

Алгоритмы
искусственного
интеллекта на языке

PROLOG

Третье издание

PROLOG

Programming for Artificial Intelligence

Third edition

IVAN BRATKO

Faculty of Computer and Information Science,
Ljubljana University
and J. Stefan Institute



Harlow, England • London • New York • Boston • San Francisco • Toronto • Sydney • Singapore • Hong Kong
Tokyo Seoul • Taipei • New Delhi • Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Алгоритмы искусственного интеллекта на языке PROLOG

Третье издание

ИВАН БРАТКО

Факультет компьютерных наук и информатики
Люблянского университета
и Институт Йозефа Штефана



Москва · Санкт-Петербург · Киев
2004

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

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

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Братко, Иван.

Б87 Алгоритмы искусственного интеллекта на языке PROLOG, 3-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс", 2004. — 640 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0664-4 (рус.)

В книге известного специалиста по программированию приведены основные сведения о языке Prolog, описан процесс разработки программ на этом языке 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 предназначена для тех, кто проходит обучение в области языка Prolog и искусственного интеллекта или интересуется этими перспективными направлениями. От читателя не требуется наличие знаний в области искусственного интеллекта. Не обязательна также значительная подготовка в области программирования.

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 Addison-Wesley UK.

Authorized translation from the English language edition published by Addison-Wesley Publishers Limited, Copyright © 2001 by Pearson Education Limited

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2004

Оглавление

Предисловие	18
Часть I. Язык Prolog	25
Глава 1. Введение в Prolog	26
Глава 2. Синтаксис и значение программ Prolog	45
Глава 3. Списки, операции, арифметические выражения	76
Глава 4. Использование структур: примеры программ	98
Глава 5. Управление перебором с возвратами	121
Глава 6. Ввод и вывод	136
Глава 7. Дополнительные встроенные предикаты	149
Глава 8. Стил и методы программирования	169
Глава 9. Операции со структурами данных	192
Глава 10. Усовершенствованные методы представления деревьев	215
Часть II. Применение языка Prolog в области искусственного интеллекта	227
Глава 11. Основные стратегии решения проблем	228
Глава 12. Эвристический поиск по заданному критерию	247
Глава 13. Декомпозиция задач и графы AND/OR	277
Глава 14. Логическое программирование в ограничениях	301
Глава 15. Представление знаний и экспертные системы	326
Глава 16. Командный интерпретатор экспертной системы	357
Глава 17. Планирование	383
Глава 18. Машинное обучение	408
Глава 19. Индуктивное логическое программирование	446
Глава 20. Качественные рассуждения	478
Глава 21. Обработка лингвистической информации с помощью грамматических правил	510
Глава 22. Ведение игры	532
Глава 23. Метапрограммирование	559
Приложение А. Некоторые различия между реализациями Prolog	590
Приложение Б. Некоторые часто используемые предикаты	592
Решения к отдельным упражнениям	595
Список литературы	611
Предметный указатель	619

Содержание

Предисловие	18
Часть I. Язык Prolog	25
Глава 1. Введение в Prolog	26
1.1. Определение отношений на основе фактов	26
1.2. Определение отношений на основе правил	30
1.3. Рекурсивные правила	35
1.4. Общие принципы поиска ответов на вопросы системой Prolog	39
1.5. Декларативное и процедурное значение программ	42
Резюме	43
Дополнительные источники информации	44
Глава 2. Синтаксис и значение программ Prolog	45
2.1. Объекты данных	45
2.1.1. Атомы и числа	46
2.1.2. Переменные	47
2.1.3. Структуры	48
2.2. Согласование	52
2.3. Декларативное значение программ Prolog	56
2.4. Процедурное значение	58
2.5. Пример: обезьяна и банан	62
2.6. Порядок предложений и целей	66
2.6.1. Опасность возникновения бесконечных циклов	66
2.6.2. Модификация программы путем переупорядочения предложений и целей	68
2.6.3. Объединение декларативного и процедурного представлений	72
2.7. Взаимосвязь между языком Prolog и логикой	73
Резюме	74
Глава 3. Списки, операции, арифметические выражения	76
3.1. Представление списков	76
3.2. Некоторые операции со списками	78
3.2.1. Проверка принадлежности к списку	79
3.2.2. Конкатенация	79
3.2.3. Добавление элемента	82
3.2.4. Удаление элемента	82
3.2.5. Подсписок	83
3.2.6. Перестановки	84
3.3. Запись в операторной форме	87
3.4. Арифметические выражения	92
Резюме	96
Глава 4. Использование структур: примеры программ	98
4.1. Выборка структурированной информации из базы данных	98
4.2. Методы абстрагирования данных	102
4.3. Моделирование недетерминированного конечного автомата	103
4.4. Консультант бюро путешествий	107
4.5. Задача с восемью ферзями	111
4.5.1. Программа 1	111

4.5.2. Программа 2	114
4.5.3. Программа 3	116
4.5.4. Заключительные замечания	119
Резюме	120
Глава 5. Управление перебором с возвратами	121
5.1. Предотвращение перебора с возвратами	121
5.1.1. Эксперимент 1	122
5.1.2. Эксперимент 2	123
5.2. Примеры использования оператора отсеечения	125
5.2.1. Вычисление максимума	125
5.2.2. Проверка принадлежности к списку, при которой возможно только одно решение	126
5.2.3. Добавление элемента к списку без дублирования	126
5.2.4. Классификация с разбивкой по категориям	127
5.3. Отрицание как недостижение цели	129
5.4. Проблемы, связанные с использованием отрицания и оператора отсеечения	132
Резюме	135
Дополнительные источники информации	135
Глава 6. Ввод и вывод	136
6.1. Обмен данными с файлами	136
6.2. Обработка файлов, состоящих из термов	139
6.2.1. Предикаты read и write	139
6.2.2. Вывод списков на внешнее устройство	141
6.2.3. Обработка файла, состоящего из термов	142
6.3. Манипулирование символами	143
6.4. Формирование и анализ атомов	144
6.5. Чтение программ	146
Резюме	147
Дополнительные сведения, касающиеся стандарта Prolog	148
Глава 7. Дополнительные встроенные предикаты	149
7.1. Проверка типа термов	149
7.1.1. Предикаты var, nonvar, atom, integer, float, number, atomic, compound	149
7.1.2. Решение числового ребуса с использованием предиката nonvar	151
7.2. Создание и декомпозиция термов: предикаты =.., functor, arg и name	156
7.3. Операторы сравнения и проверки на равенство различных типов	159
7.4. Операции с базой данных	161
7.5. Средства управления	164
7.6. Предикаты bagof, setof и findall	165
Резюме	167
Глава 8. Стиль и методы программирования	169
8.1. Общие принципы качественного программирования	169
8.2. Общий подход к разработке программ Prolog	171
8.2.1. Использование рекурсии	171
8.2.2. Обобщение	172
8.2.3. Использование графических схем	173
8.3. Стиль программирования	173
8.3.1. Некоторые правила хорошего стиля	174
8.3.2. Табличная организация длинных процедур	175
8.3.3. Комментирование	175
8.4. Отладка	176
8.5. Повышение эффективности	177
8.5.1. Повышение эффективности программы решения задачи с восемью ферзями	178

8.5.2. Повышение эффективности программы раскраски карты	179
8.5.3. Повышение эффективности конкатенации списков с использованием разностных списков	181
8.5.4. Оптимизация последнего вызова и накапливающие параметры	182
8.5.5. Моделирование массивов с помощью предиката arg	184
8.5.6. Повышение эффективности путем внесения в базу данных производных фактов	186
Резюме	190
Дополнительные источники информации	191
Глава 9. Операции со структурами данных	192
9.1. Сортировка списков	192
9.2. Представление множеств с помощью бинарных деревьев	197
9.3. Вставка и удаление в бинарном словаре	201
9.4. Отображение деревьев	206
9.5. Графы	208
9.5.1. Представление графов	208
9.5.2. Поиск пути	209
9.5.3. Поиск остовного дерева графа	211
Резюме	214
Дополнительные источники информации	214
Глава 10. Усовершенствованные методы представления деревьев	215
10.1. Двоично-троичный словарь	215
Вариант А	218
Вариант Б	218
Вариант В	218
10.2. AVL-дерево — приближенно сбалансированное дерево	221
Резюме	225
Дополнительные источники информации	225
Часть II. Применение языка Prolog в области искусственного интеллекта	227
Глава 11. Основные стратегии решения проблем	228
11.1. Вводные понятия и примеры	228
11.2. Поиск в глубину и итеративное углубление	232
11.3. Поиск в ширину	238
11.4. Анализ основных методов поиска	242
Резюме	245
Дополнительные источники информации	246
Глава 12. Эвристический поиск по заданному критерию	247
12.1. Поиск по заданному критерию	247
12.2. Применение поиска по заданному критерию для решения головоломки “игра в восемь”	256
12.3. Применение поиска по заданному критерию при планировании	261
12.4. Методы экономии пространства для поиска по заданному критерию	265
12.4.1. Потребность алгоритма A* в ресурсах времени и пространства	265
12.4.2. Метод IDA*	266
12.4.3. Метод RBFS	269
Резюме	275
Дополнительные источники информации	275
Глава 13. Декомпозиция задач и графы AND/OR	277
13.1. Представление задач в виде графов AND/OR	277
13.2. Примеры представлений в виде графа AND/OR	281
13.2.1. Представление в виде графа AND/OR задачи поиска маршрута	281
13.2.2. Задача с ханойской башней	282
13.2.3. Формулировка процесса игры в виде графа AND/OR	283
13.3. Основные процедуры поиска в графе AND/OR	284

13.4. Поиск в графе AND/OR по заданному критерию	289
13.4.1. Эвристические оценки и алгоритм поиска	289
13.4.2. Программа поиска	292
13.4.3. Пример отношений с определением задачи — поиск маршрута	298
Резюме	300
Дополнительные источники информации	300
Глава 14. Логическое программирование в ограничениях	301
14.1. Удовлетворение ограничений и логическое программирование	301
14.1.1. Удовлетворение ограничений	301
14.1.2. Решение задачи удовлетворения ограничений	303
14.1.3. Расширение Prolog для использования в качестве языка логического программирования в ограничениях	305
14.2. Применение метода CLP для обработки действительных чисел — CLP(R)	306
14.3. Планирование с помощью метода CLP	310
14.4. Программа моделирования в ограничениях	317
14.5. Применение метода CLP для поддержки конечных областей определения — CLP(FD)	321
Резюме	324
Источники дополнительной информации	325
Глава 15. Представление знаний и экспертные системы	326
15.1. Назначение и структура экспертной системы	326
15.2. Представление знаний с помощью правил вывода	328
15.3. Прямой и обратный логический вывод в системах, основанных на правилах	331
15.3.1. Обратный логический вывод	331
15.3.2. Прямой логический вывод	333
15.3.3. Сравнение прямого и обратного логического вывода	334
15.4. Формирование объяснений	336
15.5. Учет неопределенности	337
15.5.1. Простая схема учета неопределенности	337
15.5.2. Сложности, связанные с учетом неопределенности	339
15.6. Байесовские сети доверия	340
15.6.1. Вероятности, достоверности и байесовские сети доверия	340
15.6.2. Некоторые формулы из области исчисления вероятностей	344
15.6.3. Формирование рассуждений в байесовских сетях	345
15.7. Семантические сети и фреймы	348
15.7.1. Семантические сети	349
15.7.2. Фреймы	350
Резюме	355
Дополнительные источники информации	356
Глава 16. Командный интерпретатор экспертной системы	357
16.1. Формат представления знаний	357
16.2. Проектирование механизма логического вывода	361
16.2.1. Требования к организации работы программы	361
16.2.2. Организация процесса формирования рассуждений	363
16.2.3. Формирование ответов на вопросы, требующие обоснования необходимости получения запрашиваемой информации	364
16.2.4. Формирование ответов на вопросы, касающиеся описания последовательности рассуждений	365
16.3. Реализация	366
16.3.1. Процедура explore	366
16.3.2. Процедура useranswer	369
16.3.3. Усовершенствование процедуры useranswer	371
16.3.4. Процедура present	376

