное, *312*
элементарное, *312*
События
взаимно независимые, *319*
взаимоисключающие, *314*
временные, *335*
независимые, *319*; *326*
попарно независимые, *319*
стохастически независимые, *319*
Совещание по разработке авиакосмической техники, *521*
Совместимость, *438*
правил, *393*
Совокупность, *313*
Согласование, *75*
Согласованное количество круглых скобок, *560*
Соглашение по охране интеллектуальной собственности, *508*
Создание
прототипа ускоренное, *47*
системы классификации знаний, *543*
экземпляра, *853*
Сокрытие информации, *95*
Сомнение, *427*
Сомнительность, *427*
Сообщение
direct-modify, *921*
print, *855*
Сопоставление
с шаблонами, *740*
с шаблоном, *155*
Сортировка списка полей, *841*

Составление
документа с описанием системы, *546*
начального плана проверки, *544*
подробного плана проверки, *545*
руководства пользователя, *543*
Состояние, *87*; *199*; *344*
Список, *92*
правил рабочий, *70*; *576*
пустой, *93*
семантический, *511*
фактов, *552*
Сплавление данных, *399*
Среда, *414*
Среднее время наработки на отказ, *530*
Средний аттестационный балл, *402*
Средства
обработки информации, *506*
экстралогические, *242*
Средство
инструментальное, *68*
объяснения, *41*; *69*
приобретения знаний, *70*
Сталелитейный завод, *496*
Стек, *612*
фокусов, *729*
Степень
детализации, *52*
доверия, *342*; *404*
доверия вероятная, *371*
доверия возможная, *370*
доверия невозможная, *370*
доверия несомненная, *370*
доверия правдоподобная, *370*
недоверия, *404*; *419*
нехватки недоверия, *419*
отсутствия доверия, *419*
подтверждения, *404*
правдоподобия, *404*
принадлежности, *435*
Стереотип, *161*
Стратегия
управления, *80*; *396*
управления лексикографическая, *396*
Стрелка, *576*
двойная, *141*
Строгая типизация, *545*
Строка, *140*; *556*; *557*
пустая, *81*
управляющая, *645*
Структура, *195*
знаний, *159*
знаний глубокая, *159*
знаний динамическая, *377*
знаний поверхностная, *159*
знаний статическая, *377*
управления, *62*
Структурное подобие, *83*; *741*
Субъект, *215*
Сумма
вероятностная, *450*
ограниченная, *450*
ортогональная, *422*
прямая, *422*
Суперкласс, *96*; *148*; *861*
Супремум, *491*
Схема, *160*
концептуальная, *161*
сенсорно-двигательная, *160*
Сценарий, *162*

T

Тавтология, *173*
Таксономия, *85*; *363*
Текстура, *372*
Текущий фокус, *728*
Тело цикла, *781*
Теорема, *175*; *236*
Байеса, *326*
Теория
байесовская с выпуклыми множествами, *494*
вероятностей, *295*
вероятностей объективная, *315*
Демпстера–Шефера, *297*; *414*
доверительных функций, *425*
Кайберга, *494*
нечетких множеств, *297*
Терм
nil, *245*
null, *245*
Poss, *482*
первичный, *463*

Термин
команда, *561*
функция, *561*
Тест Тьюринга, *27*
Технология
SMART, *531*
информационная, *506*
медицинская, *496*
Тип
данных примитивный, *556*
поля, *556*
Товарищеская непредвзятость, *523*
Точка
останова, *589*
пересечения, *440*
Точность, *301*
Трансформация, *340*
Триплет, *149*
Троеточие, *170*
Тройка объект–атрибут–значение, *149*; *950*