16.3.5. Управляющая процедура верхнего уровня	377
16.3.6. Пояснения к программе командного интерпретатора	378
16.3.7. Отрицаемые цели	378
16.4. Заключительные замечания	380
Резюме	382
Дополнительные источники информации	382
Глава 17. Планирование	383
17.1. Представление действий	383
17.2. Разработка планов с помощью анализа целей и средств	387
17.3. Защита целей	390
17.4. Процедурные аспекты и режим поиска в ширину	393
17.5. Регрессия целей	395
17.6. Сочетание планирования по принципу анализа целей и средств с эвристическим поиском по заданному критерию	398
17.7. Неконкретизированные действия и планирование с частичным упорядочением	401
17.7.1. Неконкретизированные действия и цели	402
17.7.2. Планирование с частичным упорядочением	404
Резюме	406
Дополнительные источники информации	407
Глава 18. Машинное обучение	408
18.1. Введение	408
18.2. Проблема изучения понятий на примерах	409
18.2.1. Понятия, представленные в виде множеств	409
18.2.2. Примеры и гипотезы	410
18.2.3. Языки описания объектов и понятий	412
18.2.4. Точность гипотез	413
18.3. Подробный пример формирования реляционных описаний в результате обучения	414
18.4. Обучение с помощью простых правил вывода	419
18.4.1. Описание объектов и понятий с использованием атрибутов	419
18.4.2. Логический вывод правил на основании примеров	422
18.5. Логический вывод деревьев решения	426
18.5.1. Основной алгоритм логического вывода дерева	426
18.5.2. Выбор “наилучшего” атрибута	428
18.5.3. Реализация процедуры обучения с помощью дерева решения	430
18.6. Обучение по зашумленным данным и отсеечение частей деревьев	433
18.7. Успех обучения	439
18.7.1. Критерии успеха обучения	440
18.7.2. Оценка точности гипотез, полученных путем обучения	440
18.7.3. Влияние отсеечения частей на точность и наглядность деревьев решения	441
Резюме	442
Дополнительные источники информации	443
Глава 19. Индуктивное логическое программирование	446
19.1. Введение	446
19.2. Формирование программ Prolog на примерах	449
19.2.1. Постановка задачи обучения	449
19.2.2. Граф усовершенствования	450
19.2.3. Программа MINIHYPHER	452
19.2.4. Обобщение, уточнение и тэта-классификация	458
19.3. Программа HYPHER	459
19.3.1. Оператор усовершенствования	460
19.3.2. Поиск	463
19.3.3. Программа HYPHER	463

19.3.4. Эксперименты с программой HYPER	470
Одновременное изучение предикатов odd(L) и even(L)	470
Изучение предиката path(StartNode, GoalNode, Path)	471
Изучение предиката, предназначенного для сортировки по методу вставки	472
Изучение предиката, позволяющего распознать конструкцию арки	474
Резюме	476
Дополнительные источники информации	477
Глава 20. Качественные рассуждения	478
20.1. Здравый смысл, качественные рассуждения и обыденные физические представления	478
20.1.1. Различие между количественными и качественными рассуждениями	478
20.1.2. Качественное абстрагирование количественной информации	479
Абстрагирование числовых данных путем их замены символическими значениями и интервалами	480
Абстрагирование производных по времени путем их замены обозначениями направлений изменения	480
Абстрагирование функций путем замены монотонными отношениями	480
Абстрагирование возрастающих временных последовательностей	480
20.1.3. Назначение и область применения качественного моделирования и качественных рассуждений	481
20.2. Качественные рассуждения о статических системах	482
Вопросы прогностического типа	486
Вопросы диагностического типа	486
Вопросы управленческого типа	486
20.3. Качественные рассуждения о динамических системах	486
20.4. Программа качественного машинного моделирования	493
20.4.1. Способы представления качественных состояний	496
20.4.2. Ограничения	497
20.4.3. Переходы между качественными состояниями	498
20.5. Описание программы качественного машинного моделирования	502
Резюме	507
Дополнительные источники информации	508
Глава 21. Обработка лингвистической информации с помощью грамматических правил	510
21.1. Применение грамматических правил в программе на языке Prolog	510
21.2. Обработка смысла	516
21.2.1. Формирование деревьев синтаксического анализа	516
21.2.2. Применение деревьев синтаксического анализа для извлечения смысла	518
21.2.3. Совместное применение синтаксических и семантических конструкций в системе обозначений DCG	520
21.3. Определение смысла фраз на естественном языке	521
21.3.1. Представление смысла простых фраз с помощью логических высказываний	521
21.3.2. Смысл определяющих слов “a” и “every”	525
21.3.3. Обработка относительных предложений	527
Резюме	531
Дополнительные источники информации	531
Глава 22. Ведение игры	532
22.1. Игры с полной информацией, с двумя участниками	532
22.2. Принцип минимакса	534
22.3. Альфа-бета-алгоритм: эффективная реализация принципа минимакса	537

22.4. Программы, основанные на принципе минимакса: усовершенствования и ограничения	541
22.5. Типовые знания и механизм советов	543
22.5.1. Цели и ограничения ходов	543
22.5.2. Выполнимость совета	544
22.5.3. Объединение элементарных советов в правила и таблицы советов	544
22.6. Программа ведения шахматного эндшпиля на языке Advice Language 0	546
22.6.1. Миниатюрный интерпретатор AL0	546
22.6.2. Программа ведения эндшпиля “король и ладья против короля” на основе таблицы советов	549
Резюме	556
Дополнительные источники информации	557
Глава 23. Метапрограммирование	559
23.1. Метапрограммы и метаинтерпретаторы	559
23.2. Метаинтерпретаторы Prolog	560
23.2.1. Простейший метаинтерпретатор Prolog	560
23.2.2. Трассировка с помощью метаинтерпретатора	562
23.2.3. Формирование деревьев доказательства	563
23.3. Обобщение на основе объяснения	564
Дано	565
Найти	565
23.4. Объектно-ориентированное программирование	570
23.5. Программирование, управляемое шаблонами	576
23.5.1. Архитектура, управляемая шаблонами	576
23.5.2. Программы Prolog, реализованные в виде систем, управляемых шаблонами	578
23.5.3. Пример разработки программ, управляемых шаблонами	579
Модуль 1	579
Модуль 2	579
23.5.4. Простой интерпретатор для программ, управляемых шаблонами	580
23.5.5. Возможные усовершенствования	582
Проект	583
23.6. Простая программа автоматического доказательства теорем, реализованная в виде программы, управляемой шаблонами	583
Резюме	588
Дополнительные источники информации	589
Приложение А. Некоторые различия между реализациями Prolog	590
Динамические и статические предикаты	590
Предикаты assert и retract	590
Неопределенные предикаты	590
Отрицание как недостижение успеха — операторы not и “\+”	591
Предикат name(Atom, CodeList)	591
Загрузка программ с помощью предикатов consult и reconsult	591
Модули	591
Приложение Б. Некоторые часто используемые предикаты	592
Решения к отдельным упражнениям	595
Глава 1	595
Глава 2	595
Глава 3	596
Глава 4	599
Глава 5	600
Глава 6	601
Глава 7	601
Глава 8	602

Глава 9	603
Глава 10	604
Глава 11	605
Глава 12	606
Глава 13	606
Глава 14	606
Глава 15	606
Глава 17	607
Глава 18	607
Глава 19	608
Глава 20	608
Глава 21	610
Глава 23	610
Список литературы	611
Предметный указатель	619

Посвящаю третье издание этой книги моей матери, самому доброму человеку из всех, кого я знаю, и моему отцу, который во время Второй Мировой войны сумел убежать из концентрационного лагеря, прорыв подземный туннель, о чем он написал в своем романе "Телескоп".

Из предисловия Патрика Уинстона ко второму изданию

Никогда не забуду того волнения, которое я испытал, увидев в действии свою первую программу, написанную в стиле Prolog. Эта программа была частью знаменитой системы Shrdlu Терри Винограда (Terry Winograd). Решатель задач, встроенный в систему, работал в “мире блоков” и заставлял руку робота (точнее, ее модель) перемещать кубики на экране дисплея, решая хитроумные задачи, поставленные оператором.

Решатель задач в мире блоков Винограда был написан на языке Microplanner, который, как мы теперь понимаем, был своего рода языком Prolog в миниатюре. Несмотря на все недостатки языка Microplanner, вся работа решателя задач в мире блоков была явно нацелена на достижение определенных целей, поскольку любой язык типа Prolog побуждает программистов мыслить в терминах целей. Благодаря использованию таких целенаправленных процедур, как “схватить”, “освободить верх блока”, “убрать в сторону”, “переместить” и “отпустить”, действия этой ясной, понятной и краткой программы казались удивительно интеллектуальными.

Решатель задач в мире блоков Винограда навсегда изменил мое представление о программировании. Я даже переписал этот решатель задач в мире блоков на языке Lisp и привел в своем учебнике по Lisp, поскольку эта программа неизменно впечатляла меня мощью заложенной в ней философии целевого программирования и той легкостью, с какой создаются целенаправленные программы.

Однако обучение целевому программированию на примерах программ Lisp можно сравнить с чтением произведений Шекспира не на английском языке. Это позволяет получить определенное впечатление, но эстетическое воздействие становится гораздо слабее, чем при восприятии оригинала. Поэтому лучшим способом обучения целевому программированию является чтение и написание программ на языке Prolog, поскольку он как раз и предназначен для программирования в терминах целей.

Вообще говоря, развитие языков программирования происходит на пути перехода от языков низкого уровня, в которых программист определяет, как должны быть выполнены те или иные действия, к языкам высокого уровня, позволяющим просто указать, что должно быть сделано. Например, с появлением языка Fortran программисты избавились от необходимости общаться с компьютером на прокрустовом языке адресов и регистров. Теперь они уже могли выражать свои мысли на естественном (или почти естественном) языке, не считая той небольшой уступки, что для этого должны были использоваться ограниченные, 80-колоновые перфокарты.

Однако Fortran и почти все другие языки программирования все еще остаются языками процедурного типа, которые требуют точно указывать весь процесс решения задачи. На мой взгляд, современный Lisp достиг абсолютного предела развития языков этого типа, поскольку Lisp (вернее, его версия Common Lisp) обладает непревзойденными выразительными возможностями, но качество создаваемых на нем программ полностью зависит от того, насколько сможет ими воспользоваться сам программист. А с появлением языка Prolog, с другой стороны, наметился четко выраженный переход к использованию языков декларативного типа, которые побуждают программиста описывать ситуации и формулировать задачи, а не регламентировать во всех подробностях процедуры решения этих задач.

Отсюда следует, насколько важен вводный курс по языку Prolog для всех студентов, изучающих вычислительную технику и программирование, — ведь просто не существует лучшего способа понять, что представляет собой декларативное программирование.