У

Узел, *145*; *194*
двухвходовой, *745*
дочерний, *194*; *955*
запрашивающий, *365*
корневой, *194*; *955*
листовой, *194*; *955*
одновходовой, *741*
ответа, *955*
принятия решений, *955*
родительский, *194*; *955*
родовой, *148*
соединения, *745*
терминальный, *742*
шаблона, *741*
Узкое место в процессе приобретения знаний, *43*
Умозаключение, *129*
Универсальный решатель задач, *50*
Унификатор, *261*
наиболее общий, *261*
Унификация, *99*; *259*
Упорядочение шаблонов, *755*
Управление
вертолетом, *496*
двигателем автомобиля, *496*
двигателем пылесоса, *496*
задней подсветкой видеокамеры, *496*
знаниями, *506*
информацией, *506*
исполнением правил, *726*
лифтом, *497*
ресурсами, *541*
роботами, *495*
системами подземного транспорта, *496*
стиральными машинами, *496*
Уравнение присваивания значения возможности, *483*
Условный элемент, *575*
and явный, *659*
and, *588*; *628*; *662*
and неявный, *659*
exists, *667*
forall, *670*
logical, *673*; *878*
not, *664*
or, *662*
or явный, *659*
test, *652*; *758*
шаблона, *575*
Устройство
ввода данных стандартное, *643*
вывода стандартное, *585*
Утверждение, *177*; *236*; *365*
категорическое, *215*
контингентное, *189*
нечеткое, *436*

Ф

Факт, *135*; *561*
initial-fact, *572*
безусловно обоснованный, *677*
дублирующийся, *566*
зависимый, *675*; *677*
непостоянный, *756*
с конструкцией deftemplate, *563*
совместимый, *281*
упорядоченный, *563*
управляющий, *650*
Фасет слота, *853*
Фигура силлогизма, *216*

Флажок, *564*
формата, *645*
Фокус-группа, *523*
Форма
дизъюнктивная, *243*
инфиксная, *631*
каноническая, *483*
контрапозитивная, *229*
нормальная, *242*
нормальная конъюнктивная, *242*
обратная, *229*
отрицательная по качеству, *216*
предваренная, *257*
префиксная, *631*
противоположная, *229*
с логическими выражениями, *242*; *243*
с хорновскими выражениями, *242*
с шансами и правдоподобием, *345*
стандартная, *215*
утвердительная по качеству, *216*
функциональная, *92*
Формирование
логических выводов, *129*; *134*
обратного логического вывода, *262*
прямого логического вывода, *262*
рассуждений непосредственное, *221*
Формула
выполнимая, *238*
действительная, *238*
доказанная, *238*
невыполнимая, *238*
недействительная, *238*
несовместимая, *238*
правильно построенная, *235*
совместимая, *238*
Фрагмент, *50*
Фрейм, *100*; *162*
действия, *166*
причинных знаний, *166*
ситуативный, *166*
Функция, *90*
∗, *1025*
∗∗, *1026*
+, *1025*
−, *1025*
/, *1025*
<, *1051*
<=, *1051*
<>, *1051*
=, *1051*
>, *1051*
>=, *1051*
abs, *1024*
acos, *1055*
acosh, *1056*
acot, *1056*
acoth, *1056*
acsc, *1056*
acsch, *1056*
agenda, *1023*
and, *1049*
any-instancep, *926*; *1041*
apropos, *1037*
asec, *1056*
asech, *1056*
asin, *1056*
asinh, *1056*
assert, *1039*
assert-string, *1039*
atan, *1056*
atanh, *1056*
batch, *1037*
batch*, *1037*
Bel, *424*
bind, *638*; *1051*
bload, *1037*
break, *779*; *787*; *1051*
browse-classes, *1027*
bsave, *1038*
build, *1054*
call-next-handler, *895*; *1033*
call-next-method, *1034*
call-specific-method, *1034*
check-syntax, *1054*
class, *936*; *1041*
class-abstractp, *1027*
class-existp, *1027*
class-reactivep, *1027*
class-slots, *936*; *1028*
class-subclasses, *1028*
class-superclasses, *1028*
clear, *1038*