Многие страницы этой книги могут служить хорошей иллюстрацией того различия, которое существует между процедурным и декларативным стилями программирования. Например, в первой главе это различие иллюстрируется на задачах, относящихся к семейным отношениям. Язык Prolog позволяет явно и естественно определить понятие “дед” (grandfather), указав, что дед — это отец (father) одного из родителей (parent). На этом языке определение предиката grandfather выглядит так:

```
grandfather( X, Z) :- father( X, Y), parent( Y, Z).
```

И сразу после ввода в систему Prolog определения предиката grandfather ей можно задать вопрос, например, каково имя каждого из дедов Патрика. Ниже приведен вопрос, который снова сформулирован в системе обозначений Prolog, наряду с типичным ответом.

```
?- grandfather( X, patrick).  
X = james;  
X = carl
```

Именно система Prolog должна найти способ решения этой задачи, просматривая в базе данных все, что касается отношений father и parent. Программист указывает только, что дано и на какой вопрос должен быть получен ответ. Он в большей степени обязан предоставить системе нужную информацию, чем алгоритмы, с помощью которых ведется обработка этой информации.

Поняв, что очень важно изучить Prolog, нужно решить, как лучше это сделать. Я убежден, что изучение языка программирования во многом сходно с изучением естественного языка. Так, например, в первом случае можно воспользоваться справочным руководством, а во втором — словарем. Но никто не изучает язык по словарю, поскольку слова — это только часть знаний, необходимых для овладения языком. Для этого необходимо также узнать, по каким правилам формируются осмысленные сочетания слов, а затем приобщиться к высшим достижениям искусства речи под руководством тех, кто владеет литературным стилем.

Точно так же, никто не изучает язык программирования только по справочному руководству, поскольку в нем почти ничего не говорится о том, как использовать элементарные конструкции языка те, кто им виртуозно владеет. Поэтому нужен учебник, а признаком хорошего учебника является изобилие примеров, ведь в них сосредоточен богатый опыт, на котором мы в основном и учимся.

В этой книге первый пример появляется уже на первой странице, а далее на читателя как из рога изобилия обрушивается поток примеров программ на языке Prolog, написанных программистом-энтузиастом, горячим приверженцем декларативной идеологии программирования. После тщательного изучения этих примеров читатель не только узнает, как действует система Prolog, но и станет обладателем личной коллекции образцов программ, готовых к употреблению: он может разбирать эти программы на части, приспосабливать каждую часть к своей задаче, а затем снова собирать их вместе, получая при этом новые программы. Такое усвоение предшествующего опыта можно считать первым шагом на пути к овладению мастерством программирования.

Полезным побочным эффектом изучения примеров качественных программ является то, что они позволяют не только освоить само программирование, но и многое узнать об интересной области науки. В данной книге такой научной дисциплиной, лежащей в основе большинства примеров, является искусственный интеллект. Читатель ознакомится с такими идеями в области автоматического решения задач, как сведение задач к подзадачам, прямое и обратное построение цепочки рассуждений, получение объяснения последовательности рассуждений и объяснения предположений, а также освоит различные методы поиска.

Одним из замечательных свойств языка Prolog является то, что он достаточно прост, чтобы студенты могли его использовать непосредственно в процессе изучения вводного курса по искусственному интеллекту. Я не сомневаюсь, что многие преподаватели включают эту книгу в свои курсы искусственного интеллекта с тем, чтобы студенты смогли сами увидеть, как абстрактные идеи приобретают конкретные и действенные формы.

Полагаю, что среди учебников по языку Prolog эта книга окажется особенно популярной, и не только благодаря наличию качественных примеров, но и в связи с другими ее привлекательными особенностями:

- во всех главах книги имеются тщательно составленные резюме;
- все изучаемые понятия подкрепляются многочисленными упражнениями;
- понятие абстракции данных представлено наиболее наглядно — с помощью процедур доступа к элементам структур;
- обсуждению вопросов стиля и методологии программирования посвящена целая глава;
- со всей откровенностью обозначены все трудности, с которыми приходится сталкиваться в ходе программирования на языке Prolog, а не только привлекательные стороны этого процесса.

Все это говорит о том, что перед нами прекрасно написанная, интересная и полезная книга.

Я до сих пор храню первое издание этой книги в своей библиотеке, где она стоит на одной полке с другими выдающимися учебниками по языкам программирования, поскольку обладает такими превосходными особенностями, как ясность и прямота изложения, а также наличие многочисленных примеров, тщательно составленных резюме и многочисленных упражнений. А поскольку эта книга является учебником по языку программирования, мне особенно нравится сделанный в ней акцент на абстракцию данных, внимательное отношение к выбору стиля программирования и непредвзятый анализ не только преимуществ, но и недостатков языка Prolog.

Предисловие

Язык Prolog

Prolog — это язык программирования, сосредоточенный вокруг небольшого набора основных механизмов, включая сопоставление с образцом, древовидное представление структур данных и автоматический перебор с возвратами. Этот ограниченный набор средств образует удивительно мощную и гибкую среду программирования. Prolog особенно хорошо подходит для решения задач, в которых рассматриваются объекты (в частности, структурированные объекты) и отношения между ними. Например, на языке Prolog совсем не сложно выразить пространственные связи между объектами, допустим, указать, что синий шар находится за зеленым. Столь же просто определить можно более общее правило: если объект X ближе к наблюдателю, чем объект Y , а Y ближе, чем Z , то X должен быть ближе, чем Z . После этого система Prolog получает возможность формировать рассуждения о пространственных связях и их совместимости с общим правилом. Благодаря таким особенностям Prolog становится мощным языком для искусственного интеллекта и нечислового программирования в целом. Можно указать известные примеры символических вычислений, реализация которых на других стандартных языках вызывает необходимость создавать десятки страниц кода, чрезвычайно сложного в изучении. А после реализации тех же алгоритмов на языке Prolog результатом становится кристально ясная программа, которая легко помещается на одной странице.

Основные вехи развития языка Prolog

Prolog стал воплощением идеи использования логики в качестве языка программирования, которая зародилась в начале 1970-х годов, и само его название является сокращением от слов “programming in logic” (программирование в терминах логики). Первыми исследователями, которые занялись разработкой этой идеи, были Роберт Ковальски (Robert Kowalski) из Эдинбурга (теоретические основы), Маартен ван Эмден (Maarten van Emden) из Эдинбурга (экспериментальная демонстрационная система) и Ален Колмероз (Alain Colmerauer) из Марселя (реализация). Популяризации языка Prolog во многом способствовала эффективная реализация этого языка в середине 1970-х годов Дэвидом Д. Г. Уорреном (David D.H. Warren) из Эдинбурга. К числу новейших достижений в этой области относятся средства программирования на основе логики ограничений (Constraint Logic Programming — CLP), которые обычно реализуются в составе системы Prolog. Средства CLP показали себя на практике как исключительно гибкий инструмент для решения задач составления расписаний и планирования материально-технического снабжения. А в 1996 году был опубликован официальный стандарт ISO языка Prolog.

Наиболее заметные тенденции в истории развития языка Prolog

В развитии языка Prolog наблюдаются очень интересные тенденции. Этот язык быстро приобрел популярность в Европе как инструмент практического программирования. В Японии вокруг языка Prolog были сосредоточены все разработки компьютеров пятого поколения. С другой стороны, в США этот язык в целом был принят с

небольшим опозданием в связи с некоторыми историческими причинами. Одна из них состояла в том, что Соединенные Штаты вначале познакомились с языком Microplanner, который также был близок к идее логического программирования, но неэффективно реализован. Определенная доля низкой популярности Prolog в этой стране объясняется также реакцией на существовавшую вначале “ортодоксальную школу” логического программирования, представители которой настаивали на использовании чистой логики и требовали, чтобы логический подход не был “запятнан” практическими средствами, не относящимися к логике. В прошлом это привело к широкому распространению неверных взглядов на язык Prolog. Например, некоторые считали, что на этом языке можно программировать только рассуждения с выводом от целей к фактам. Но истина заключается в том, что Prolog — универсальный язык программирования и на нем может быть реализован любой алгоритм. Далекая от реальности позиция “ортодоксальной школы” была преодолена практиками языка Prolog, которые приняли более прагматический подход, воспользовавшись плодотворным объединением нового, декларативного подхода с традиционным, процедурным.

Изучение языка Prolog

Поскольку истоки языка Prolog лежат в области математической логики, преподавание этого языка часто начинают с изучения логики. Но, как оказалось, подобный вводный курс, насыщенный математическими дисциплинами, не очень эффективен, если цель состоит в изучении Prolog как инструмента практического программирования. Поэтому в данной книге меньше всего внимания уделяется математическим аспектам, а изложение основывается на изучении искусства использования нескольких базовых механизмов Prolog для решения интересных задач. Обычные языки программирования — процедурно ориентированны, а Prolog является представителем нового поколения программных средств, основанных на применении описательного, или декларативного, подхода. Для его освоения требуется во многом изменить свое представление о способах решения задач, овладеть намного более продуктивными подходами, поэтому обучение программированию на языке Prolog становится увлекательной интеллектуальной деятельностью. Многие полагают, что каждый, кто проходит обучение в области информатики, должен на определенном этапе получить хотя бы некоторое представление о языке Prolog, поскольку этот язык требует использования иного принципа решения задач и позволяет по-другому взглянуть на традиционные языки программирования.

Содержание книги

В части I приведены вводные сведения о языке Prolog и показан процесс разработки программ Prolog. Кроме того, в нее включено описание методов обработки таких важных структур данных, как деревья и графы, поскольку эти методы находят широкое распространение. В части II 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. Методы искусственного интеллекта описываются и разрабатываются до такой степени детализации, которая позволяет успешно реализовать их на языке Prolog и получить законченные программы. Затем эти программы могут использоваться в качестве структурных блоков для сложных приложений. В заключительной главе, посвященной метапрограммированию, показано, как можно применять Prolog для реализации других языков и принципов программирования, включая объектно-ориентированное программирование, программирова-

ние, управляемое шаблонами, а также написание интерпретаторов Prolog на языке Prolog. Во всей книге акцент делается на создание наиболее удобных для понимания программ; в этих программах исключены все затрудняющие понимание “программистские трюки”, которые основаны на средствах системы, зависящих от конкретной реализации.

Различия между вторым и третьим изданиями

Весь материал книги пересмотрен и обновлен. Введены новые главы, в которых рассматриваются следующие темы:

- программирование на основе логики ограничений (Constraint Logic Programming — CLP);
- индуктивное логическое программирование;
- качественные рассуждения.

В числе других важных изменений можно назвать следующие:

- описание сетей доверия (байесовских сетей) в главе по представлению знаний и экспертным системам;
- описание программ поиска по заданному критерию (IDA*, RBFS), характеризующихся низкими требованиями к памяти, в главе по эвристическому поиску;
- существенные дополнения в главе, посвященной машинному обучению;
- описание дополнительных методов повышения эффективности программ в главе, посвященной стилю и методам программирования.

Во всей книге больше внимания уделено различиям между реализациями Prolog и, в случае необходимости, даны конкретные ссылки на стандарт Prolog (см. также приложение А).

Для кого предназначена эта книга

Данная книга предназначена для тех, кто проходит обучение в области языка Prolog и искусственного интеллекта. Ее можно использовать в составе курса по языку Prolog или курса по искусственному интеллекту, в котором принципы искусственного интеллекта иллюстрируются на основе Prolog. Предполагается, что читатель имеет основной запас общих знаний в области компьютерных наук, но наличие знаний в области искусственного интеллекта не требуется. Не обязательна также значительная подготовка в области программирования; напротив, богатый опыт и привычка к обычному процедурному стилю программирования (например, на языке С или Pascal) могут даже стать препятствием к освоению нового способа мышления, который требуется при изучении языка Prolog.

Стандартный синтаксис, используемый в книге

Среди нескольких диалектов Prolog наиболее широкое распространение получил так называемый *эдинбургский синтаксис*, известный также как *синтаксис DEC-10*; этот же диалект стал основой стандарта ISO для языка Prolog. Он применяется и в данной книге. Для обеспечения совместимости с различными реализациями Prolog в этой книге используется лишь относительно небольшое подмножество встроенных средств, которые являются общими для многих версий Prolog.

Как читать эту книгу

Прежде всего необходимо прочитать подряд весь материал части I. Но раздел 2.4 с описанием процедурного смысла программ Prolog является более сложным, поэтому трудный для восприятия материал этого раздела при первом чтении можно пропустить. В главе 4 представлены примеры программ, которые могут быть прочитаны (или пропущены) избирательно. Глава 10, посвященная сложным способам представления деревьев, также может быть пропущена.

В части II читатель получает больше возможностей выбора изучаемого материала, поскольку главы этой части в основном не зависят друг от друга. Но, вполне естественно, некоторые темы все еще должны быть изучены после других, например, те, что опираются на основные стратегии поиска (глава 11). На рис. 1 показана рекомендуемая последовательность чтения глав.

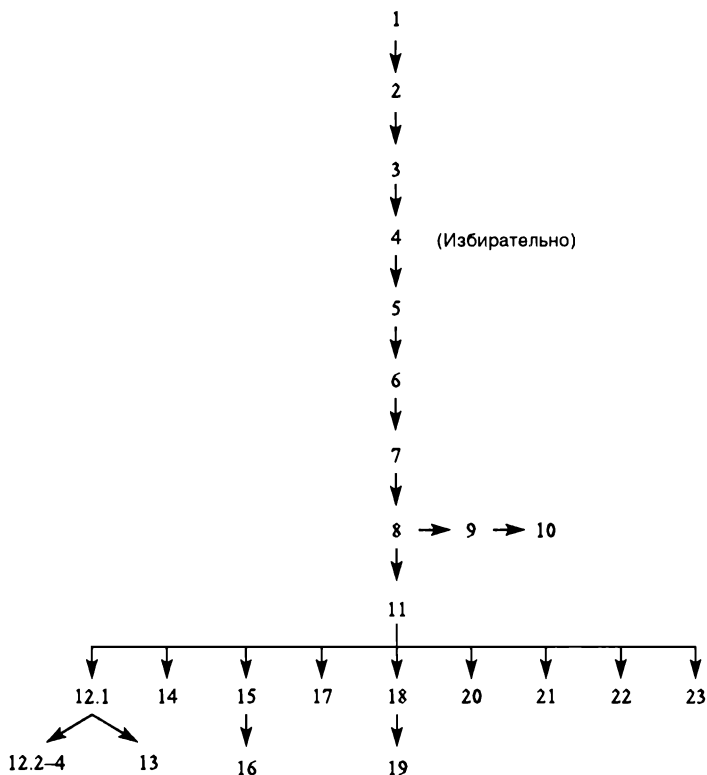


Рис. 1. Рекомендуемая последовательность изучения глав книги

Код программ и материалы курсов

Исходный код всех программ этой книги и соответствующие материалы курсов можно получить на сопровождающем Web-узле (www.booksites.net/bratko).

Благодарности

Пробуждению у меня интереса к языку Prolog я обязан Дональду Мичи (Donald Michie). Я очень благодарен Лоренсу Берду (Lawrence Byrd), Фернандо Перейре (Fernando Pereira) и Дэвиду Г. Д. Уоррену (David H. D. Warren), которые в свое время входили в состав группы разработчиков Prolog в Эдинбурге, за их советы по программированию и интересные дискуссии. Книга стала намного лучше под влиянием комментариев и предложений к предыдущим изданиям Эндрю Макгеттрика (Andrew McGettrick) и Патрика Г. Уинстона (Patrick H. Winston). Из числа тех, кто прочитал отдельные части рукописи и внес существенные комментарии, хочу особо отметить Дамиана Бояджиева (Damjan Bojadžiev), Рода Брайстоу (Rod Bristow), Питера Кларка (Peter Clark), Франса Кенена (Frans Coenen), Дэвида К. Додсона (David C. Dodson), Сашо Джероски (Sašo Džeroski), Богдана Филипча (Bogdan Filipič), Вана Фоккинга (Wan Fokkink), Матяжа Гамса (Matjaž Gams), Питера Г. Гринфилда (Peter G. Greenfield), Марко Гробелника (Marko Grobelnik), Криса Хинде (Chris Hinde), Игоря Кононенко (Igor Kononenko), Матевжа Ковачича (Matevž Kovatič), Эдуардо Моралеса (Eduardo Morales), Игоря Можетича (Igor Možetič), Тимоти Б. Ниблетта (Timothy B. Niblett), Душана Петерца (Dušan Peterc), Уроша Помпе (Uroš Pompe), Роберта Родошека (Robert Rodošek), Агату Сакхе (Agata Saje), Клода Саммю (Claude Sammut), Сима Сэя (Cem Say), Ашвина Сринивасана (Ashwin Srinivasan), Дориана Сью (Dorian Sue), Питера Тансига (Peter Tancig), Таню Урбанчич (Tanja Urbančič), Марка Уоллеса (Mark Wallace), Уильяма Уолли (William Walley), Саймона Уэйлгана (Simon Weilguny), Блажа Зупана (Blaž Zupan) и Дарко Зупанича (Darko Zupanič). Выражаю особую признательность Симу Сэю за то, что он проверил значительную часть программ и преподнес мне ценный подарок, обнаружив скрытые ошибки. Некоторые читатели, особенно Г. Улснам (G. Oulsnam) и Изток Тврды (Iztok Tvrdu), помогли выявить ошибки в предыдущих изданиях. Хочу также поблагодарить Карен Мосман (Karen Mosman), Джули Найт (Julie Knight) и Карен Сазерленд (Karen Sutherland), сотрудников издательства Pearson Education, за их плодотворную работу в процессе подготовки этой книги. Саймон Пламтри (Simon Plumptre) и Дебра Майзон-Этерингтон (Debra Myson-Etherington) внесли большой вклад в предыдущие издания. Основная часть иллюстраций была подготовлена Дарко Симершеком (Darko Simeršek). И, наконец, хочу отметить, что эта книга не могла бы появиться в свет без стимулирующего влияния творческой деятельности всего международного сообщества специалистов по логическому программированию.

Издательство Pearson Education выражает свою признательность корпорации Plenum Publishing Corporation за полученное разрешение воспроизвести материал, аналогичный приведенному в главе 10 книги *Human and Machine Problem Solving* (1989), К. Gilhooly (ed.).

Иван Братко
Январь 2000 года

От редактора перевода

Считаю свои долгом привести несколько рекомендаций, которые, надеюсь, позволят читателю быстрее приступить к работе с этой замечательной книгой.

Выбор дистрибутива Prolog

Принятие стандарта языка Prolog стало стимулом для создания многих независимых реализаций этого языка, в том числе предоставляемых бесплатно для широкого круга пользователей. Кроме того, и для многих коммерческих систем предусмотрены демонстрационные версии, с лицензией на ограниченный, но достаточно большой период времени, а также так называемые *персональные выпуски* (Personal Edition), которые предоставляются бесплатно для личного пользования. Одним из самых интересных продуктов последнего типа является Visual Prolog (<http://www.visual-prolog.com> или <http://www.pdc.dk>). Поэтому читатель может без особых сложностей найти и загрузить наиболее подходящую для себя версию. Постоянно обновляемый обзор новейших реализаций Prolog можно найти по адресу

<http://dmoz.org/Computers/Programming/Languages/Prolog/Implementations/>

На указанной странице отмечен звездочкой бесплатный пакет SWI-Prolog, который действительно представляет наибольший интерес для тех, кто только приступает к изучению этого языка.

Режимы работы интерпретатора Prolog

Все примеры, приведенные в данной книге, могут быть выполнены с помощью интерпретатора Prolog. После вызова на выполнение интерпретатор выводит приглашение “?-”, которое свидетельствует о том, что работа ведется в режиме выполнения запросов. После ввода запроса интерпретатор обращается к своей базе данных, находит в ней факты и правила, необходимые для ответа на запрос, формирует и выводит ответ. Непосредственно после загрузки в базе данных интерпретатора находятся только стандартные предикаты, обеспечивающие работу системы и выполнение вспомогательных функций. Задача ввода всего того, что требуется для решения практической проблемы (аналога программы на других языках программирования), возлагается на пользователя. В ходе этого система как бы “накапливает знания”, поэтому весь этот процесс принято называть *консультированием* (consulting). Для консультирования системы можно либо пользоваться предикатом `assert` (а также его версиями `asserta` и `assertz`), либо ввести программу из файла. Файл с программой может находиться на внешнем носителе информации, и в таком случае для его ввода применяется предикат `consult` с указанием имени файла. Есть также сокращенный способ указания имени файла — в квадратных скобках. Последний способ позволяет также ввести необходимые факты и правила непосредственно в командной строке интерпретатора. После ввода в приглашении имени “псевдофайла” пользовательского терминала `user` система переходит в режим ввода программы с терминала, как показано ниже.

```
?- [user].  
|: man(dima).  
|: man(kolya).  
|: man(petr).  
|: end_of_file.  
% user://3 compiled 28.12 sec, 80 bytes
```

Для завершения работы в этом режиме необходимо ввести атом `end_of_file.`, с точкой в конце (некоторые системы позволяют завершить ввод с помощью комбинации клавиш, обозначающих конец ввода, таких как `<Ctrl+D>` или `<Ctrl+Z>`).

Разработка на языке Prolog с учетом декларативных и процедурных аспектов

Вне всякого сомнения, программа на языке Prolog способна стать “мозгом” чрезвычайно интересных приложений. Но иногда выполнение чисто “телесных” функций (ввод-вывод, преобразование форматов данных, диалог с пользователем и т.д.) целесообразно возложить на другие языки программирования. В настоящее время подобная практика разработки прикладных программных комплексов на основе Prolog находит очень широкое распространение. К счастью, системы Prolog очень хорошо сочетаются с такими системами программирования, как Java, Delphi, C++ и др. Это позволяет оптимальным образом реализовать в разрабатываемом приложении и процедурные, алгоритмические функции, и методы решения декларативных, интеллектуальных задач. Соответствующие инструментальные средства можно легко найти в Web. (Укажите в поисковом запросе “Prolog” и нужный вам язык программирования.)

Часть I

Язык Prolog

В этой части...

Глава 1. Введение в Prolog	26
Глава 2. Синтаксис и значение программ Prolog	45
Глава 3. Списки, операции, арифметические выражения	76
Глава 4. Использование структур: примеры программ	98
Глава 5. Управление перебором с возвратами	121
Глава 6. Ввод и вывод	136
Глава 7. Дополнительные встроенные предикаты	149
Глава 8. Стиль и методы программирования	169
Глава 9. Операции со структурами данных	192
Глава 10. Усовершенствованные методы представления деревьев	215

Введение в Prolog

В этой главе...