clear-focus-stack, *1023*
close, *642*; *1045*
conserve-mem, *1046*
cos, *1056*
cosh, *1056*
cot, *1056*
coth, *1056*
create$, *1047*
csc, *1057*
csch, *1057*
defclass-module, *1028*
deffacts-module, *1030*
deffunction-module, *1030*
defgeneric-module, *1031*
defglobal-module, *1031*
definstances-module, *1032*
defrule-module, *1036*
deftemplate-module, *1037*
deg-grad, *1025*
deg-rad, *1025*
delayed-do-for-all-instances, *929*; *1041*
delete-instance, *1042*
delete-member$, *1048*
delete-member$, *936*
delete$, *1047*
dependencies, *1039*
dependents, *1039*
describe-classes, *1028*
direct-slot-delete$, *1042*
direct-slot-insert$, *1042*
direct-slot-replace$, *1042*
div, *1025*
do-for-all-instances, *929*; *1042*
do-for-instance, *929*; *1042*
dribble-off, *1026*
dribble-on, *1027*
duplicate, *1039*
dynamic-get, *1042*
dynamic-put, *1042*
eq, *1049*
eval, *1054*
evenp, *1049*
exit, *1038*
exp, *1026*
expand$, *1053*
explode$, *1048*
explode$, *648*
fact-existp, *1040*
fact-index, *1040*
fact-relation, *1040*
fact-slot-names, *1040*
fact-slot-value, *1040*
facts, *1040*
fetch, *1055*
find-all-instances, *928*; *1042*
find-instance, *928*; *1042*
first$, *1048*
float, *1025*
floatp, *1049*
focus, *1023*
format, *644*; *1045*
funcall, *1046*
gensym, *892*; *1046*
gensym*, *967*; *1046*
get-auto-float-dividend, *1038*
get-class-defaults-mode, *1028*
get-current-module, *719*; *1035*
get-defclass-list, *1028*
get-deffacts-list, *1030*
get-deffunction-list, *799*; *1030*
get-defgeneric-list, *830*; *1031*
get-defglobal-list, *804*; *1032*
get-definstances-list, *858*; *1032*
get-defmesage-handler-list, *889*
get-defmessage-handler-list, *1033*
get-defmethod-list, *829*; *1034*
get-defmodule-list, *1035*
get-defrule-list, *1036*
get-deftemplate-list, *1037*
get-dynamic-constraint-checking, *697*; *1038*
get-fact-duplication, *1040*
get-fact-list, *1040*
get-focus, *731*; *1023*
get-focus-stack, *731*; *1023*
get-function-restrictions, *1046*
get-incremental-reset, *1036*
get-method-restrictions, *1034*
get-profile-percent-threshold, *1053*
get-reset-globals, *1032*
get-salience-evaluation, *1023*
get-sequence-operator-recognition, *1054*

get-static-constraint-checking, *696*; *1038*
get-strategy, *1024*
grad-deg, *1026*
halt, *780*; *787*; *1024*
if, *779*; *1051*
implode$, *1048*
init-slots, *1043*
insert$, *1048*
instance-address, *1043*
instance-addressp, *1043*
instance-existp, *1043*
instance-name, *1043*
instance-name-to-symbol, *1043*
instance-namep, *1043*
instancep, *1043*
instances, *1043*
integer, *1025*
integerp, *1049*
length, *1046*
length$, *1048*
lexemep, *1050*
list-defclasses, *1028*
list-deffacts, *1030*
list-deffunctions, *1030*
list-defgenerics, *1031*
list-defglobals, *1032*
list-definstances, *1033*
list-defmessage-handlers, *1033*
list-defmethods, *1034*
list-defmodules, *1035*
list-defrules, *1036*
list-deftemplates, *1037*
list-focus-stack, *1024*
list-watch-items, *1027*
load, *1038*
load*, *1038*
load-facts, *835*; *1040*
load-instances, *1044*
log, *1026*
loop-for-count, *779*; *784*; *1052*
lowcase, *1054*
make-instance, *1044*
matches, *1036*
max, *359*; *1025*
mem-requests, *1046*
mem-used, *1046*
member$, *1048*
message-handler-existp, *1033*
min, *358*; *1025*
mod, *657*
mod , *1026*
modify, *1041*
multifieldp, *1050*
neq, *1050*
next-handlerp, *1033*
next-methodp, *1035*
not, *1050*
nth$, *1048*
numberp, *1050*
oddp, *1050*
open, *640*; *1045*
options, *1038*
or, *1050*
override-next-handler, *1033*
override-next-method, *1035*
π, *1026*
pointerp, *1050*
pop-focus, *731*; *1024*
ppdefclass, *1028*
ppdeffacts, *1030*
ppdeffunction, *1031*
ppdefgeneric, *1031*
ppdefglobal, *1032*
ppdefinstances, *1033*
ppdefmessage-handler, *1034*
ppdefmethod, *1035*
ppdefmodule, *1035*
ppdefrule, *1036*
ppdeftemplate, *1037*
ppinstance, *1044*
preview-generic, *1031*; *1035*
preview-send, *1034*
print-region, *1055*
printout, *643*; *1045*
profile, *1053*
profile-info, *1053*
profile-reset, *1053*
progn, *1052*
progn$, *779*; *786*; *1052*
R, *287*
rad-deg, *1026*
random, *839*; *1047*

read, *639*; *1045*
readline, *646*; *1045*
refresh, *1036*
refresh-agenda, *1024*
release-mem, *1046*
remove, *1045*
remove-break, *1036*
rename, *1045*
replace-member$, *1049*
replace$, *1048*
reset, *1038*
rest$, *1049*
restore-instances, *1044*
retract, *1041*
return, *790*; *1052*
round, *1026*
run, *1024*
save, *1038*
save-facts, *835*; *1041*
save-instances, *1044*
sec, *1057*
sech, *1057*
seed, *839*; *1047*
send, *1044*
set-auto-float-dividend, *1038*
set-break, *1036*
set-class-defaults-mode, *1028*
set-current-module, *719*; *1035*
set-dynamic-constraint-checking, *697*; *1039*
set-fact-duplication, *1041*
set-incremental-reset, *1036*
set-profile-percent-threshold, *1053*
set-reset-globals, *805*; *1032*
set-salience-evaluation, *1024*
set-sequence-operator-recognition, *1054*
set-static-constraint-checking, *696*; *1039*
set-strategy, *1024*
setgen, *1047*
show-breaks, *1036*
show-defglobals, *1032*
sin, *1057*
sinh, *1057*
slot-allowed-values, *1028*
slot-cardinality, *1029*
slot-default-value, *1029*
slot-delete$, *1044*
slot-direct-accessp, *1029*
slot-existp, *1029*
slot-facets, *1029*
slot-initablep, *1029*
slot-insert$, *1044*
slot-publicp, *1029*
slot-range, *1029*
slot-replace$, *1044*
slot-sources, *1029*
slot-types, *1029*
slot-writeablep, *1029*
sort, *841*; *1047*
sqrt, *1026*
str-cat, *1054*
str-compare, *1054*
str-index, *1055*
str-length, *1055*
string-to-field, *840*; *1055*
stringp, *1050*
sub-string, *1055*
subclassp, *1030*
subseq$, *1049*
subsetp, *1049*
superclassp, *1030*
switch, *779*; *1052*
sym-cat, *1055*
symbol-to-instance-name, *1044*
symbolp, *1050*
system, *1039*
tan, *1057*
tanh, *1057*
time, *703*; *1047*
timer, *1047*
toss, *1055*
type, *1031*
undefclass, *1030*
undeffacts, *1030*
undeffunction, *1031*
undefgeneric, *1031*
undefglobal, *1032*
undefinstances, *1033*
undefmessage-handler, *1034*
undefmethod, *1035*
undefrule, *1037*
undeftemplate, *1037*

unmake-instance, *1044*
unwatch, *1027*
upcase, *1055*
watch, *1027*
while, *779*; *781*; *1053*
активизации, *107*
внешняя, *651*
вознаграждения, *287*
выпуклая, *494*
доверительная, *424*
запроса, *928*; *929*
заранее определенная, *651*
комбинирующая, *411*
определяемая пользователем, *558*; *651*; *801*
перегруженная, *811*
перехода между состояниями, *286*
предикативная, *182*; *642*; *651*
примитивная, *92*
принадлежности, *435*
пропозициональная, *233*
различительная, *434*
сигмоидальная, *107*
сколемовская, *256*
совместимости, *435*
универсальная, *811*; *867*
характеристическая, *434*; *435*