1.1. Определение отношений на основе фактов	26
1.2. Определение отношений на основе правил	30
1.3. Рекурсивные правила	35
1.4. Общие принципы поиска ответов на вопросы системой Prolog	39
1.5. Декларативное и процедурное значение программ	42

В этой главе описаны основные механизмы языка Prolog на примере многочисленных программ. Изложение материала является в основном неформальным, но в нем представлены многие важные концепции, такие как предложения, факты, правила и процедуры Prolog. Кроме того, рассматривается встроенный механизм перебора с возвратами системы Prolog и описываются различия между декларативным и процедурным значениями программы.

1.1. Определение отношений на основе фактов

Prolog — это язык программирования для символических, нечисловых вычислений. Он особенно хорошо приспособлен для решения проблем, которые касаются объектов и отношений между объектами. На рис. 1.1 приведен подобный пример: семейные отношения. Тот факт, что Том является одним из родителей Боба, можно записать на языке Prolog следующим образом:

```
parent( tom, bob ).
```

В данном случае в качестве имени отношения выбрано слово `parent`; `tom` и `bob` являются параметрами этого отношения. По причинам, которые станут понятными позже, такие имена, как `tom`, записываются со строчной буквой в начале. Все дерево семейных отношений, показанное на рис. 1.1, определено с помощью следующей программы Prolog:

```
parent( pam, bob ).  
parent( tom, bob ).  
parent( tom, liz ).  
parent( bob, ann ).  
parent( bob, pat ).  
parent( pat, jim ).
```

Эта программа состоит из шести предложений, каждое из которых объявляет один факт об отношении `parent`. Например, факт `parent( tom, bob )` представляет собой конкретный экземпляр отношения `parent`. Такой экземпляр называют также *связью*. В целом *отношение* определяется как множество всех своих экземпляров.

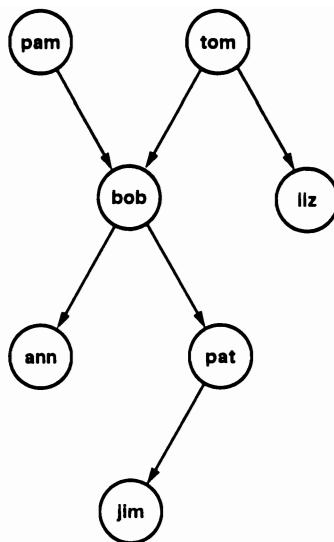


Рис. 1.1. Дерево семейных отношений

После передачи соответствующей программы в систему Prolog последней можно задать некоторые вопросы об отношении `parent`, например, является ли Боб одним из родителей Пэт? Этот вопрос можно передать системе Prolog, введя его на терминале:

```
?- parent( bob, pat).
```

Обнаружив, что это — факт, о существовании которого утверждается в программе, Prolog отвечает:

```
yes
```

После этого можно задать еще один вопрос:

```
?- parent( liz, pat).
```

Система Prolog ответит:

```
no
```

поскольку в программе нет упоминания о том, что Лиз является одним из родителей Пэт. Система ответит также “no” на вопрос

```
?- parent( tom, ben).
```

поскольку в программе даже не встречалось имя Бэн.

Кроме того, системе можно задать более интересные вопросы. Например, кто является родителями Лиз?

```
?- parent( X, liz).
```

На этот раз Prolog ответит не просто “yes” или “no”, а сообщит такое значение `X`, при котором приведенное выше утверждение является истинным. Поэтому ответ будет таковым:

```
X = tom
```

Вопрос о том, кто является детьми Боба, можно сообщить системе Prolog следующим образом:

```
?- parent( bob, X).
```

На этот раз имеется больше одного возможного ответа. Система Prolog вначале выдаст в ответ одно решение:

```
X = ann
```

Теперь можно потребовать у системы сообщить еще одно решение (введя точку с запятой), и Prolog найдет следующий ответ:

```
X = pat
```

Если после этого будут затребованы дополнительные решения, Prolog ответит "no", поскольку все решения уже исчерпаны.

Этой программе может быть задан еще более общий вопрос о том, кто является чьим родителем? Этот вопрос можно также сформулировать иным образом:

Найти X и Y, такие, что X является одним из родителей Y.

Этот вопрос может быть оформлен на языке Prolog следующим образом:

```
?- parent( X, Y).
```

После этого Prolog начнет отыскивать все пары родителей и детей одну за другой. Решения отображаются на дисплее по одному до тех пор, пока Prolog получает указание найти следующее решение (в виде точки с запятой) или пока не будут найдены все решения. Ответы выводятся следующим образом:

```
X = pam  
Y = bob;  
X = tom  
Y = bob;  
X = tom  
Y = liz;  
...
```

Чтобы прекратить вывод решений, достаточно нажать клавишу <Enter> вместо точки с запятой.

Этой программе, рассматриваемой в качестве примера, можно задать еще более сложный вопрос, например, спросить о том, кто является родителями родителей Джима (дедушками и бабушками). Поскольку в программе непосредственно не предусмотрено использование соответствующего отношения `grandparent`, этот запрос необходимо разбить на следующие два этапа (рис. 1.2).

1. Кто является одним из родителей Джима? Предположим, что это — некоторый объект Y.
2. Кто является одним из родителей Y? Предположим, что это — некоторый объект X.

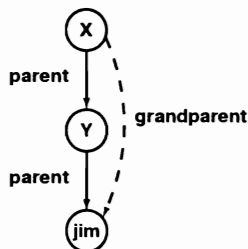


Рис. 1.2. Отношение `grandparent`, выраженное как композиция двух отношений `parent`

Подобный сложный запрос записывается на языке Prolog как последовательность двух простых:

```
?- parent( Y, jim), parent( X, Y).
```

Ответ должен быть следующим:

```
X = bob  
Y = pat
```


Этот составной запрос можно прочесть таким образом: найти такие X и Y , которые удовлетворяют следующим двум требованиям:

`parent( Y, jim)` и `parent( X, Y)`

Если же будет изменен порядок следования этих двух требований, то логический смысл всего выражения останется тем же:

`parent( X, Y)` и `parent( Y, jim)`

Безусловно, такое же действие может быть выполнено и в программе Prolog, поэтому запрос:

`?- parent( X, Y), parent( Y, jim).`

выдаст тот же результат.

Аналогичным образом, программе можно задать вопрос о том, кто является внуками Тома:

`?- parent( tom, X), parent( X, Y).`

Система Prolog ответит следующим образом:

```
X = bob
Y = ann;
X = bob
Y = pat
```

Могут быть также заданы и другие вопросы, например, имеют ли Энн и Пэт общих родителей. Эту задачу также можно разбить на два этапа.

1. Кто является одним из родителей Энн (X)?
2. Является ли (тот же) X одним из родителей Пэт?

Поэтому соответствующий вопрос в языке Prolog выглядит следующим образом:

`?- parent( X, ann), parent( X, pat).`

Ответом на него является:

```
X = bob
```

Рассматриваемый пример программы позволяет проиллюстрировать перечисленные ниже важные понятия.

- В языке Prolog можно легко определить отношение, такое как `parent`, задавая n -элементные кортежи объектов, которые удовлетворяют этому отношению.
- Пользователь может легко запрашивать систему Prolog об отношениях, определенных в программе.
- Программа Prolog состоит из предложений. Каждое предложение оканчивается точкой.
- Параметрами отношений могут быть (кроме всего прочего) определенные объекты, или константы (такие как `tom` и `ann`), а также объекты более общего характера (такие как X и Y). Объекты первого типа, применяемые в рассматриваемой программе, называются *атомами*. Объекты второго типа называются *переменными*.
- Вопросы к системе состоят из одной или нескольких *целей*. Последовательность целей, такая как
`parent( X, ann), parent( X, pat)`

означает конъюнкцию целей:

X является одним из родителей Энн и
 X является одним из родителей Пэт.

Слово “цель” (`goal`) используется для обозначения таких вопросов потому, что система Prolog воспринимает вопросы как цели, которых необходимо достичь.

- Ответ на вопрос может быть положительным или отрицательным, в зависимости от того, может ли быть достигнута соответствующая цель или нет. В слу-

чае положительного ответа считается, что соответствующая цель была достижимой и что цель достигнута. В противном случае цель была недостижимой и не достигнута.

- Если вопросу соответствует несколько ответов, Prolog отыскивает столько ответов, сколько потребует пользователь (в пределах возможного).

Упражнения

1.1. При условии, что определено отношение `parent`, как описано в этом разделе (см. рис. 1.1), укажите, какой ответ даст система Prolog на приведенные ниже вопросы?

- а) `?- parent( jim, X).`
- б) `?- parent( X, jim).`
- в) `?- parent( pam, X), parent( X, pat).`
- г) `?- parent( pam, X), parent( X, Y), parent( Y, jim).`

1.2. Сформулируйте на языке Prolog перечисленные ниже вопросы об отношении `parent`.

- а) Кто является родителем Пэт?
- б) Имеет ли Лиз ребенка?
- в) Кто является дедушкой или бабушкой Пэт?

1.2. Определение отношений на основе правил

Рассматриваемый пример программ можно легко дополнить с применением многих интересных способов. Вначале введем информацию о мужском или женском поле людей, участвующих в отношении `parent`. Эту задачу можно решить, добавив следующие факты к программе:

```
female( pam ).
male( tom ).
male( bob ).
female( liz ).
female( pat ).
female( ann ).
male( jim ).
```

В этом случае введены отношения `male` и `female`. Эти отношения являются унарными (или одноместными). Бинарные отношения типа `parent` определяют связь между парами объектов. С другой стороны, унарные отношения могут использоваться для объявления простых свойств объектов, которые они могут иметь или не иметь. Первое из приведенных выше унарных предложений можно прочесть таким образом: Пэм — женщина. Вместо этого информацию, объявленную в двух унарных отношениях, можно передать с помощью одного бинарного отношения. В таком случае приведенная выше часть программы примет примерно такой вид:

```
sex( pam, feminine ).
sex( tom, masculine ).
sex( bob, masculine ).
```

В качестве следующего дополнения к программе введем отношение `offspring` (отпрыск), обратное отношению `parent`. Отношение `offspring` можно определить таким же образом, как и `parent`, предоставив список обычных фактов об отношении `offspring`, и в качестве каждого факта указать такую пару людей, что один из них является сыном или дочерью другого, например:

```
offspring( liz, tom ).
```

Но отношение `offspring` можно определить гораздо более изящно, используя то, что оно является противоположным `parent` и что `parent` уже было определено. Этот альтернативный способ может быть основан на следующем логическом утверждении:

Для всех X и Y ,

Y является сыном или дочерью X , если
 X является родителем Y .

Такая формулировка уже более близка к синтаксису языка Prolog. Соответствующее предложение Prolog, которое имеет тот же смысл, выглядит таким образом:
`offspring( Y, X) :- parent( X, Y).`

Это предложение можно также прочитать так:

Для всех X и Y ,

если X является родителем Y , то
 Y является сыном или дочерью X .

Предложения Prolog, такие как

`offspring( Y, X) :- parent( X, Y).`

называются *правилами*. Между фактами и правилами существует важное различие. Такие факты, как

`parent( tom, liz).`

представляют собой логические утверждения, которые всегда и безусловно являются истинными. С другой стороны, правила представляют собой утверждения, которые становятся истинными, если удовлетворяются некоторые условия. Поэтому принято считать, что правила состоят из следующих частей:

- условие (правая часть правила);
- заключение (левая часть правила).

Часть, соответствующая заключению, называется также *головой предложения*, а часть, соответствующая условию, — *телом предложения*, как показано ниже.

`offspring( Y, X) :- parent( X, Y).`

голова

тело

Если условие `parent( X, Y)` является истинным, то его логическим следствием становится `offspring( Y, X)`.

Применение правил в языке Prolog иллюстрируется в следующем примере. Зададим программе вопрос, является ли Лиз дочерью Тома:

`?- offspring( liz, tom).`

Поскольку в программе отсутствуют факты о дочерях и сыновьях, единственным способом поиска ответа на этот вопрос является использование правила с определением отношения `offspring`. Данное правило является общим в том смысле, что оно применимо к любым объектам X и Y ; но его можно также применить и к таким конкретным объектам, как `liz` и `tom`. Чтобы применить правило к объектам `liz` и `tom`, вместо Y необходимо подставить `liz`, а вместо X — `tom`. Это действие называется *конкретизацией переменных* (в данном случае — X и Y), которое выполняется следующим образом:

$X = tom$ и $Y = liz$

После конкретизации будет получен частный случай общего правила, который выглядит следующим образом:

`offspring( liz, tom) :- parent( tom, liz).`

Часть с обозначением условия принимает вид

`parent( tom, liz)`

После этого система Prolog пытается определить, является ли истинной часть с обозначением условия. Поэтому первоначальная цель:

`offspring( liz, tom)`

заменяется подцелью:

```
parent( tom, liz)
```

Оказалось, что задача достижения этой (новой) цели является тривиальной, поскольку ее можно найти как факт в рассматриваемой программе. Это означает, что часть данного правила с обозначением заключения также является истинной, и Prolog в качестве ответа на вопрос выводит `yes`.

Теперь введем в рассматриваемый пример программы еще некоторую информацию о семейных отношениях. Определение отношения `mother` может быть основано на следующем логическом утверждении:

Для всех X и Y ,
 X является матерью Y , если
 X является одним из родителей Y и
 X — женщина.

Это утверждение можно перевести на язык Prolog в виде следующего правила:

```
mother( X, Y) :- parent( X, Y), female(X).
```

Запятая между двумя условиями указывает на конъюнкцию этих условий; это означает, что оба условия должны быть истинными.

Такие отношения, как `parent`, `offspring` и `mother`, можно проиллюстрировать с помощью схем, подобных приведенным на рис. 1.3. Эти схемы соответствуют следующим соглашениям. Узлы графов относятся к объектам, т.е. параметрам отношений. Дуги между узлами соответствуют бинарным (или двухместным) отношениям. Дуги направлены от первого параметра отношения ко второму. Унарные отношения обозначаются на схемах путем проставления отметки на соответствующих объектах с именем отношения. Отношения, которые определены на основе других отношений, представлены в виде пунктирных дуг. Поэтому каждую схему необходимо интерпретировать следующим образом: если соблюдаются отношения, обозначенные сплошными дугами, то соблюдаются и созданные на их основе отношения, обозначенные пунктирными дугами. Согласно рис. 1.3, отношение `grandparent` можно непосредственно записать на языке Prolog следующим образом:

```
grandparent( X, Z) :- parent( X, Y), parent( Y, Z).
```

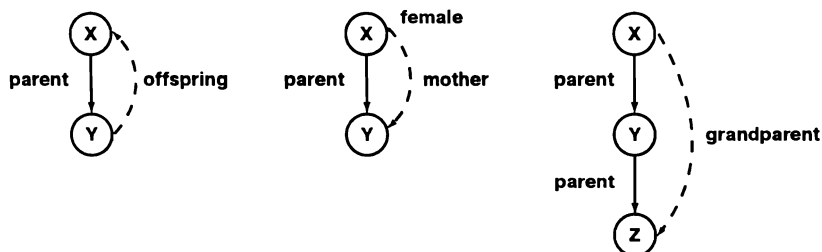


Рис. 1.3. Графы, которые определяют отношения `offspring`, `mother` и `grandparent` в терминах других отношений

На данном этапе необходимо кратко рассмотреть вопрос о компоновке программ. Система Prolog предоставляет почти полную свободу выбора компоновки программ. Поэтому программист может вставлять в текст программы пробелы и пустые строки в полном соответствии со своими вкусами. Но, как правило, следует стремиться к тому, чтобы программы выглядели четкими и аккуратными и, самое главное, были удобными для чтения. Для этого чаще всего голова предложения и каждая цель в его теле записываются на отдельной строке. При этом желательно обозначать цели отступом, чтобы различия между головой и целями стали более очевидными. Например, в соответствии с этими соглашениями правило `grandparent` должно быть записано следующим образом:

```
grandparent( X, Z ) :-
    parent( X, Y),
    parent( Y, Z).
```

Схема отношения *sister* (рис. 1.4) имеет следующее определение:

Для любого X и Y

X является сестрой Y, если

- 1) X и Y имеют общего родителя и
- 2) X - женщина.

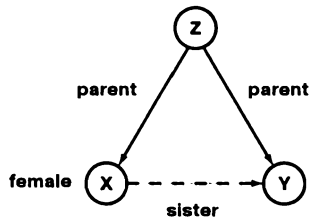


Рис. 1.4. Определение отношения *sister*

Граф, представленный на рис. 1.4, можно перевести на язык Prolog следующим образом:

```
sister( X, Y ) :- parent( Z, X ), parent( Z, Y ), female(X).
```

Обратите внимание на то, каким способом было выражено требование “X и Y имеют общего родителя”. Для этого использовалась следующая логическая формулировка: некоторый Z должен быть родителем X, и тот же Z должен быть родителем Y. Альтернативный, но менее изящный способ мог предусматривать использование следующей цепочки утверждений: Z1 является родителем X, Z2 является родителем Y и Z1 равно Z2.

Теперь системе можно задать вопрос:

```
?- sister( ann, pat ).
```

Ответом должно быть “yes”, как и следовало ожидать (см. рис. 1.1). Поэтому можно сделать вывод, что отношение *sister* в том виде, в каком оно определено, действует правильно. Но в рассматриваемой программе имеется незаметный на первый взгляд недостаток, который обнаруживается при получении ответа на вопрос о том, кто является сестрой Пэт:

```
?- sister( X, pat ).
```

Prolog находит два ответа, и один из них может оказаться неожиданным.

```
X = ann;
X = pat
```

Итак, Пэт является сестрой самой себя?! По-видимому, такой исход не подразумевался при определении отношения *sister*. Но согласно правилу, касающемуся сестер, ответ системы Prolog является полностью обоснованным. В правиле о сестрах нет упоминания о том, что X и Y не должны быть одинаковыми, если X рассматривается как сестра Y. Поскольку это требование не предъявляется, система Prolog (вполне обоснованно) предполагает, что X и Y могут быть одинаковыми, и поэтому приходит к заключению, что любая женщина, имеющая родителя, является сестрой самой себя.

Чтобы исправить приведенное выше правило о сестрах, необходимо дополнительно указать, что X и Y должны быть разными. Как показано в следующих главах, такую задачу можно решить несколькими способами, но на данный момент предположим, что системе Prolog уже известно отношение *different* и условие

```
different( X, Y )
```

удовлетворяется, если и только если X и Y не равны. Поэтому усовершенствованное правило для отношения `sister` может выглядеть следующим образом:

```
sister( X, Y) :-  
    parent( Z, X),  
    parent( Z, Y),  
    female( X),  
    different( X, Y).
```

На основании изложенного в этом разделе можно сделать следующие важные выводы.

- Программы Prolog можно дополнять, вводя новые предложения.
- Предложения Prolog относятся к трем типам: факты, правила и вопросы.
- С помощью фактов можно вводить в программу сведения, которые всегда и безусловно являются истинными.
- С помощью правил можно вводить в программу сведения, которые являются истинными в зависимости от заданного условия.
- Задавая программе вопросы, пользователь может узнавать, какие сведения являются истинными.
- Предложения языка Prolog состоят из головы и тела. Тело представляет собой список целей, разделенных запятыми. Запятые рассматриваются как знаки конъюнкции.
- Факты представляют собой предложения, которые имеют голову и пустое тело. Вопросы имеют только тело. Правила имеют голову и (непустое) тело.
- В процессе вычисления переменные можно заменять другими объектами. В таком случае переменная становится конкретизированной.
- Предполагается, что на переменные распространяется действие квантора всеобщности, который имеет словесное выражение “для всех”. Но если переменные появляются только в теле, их можно трактовать несколькими способами. Например, предложение
`hasachild( X) :- parent( X, Y).`

можно прочитать двумя приведенными ниже способами.

- а) Для всех X и Y,
если X является родителем Y, то
X имеет ребенка.
- б) Для всех X,
X имеет ребенка, если
существует некоторый Y, такой, что X является родителем Y.

Упражнения

1.3. Преобразуйте приведенные ниже утверждения в правила Prolog.

- а) Каждый, кто имеет ребенка, счастлив (введите отношение `happy` с одним параметром).
- б) Для всех X, если X имеет ребенка, имеющего сестру, то X имеет двоих детей (введите новое отношение `hastwochildren`).

1.4. Определите отношение `grandchild` с использованием отношения `parent`. Подсказка: оно должно быть аналогично отношению `grandparent` (см. рис. 1.3).

1.5. Определите отношение `aunt( X, Y)` в терминах отношений `parent` и `sister`. Для упрощения этой задачи вы можете сначала нарисовать схему для отношения, определяющего понятие `aunt` (тетя), в стиле рис. 1.3.

1.3. Рекурсивные правила

Введем еще одно отношение в рассматриваемую программу с описанием семьи — отношение *predecessor* (предок). Это отношение будет определено в терминах отношения *parent*. Его можно представить с помощью двух правил. Первое правило определяет прямых (непосредственных) предков, а второе правило — не прямых предков. Говорят, что некоторый *X* является непрямым предком некоторого *Z*, если существует цепочка родительских связей между людьми от *X* до *Z*, как показано на рис. 1.5. В примере, приведенном на рис. 1.1, Том является прямым предком Лиз и непрямым предком Пэт.

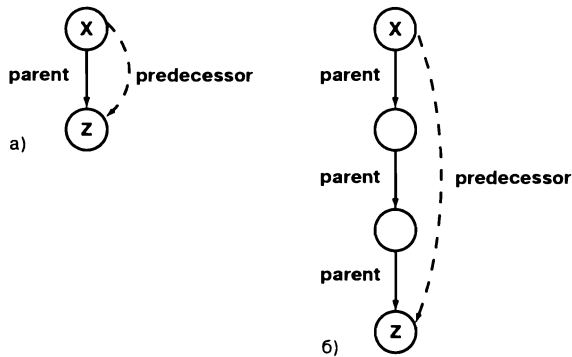


Рис. 1.5. Примеры отношения *predecessor*: а) *X* является прямым предком *Z*; б) *X* является непрямым предком *Z*

Первое правило является простым и может быть сформулировано следующим образом:

Для всех *X* и *Z*,
X — предок *Z*, если
X — родитель *Z*.