Х

Характеристики экспертной системы, *43*
Хранилище данных, *87*; *129*

Ц

Цель, *45*; *94*; *152*
Центр масс, *480*
Цепь
логического вывода, *262*
обратная, *262*
причинная, *250*
прямая, *262*
рассуждений, *214*
Цикл, *195*
выборка–выполнение, *75*
жизненный, *533*
распознавание–действие, *75*; *84*
ситуация–действие, *75*
ситуация–отклик, *75*
Цилиндрическое расширение, *458*

Ч

Частичное соответствие, *740*
Часть
левая, *71*; *92*
правая, *74*; *92*
условная, *71*
шаблона, *71*
Часть памяти
левая, *745*
правая, *743*
Число, *556*
с плавающей точкой, *556*
трансфинитное, *467*
целое, *556*
Член силлогизма
младший, *215*
средний, *216*
старший, *215*

Ш

Шаблон, *71*; *575*
initial-object, *881*
модели правил, *285*
наиболее конкретный, *756*
объекта, *869*
управляющий, *649*
Шанс, *342*
Ширина
интервала, *443*
общая, *443*
Штрих, *180*

Э

Эвристика, *42*; *131*; *159*; *212*
Эквивалентность, *229*
логическая, *179*
материальная, *179*
Экземпляр, *95*; *559*
подстановочный, *259*
Эксперт, *34*
Экспорт, *723*
Элемент, *170*
instances, *856*
slots, *856*

выборки, *312*
условный, *71*
фокальный, *420*
Эмерджентность, *117*
Эмуляция, *33*
Эпистемология, *132*
Этап
верификации знаний, *546*
определения знаний, *542*
оценки системы, *547*
подробного проектирования, *545*
проектирования знаний, *544*
разработки кода и отладки, *546*
усовершенствования крупный, *537*
усовершенствования мелкий, *537*

Я

Язык, *67*
CLIPS, *552*
COOL, *96*; *553*
базовый, *519*
второго порядка, *241*
первого порядка, *241*
разметки расширяемый, *158*
чувствительный к регистру, *557*

Научно-популярное издание

Джозеф Джарратано, Гари Райли

Экспертные системы: принципы разработки и программирование

4-е издание

Литературный редактор *И.А. Попова*
Верстка *А.Н. Полинчик*
Художественные редакторы *Е.П. Дынник*
Корректоры *Л.А. Гордиенко*, *О.В. Мишутина*

Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

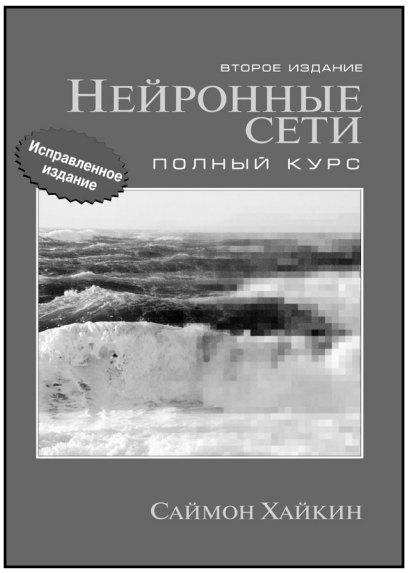
Подписано в печать 22.11.2006. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 92,9. Уч.-изд. л. 58,7.
Тираж 3000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

НЕЙРОННЫЕ СЕТИ ПОЛНЫЙ КУРС

2-Е ИЗДАНИЕ

Саймон Хайкин



www.williamspublishing.com

В книге рассматриваются основные парадигмы искусственных нейронных сетей. Представленный материал содержит строгое математическое обоснование всех нейросетевых парадигм, иллюстрируется примерами, описанием компьютерных экспериментов, содержит множество практических задач, а также обширную библиографию.

В книге также анализируется роль нейронных сетей при решении задач распознавания образов, управления и обработки сигналов. Структура книги очень удобна для разработки курсов обучения нейронным сетям и интеллектуальным вычислениям.

Книга будет полезна для инженеров, специалистов в области информатики, физиков и специалистов в других областях, а также для всех тех, кто интересуется искусственными нейронными сетями.