Это утверждение можно сразу же перевести на язык Prolog таким образом:

```
predecessor( X, Z ) :-  
    parent( X, Z ).
```

Второе правило, с другой стороны, является более сложным, поскольку решение задачи представления цепочки родительских связей может вызвать некоторые проблемы. Одна из попыток определить не прямых предков показана на рис. 1.6. Согласно этому рисунку, отношение *predecessor* должно быть определено как множество следующих предложений:

```
predecessor( X, Z ) :-  
    parent( X, Z ).
```

```
predecessor( X, Z ) :-  
    parent( X, Y ),  
    parent( Y, Z ).
```

```
predecessor( X, Z ) :-  
    parent( X, Y1 ),  
    parent( Y1, Y2 ),  
    parent( Y2, Z ).
```

```
predecessor( X, Z ) :-  
    parent( X, Y1 ),
```

```
parent( Y1, Y2),
parent( Y2, Y3),
parent( Y3, Z).
...
```

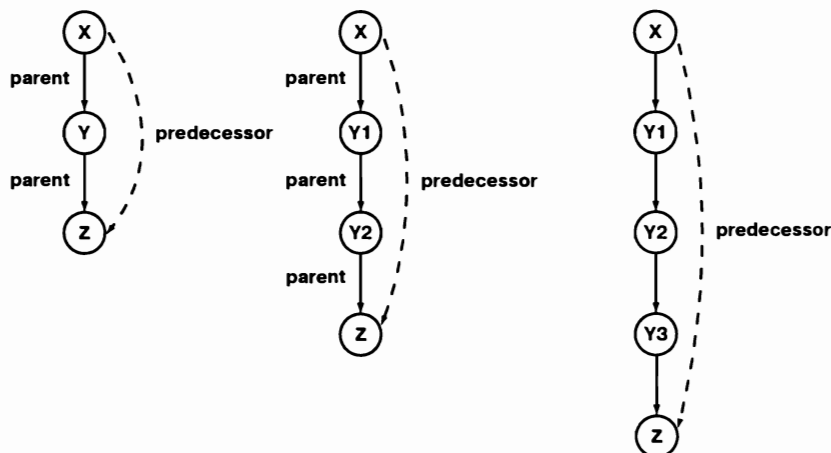


Рис. 1.6. Пары предков и потомков, находящиеся друг от друга на разных расстояниях

Эта программа является слишком длинной, но еще более важно то, что пределы ее действия довольно ограничены. Она позволяет находить предков в генеалогическом дереве только до определенного уровня, поскольку длина цепочки людей между предками и потомками лимитируется длиной существующих предложений с определением предков.

Но для формулировки отношения `predecessor` можно применить гораздо более изящную и правильную конструкцию. Она является правильной в том смысле, что позволяет находить предков до любого колена. Основная идея состоит в том, что отношение `predecessor` должно быть определено в терминах себя самого. Эта идея иллюстрируется на рис. 1.7 и выражается в виде следующего логического утверждения:

Для всех X и Z ,
 X — предок Z ,
 если имеется такой Y , что
 1) X — родитель Y и
 2) Y — предок Z .

Предложение Prolog, имеющее такой же смысл, приведено ниже.

```
predecessor( X, Z) :-
    parent( X, Y),
    predecessor( Y, Z).
```

Таким образом, сформулирована полная программа для отношения `predecessor`, которая состоит из двух правил: первое из них определяет прямых, а вторая — не-прямых предков. Оба правила, записанные вместе, приведены ниже.

```
predecessor( X, Z) :-
    parent( X, Z).

predecessor( X, Z) :-
    parent( X, Y),
    predecessor( Y, Z).
```

Ключом к анализу этой формулировки является то, что отношение `predecessor` используется для определения самого себя. Такая конструкция может показаться на

первый взгляд непонятной, поскольку возникает вопрос, можно ли определить некоторое понятие, используя для этого то утверждение, которое само еще не было полностью определено. Подобные определения имеют общее название *рекурсивных*. С точки зрения логики они являются полностью правильными и понятными; кроме того, они становятся очевидными после изучения схемы, приведенной на рис. 1.7. Но остается нерешенным вопрос, способна ли система Prolog использовать рекурсивные правила. Как оказалась, эта система действительно способна очень легко применять рекурсивные определения. Рекурсивное программирование фактически является одним из фундаментальных принципов программирования на языке Prolog. Без использования рекурсии на языке Prolog невозможно решать какие-либо сложные задачи.

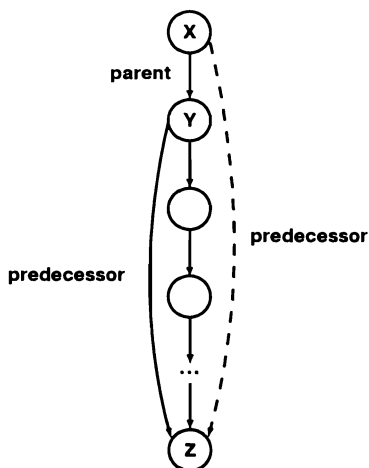


Рис. 1.7. Рекурсивная формулировка отношения predecessor

Возвращаясь к рассматриваемой программе, можно задать системе Prolog вопрос о том, кто является потомками Пэм. Иными словами, для кого Пэм является предком?

```
?- predecessor( pam, X).
X = bob;
X = ann;
X = pat;
X = jim
```

Ответы системы Prolog, безусловно, верны и логически следуют из определения отношений predecessor и parent. Тем не менее остается открытым весьма важный вопрос: как фактически система Prolog использует программу для поиска этих ответов?

Неформальное описание того, как эта задача решается в системе Prolog, приведено в следующем разделе. Но вначале соединим все фрагменты программы с описанием семьи, которая постепенно совершенствовалась путем добавления новых фактов и правил. Окончательная форма этой программы приведена в листинге 1.1. Прежде чем перейти к описанию этого листинга, необходимо сделать следующие замечания: во-первых, дать определение термина *процедура*, а во-вторых, отметить, как используются комментарии в программах.

Листинг 1.1. Программа family с описанием семьи

```
parent( pam, bob).           % Пэм является одним из родителей Боба
parent( tom, bob).
parent( tom, liz).
parent( bob, ann).
parent( bob, pat).
```

```

parent( pat, jim).

female( pam).           % Пэм - женщина
male( tom).             % Том - мужчина
male( bob).
female( liz).
female( ann).
female( pat).
male( jim).

offspring( Y, X) :-      % Y является сыном или дочерью X, если
    parent( X, Y).      % X является одним из родителей Y

mother( X, Y) :-         % X является матерью Y, если
    parent( X, Y),      % X является одним из родителей Y и
    female( X).         % X - женщина

grandparent( X, Z) :-   % X является дедушкой или бабушкой Z, если
    parent( X, Y),      % X является одним из родителей Y и
    parent( Y, Z).      % Y является одним из родителей Z

sister( X, Y) :-        % X является сестрой Y, если
    parent( Z, X),
    parent( Z, Y),      % X и Y имеют одного и того же родителя и
    female( X),         % X - женщина и
    different( X, Y).    % X и Y являются разными

predecessor( X, Z) :-   % Правило pr1: X - предок Z
    parent( X, Z).

predecessor( X, Z) :-   % Правило pr2: X - предок Z
    parent( X, Y),
    predecessor( Y, Z)
.....

```

В программе, приведенной в листинге 1.1, определено несколько отношений: `parent`, `male`, `female`, `predecessor` и т.д. Например, отношение `predecessor` определено с помощью двух предложений. Принято считать, что оба эти предложения касаются отношения `predecessor`. Иногда удобно рассматривать как единое целое все множество предложений, касающихся одного и того же отношения. Подобный набор предложений называется *процедурой*.

В листинге 1.1 два правила, касающиеся отношения `predecessor`, обозначены разными именами, `pr1` и `pr2`, которые указаны в виде комментариев к программе. Эти имена будут использоваться в дальнейшем в качестве ссылок на данные правила. Обычно комментарии игнорируются системой Prolog. Они служат лишь в качестве дополнительно пояснения для лица, читающего программу. Комментарии в языке Prolog отделяются от остальной части программы специальными символьными скобками `"/"` и `"*/"`. Таким образом, комментарии в языке Prolog выглядят следующим образом:

```
/* Это - комментарий */
```

Еще один метод, более удобный для оформления коротких комментариев, предусматривает использование знака процента `"%"`. Все, что находится между знаком `"%"` и концом строки, интерпретируется как комментарий.

```
% Это - также комментарий
```

Упражнение

1.6. Рассмотрим следующее альтернативное определение отношения `predecessor`:

```
predecessor( X, Z) :-
    parent( X, Z).
```

```
predecessor( X, Z) :-
    parent( Y, Z),
    predecessor( X, Y).
```

Можно ли это определение отношения `predecessor` также считать правильным? Откорректируйте схему, приведенную на рис. 1.7, чтобы она соответствовала этому новому определению.

1.4. Общие принципы поиска ответов на вопросы системой Prolog

В этом разделе дано неформальное описание процесса поиска ответов на вопросы в системе Prolog. Вопрос к системе Prolog всегда представляет собой последовательность из одной или нескольких целей. Чтобы ответить на вопрос, Prolog пытается достичь всех целей. Но что в данном контексте означает выражение “достичь цели”? Достичь цели — это значит продемонстрировать, что цель является истинной, при условии, что отношения в программе являются истинными. Другими словами, выражение *достичь цели* означает: продемонстрировать, что цель логически следует из фактов и правил, заданных в программе. Если вопрос содержит переменные, система Prolog должна также найти конкретные объекты (вместо переменных), при использовании которых цели достигаются. Для пользователя отображаются варианты конкретизации переменных, полученные при подстановке конкретных объектов вместо переменных. Если система Prolog не может продемонстрировать для какого-то варианта конкретизации переменных, что цели логически следуют из программы, то выдает в качестве ответа на вопрос слово “no”.

Таким образом, с точки зрения математики программу Prolog следует интерпретировать так: Prolog принимает факты и правила как набор аксиом, а вопрос пользователя — как теорему, требующую доказательства; затем Prolog пытается доказать теорему, т.е. продемонстрировать, что она является логическим следствием из аксиом.

Проиллюстрируем этот подход к описанию работы системы Prolog на классическом примере. Предположим, что заданы приведенные ниже аксиомы.

Все люди способны ошибаться.
Сократ — человек.

Из этих двух аксиом логически следует теорема:

Сократ способен ошибаться.

Первая аксиома, приведенная выше, может быть переформулирована следующим образом:

Для всех X, если X — человек, то X способен ошибаться.

Соответствующим образом этот пример может быть переведен на язык Prolog, как показано ниже.

```
fallible( X) :- man( X).    % Все люди способны ошибаться
man( socrates).           % Сократ — человек
?- fallible( socrates).    % Сократ способен ошибаться?
yes                         % Да
```

Более сложный пример, взятый из программы с описанием семьи (см. листинг 1.1), представлен ниже.

```
?- predecessor( tom, pat).
```

Известно, что в этой программе определен факт `parent( bob, pat)`. Используя этот факт и правило `pr1`, можно прийти к заключению, что имеет место факт `predecessor( bob, pat)`. Это — производный факт, в том смысле, что его нельзя непосредственно найти в программе, но можно вывести из фактов и правил программы. Этап вывода, подобный этому, можно записать в более компактной форме следующим образом:

```
parent( bob, pat) ==> predecessor( bob, pat)
```

Это выражение можно прочесть так: из факта `parent( bob, pat)` следует факт `predecessor( bob, pat)`, согласно правилу `pr1`. Кроме того, известно, что определен факт `parent( tom, bob)`. Используя этот факт и производный факт `predecessor( bob, pat)`, можно сделать вывод, что имеет место факт `predecessor( tom, pat)`, согласно правилу `pr2`. Тем самым показано, что целевое выражение `predecessor( tom, pat)` является истинным. Весь этот двухэтапный процесс вывода можно записать следующим образом:

```
parent( bob, pat) ==> predecessor( bob, pat)
parent( tom, bob) и predecessor( bob, pat) ==> predecessor( tom, pat)
```

Итак, было показано, что может существовать последовательность шагов, позволяющих достичь определенной цели, т.е. выяснить, что цель является истинной. Такая последовательность шагов называется *последовательностью доказательства*. Тем не менее еще не показано, как фактически система Prolog находит такую последовательность доказательства.