ISBN 5-8459-0890-6

в продаже

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ: СТРУКТУРЫ И СТРАТЕГИИ РЕШЕНИЙ СЛОЖНЫХ ПРОБЛЕМ, 4-Е ИЗДАНИЕ

Джордж Люгер



www.williamspublishing.com

в продаже

Данная книга посвящена одной из наиболее перспективных и привлекательных областей развития научного знания — методологии искусственного интеллекта (ИИ). В ней детально описываются как теоретические основы искусственного интеллекта, так и примеры построения конкретных прикладных систем. Книга дает полное представление о современном состоянии развития этой области науки. Подробно рассматриваются вопросы представления знаний при решении задач ИИ, логика решения этих задач, алгоритмы поиска, продукционные системы и машинное обучение. Эти вопросы остаются центральными в области искусственного интеллекта. В книге также представлены результаты новейших исследований, связанных с вопросами понимания естественного языка, обучения с подкреплением, рассуждения в условиях неопределенности, эмерджентных вычислений, автоматического доказательства теорем и решения задач ИИ на основе моделей. Большое внимание уделяется описанию реальных прикладных систем, построенных на принципах ИИ, и современных областей приложения этой области знаний. Помимо математических основ искусственного интеллекта в книге затронуты его философские аспекты. В последней части книги рассматриваются технологии программирования задач из области искусственного интеллекта на языках LISP и PROLOG. Книга будет полезна как опытным специалистам в области искусственного интеллекта, так и студентам и начинающим ученым.

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ СОВРЕМЕННЫЙ ПОДХОД

ВТОРОЕ ИЗДАНИЕ

С. Рассел
П. Норвиг



www.williamspublishing.com

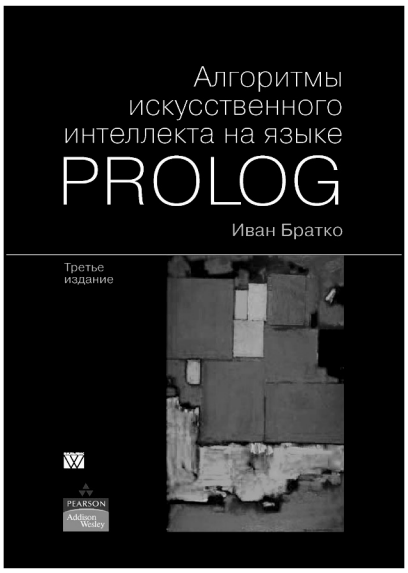
В книге представлены все современные достижения и изложены идеи, которые были сформулированы в исследованиях, проводившихся в течение последних пятидесяти лет, а также собраны на протяжении двух тысячелетий в областях знаний, ставших стимулом к развитию искусственного интеллекта как науки проектирования рациональных агентов. Теоретическое описание иллюстрируется многочисленными алгоритмами, реализации которых в виде готовых программ на нескольких языках программирования находятся на сопровождающем Web-узле. Книга предназначена для использования в базовом университетском курсе или в последовательности курсов по специальности. Применима в качестве основного справочника для аспирантов, специализирующихся в области искусственного интеллекта, а также будет небезынтересна профессионалам, желающим выйти за пределы избранной ими специальности.

ISBN 5-8459-0887-6

в продаже

АЛГОРИТМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА НА ЯЗЫКЕ PROLOG
3-Е ИЗДАНИЕ

Иван Братко



www.williamspublishing.com

Книга известного специалиста по программированию, в которой приведены основные сведения о языке Prolog, описан процесс разработки программ на Prolog и показано применение языка Prolog во многих областях искусственного интеллекта, включая решение задач и эвристический поиск, программирование в ограничениях, представление знаний и экспертные системы, планирование, машинное обучение, качественные рассуждения, обработка текста на различных языках и ведение игр. Книга содержит описание методов обработки многих важных структур данных, в том числе деревьев и графов. Задачи искусственного интеллекта представлены и разработаны до такой степени детализации, которая позволяет успешно реализовать их на языке Prolog и получить законченные программы.

ISBN 5-8459-0664-4

в продаже