Prolog ищет последовательность доказательства в порядке, обратном тому, который был только что использован. Эта система начинает не с простых фактов, заданных в программе, а с целей, и с помощью правил заменяет текущие цели новыми до тех пор, пока не обнаружится, что новые цели являются простыми фактами. Рассмотрим вопрос:

```
?- predecessor( tom, pat).
```

Система Prolog пытается достичь данной цели. Для этого она предпринимает попытки найти в программе предложение, из которого непосредственно может следовать указанная выше цель. Очевидно, что в этом случае подходящими являются только предложения `pr1` и `pr2`. Это — правила, касающиеся отношения `predecessor`. Говорят, что головы этих правил соответствуют цели.

Два предложения, `pr1` и `pr2`, представляют собой два альтернативных способа дальнейших действий системы Prolog. Она вначале проверяет предложение, которая находится на первом месте в программе:

```
predecessor( X, Z) :- parent( X, Z).
```

Поскольку целью является факт `predecessor( tom, pat)`, необходимо выполнить конкретизацию переменных в этом правиле следующим образом:

```
X = tom, Z = pat
```

Затем первоначальная цель `predecessor( tom, pat)` заменяется следующей новой целью:

```
parent( tom, pat)
```

Данный этап использования правила преобразования цели в другую цель, как указано выше, схематически показан на рис. 1.8. В программе отсутствует предложение, которое соответствует цели `parent( tom, pat)`, поэтому такая цель непосредственно не достижима. Теперь система Prolog возвращается к первоначальной цели, чтобы проверить альтернативный способ вывода главной цели `predecessor( tom, pat)`. Поэтому правило `pr2` проверяется следующим образом:

```
predecessor( X, Z) :-
  parent( X, Y),
  predecessor( Y, Z).
```

Как было указано выше, конкретизация переменных `X` и `Z` выполняется следующим образом:

```
X = tom, Z = pat
```

Но конкретизация `Y` еще не выполнена. Верхняя цель `predecessor( tom, pat)` заменяется двумя следующими целями:

```
parent( tom, Y),
predecessor( Y, pat)
```

Этот этап выполнения показан на рис. 1.9, который демонстрирует дальнейшее развитие ситуации, представленной на рис. 1.8.

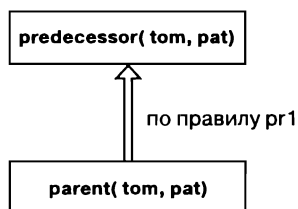


Рис. 1.8. Первый этап выполнения программы; цель, показанная сверху, является истинной, если истинна цель, показанная внизу

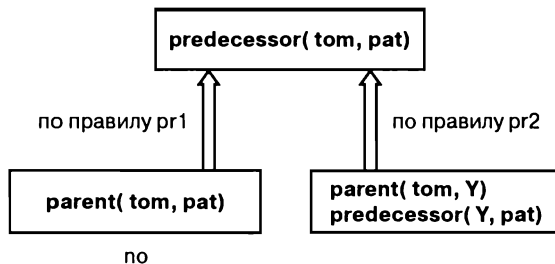


Рис. 1.9. Продолжение схемы выполнения программы, показанной на рис. 1.8

Теперь система Prolog сталкивается с необходимостью заниматься двумя целями и пытается их достичь в том порядке, в каком они записаны. Первая цель достигается легко, поскольку она соответствует одному из фактов в программе. Это соответствие вынуждает выполнить конкретизацию переменной Y и подстановку вместо нее значения bob . Таким образом, достигается первая цель, и оставшаяся цель принимает вид

```
predecessor( bob, pat)
```

Для достижения данной цели снова используется правило $pr1$. Следует отметить, что это (второе) применение того же правила не имеет ничего общего с его предыдущим применением. Поэтому система Prolog использует в правиле новый набор переменных при каждом его применении. Чтобы продемонстрировать это, переименуем переменные в правиле $pr1$ для этого этапа применения правила следующим образом:

```
predecessor( X', Z') :-  
    parent( X', Z').
```

Голова данного правила должна соответствовать текущей цели $predecessor(bob, pat)$, поэтому:

```
X' = bob,  
Z' = pat
```

Текущая цель заменяется следующей:

```
parent( bob, pat)
```

Данная цель достигается сразу же, поскольку она представлена в программе в виде факта. На этом завершается формирование схемы выполнения, которая представлена в графическом виде на рис. 1.10.

Графическая схема выполнения программы (см. рис. 1.10) имеет форму дерева. Узлы дерева соответствуют целям или спискам целей, которые должны быть достигнуты. Дуги между узлами соответствуют этапам применения (альтернативных) предложений программы, на которых цели одного узла преобразуются в цели другого узла. Верхняя цель достигается после того, как будет найден путь от корневого узла (верхней цели) к лист-узлу, обозначенному как "yes". Лист носит метку "yes", если он представляет собой простой факт. Процесс выполнения программ Prolog состоит в поиске путей, оканчивающихся такими простыми фактами. В ходе поиска система Prolog может войти в одну из ветвей, не позволяющих достичь успеха. При обнаружении того, что ветвь не позволяет достичь цели, система Prolog автоматически возвращается к предыдущему узлу и пытается использовать в этом узле альтернативное предложение.

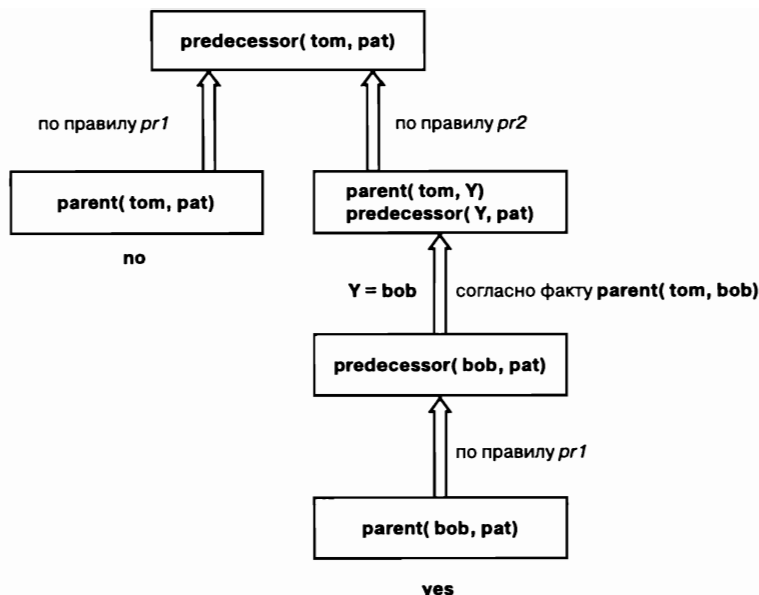


Рис. 1.10. Полная схема процесса достижения цели `predecessor( tom, pat)`. Правая ветвь показывает, что цель является достижимой

Упражнение

1.7. Укажите, каким образом система Prolog вырабатывает ответы на следующие вопросы с использованием программы, приведенной в листинге 1.1. Составьте соответствующие схемы выполнения в стиле рис. 1.8–1.10. Происходят ли возвраты при поиске ответов на некоторые вопросы?

- `?- parent( pam, bob).`
- `?- mother( pam, bob).`
- `?- grandparent( pam, ann).`
- `?- grandparent( bob, jim).`

1.5. Декларативное и процедурное значение программ

В рассмотренных выше примерах всегда было возможно понять результаты программы, не зная точно, как система фактически нашла эти результаты. Но иногда важно учитывать, как именно происходит поиск ответа в системе, поэтому имеет смысл проводить различие между двумя уровнями значения программ Prolog, а именно, между

- декларативным значением и
- процедурным значением.

Декларативное значение касается только отношений, определенных в программе. Поэтому декларативное значение регламентирует то, каким должен быть результат работы программы. С другой стороны, процедурное значение определяет также способ получения этого результата, иными словами, показывает, как фактически проводится обработка этих отношений системой Prolog.

Способность системы Prolog самостоятельно обрабатывать многие процедурные детали считается одним из ее особых преимуществ. Prolog побуждает программиста в первую очередь рассматривать декларативное значение программ, в основном независимо от их процедурного значения. А поскольку результаты программы в принципе определяются их декларативным значением, этого должно быть (по сути) достаточно для написания программы. Такая особенность важна и с точки зрения практики, так как декларативные аспекты программ обычно проще для понимания по сравнению с процедурными деталями. Но чтобы полностью воспользоваться этими преимуществами, программист должен сосредоточиться в основном на декларативном значении и, насколько это возможно, избегать необходимости отвлекаться на детали выполнения. Последние необходимо оставлять в максимально возможной степени для самой системы Prolog.

Такой декларативный подход фактически часто позволяет гораздо проще составлять программы на языке Prolog по сравнению с такими типичными процедурно-ориентированными языками программирования, как C или Pascal. Но, к сожалению, декларативный подход не всегда позволяет решить все задачи. По мере дальнейшего изучения материала этой книги станет очевидно, что процедурные аспекты не могут полностью игнорироваться программистом по практическим причинам, связанным с обеспечением вычислительной эффективности, особенно при разработке больших программ. Тем не менее необходимо стимулировать декларативный стиль мышления при разработке программ Prolog и игнорировать процедурные аспекты до такой степени, которая является допустимой с точки зрения практических ограничений.

Резюме

- *Программирование на языке Prolog* представляет собой процесс определения отношений и выдачи системе запросов об этих отношениях.
- Программа состоит из *предложений*, которые относятся к трем типам: факты, правила и вопросы.
- *Отношение* может быть определено на основе фактов путем задания *n*-элементных кортежей объектов, которые удовлетворяют отношению, или путем задания правил, касающихся этого отношения.
- *Процедура* представляет собой набор предложений, касающихся одного и того же отношения.
- *Выполнение запросов* об отношениях, передаваемых системе в виде вопросов, напоминает выполнение запросов к базе данных. Ответ системы Prolog на вопрос состоит из множества объектов, которые соответствуют этому вопросу.
- В системе Prolog для определения того, соответствует ли объект вопросу, часто применяется сложный процесс, который связан с выполнением логического вывода, рассмотрением многих альтернатив и, возможно, *перебора с возвратом*. Все эти операции выполняются системой Prolog автоматически и, в принципе, скрыты от пользователя.
- Различаются два значения *программ* Prolog: декларативное и процедурное. Декларативный подход является более привлекательным с точки зрения программирования. Тем не менее программисту также часто приходится учитывать процедурные детали.
- В данной главе представлены следующие понятия:
 - предложение, факт, правило, вопрос;
 - голова предложения, тело предложения;
 - рекурсивное правило, рекурсивное определение;
 - процедура;

- атом, переменная;
- конкретизация переменной;
- цель;
- цель, которая является достижимой, цель, которая достигнута;
- цель, которая является недостижимой, цель, которая не достигнута;
- перебор с возвратами;
- декларативное значение, процедурное значение.

Дополнительные источники информации

В различных реализациях Prolog используются разные синтаксические соглашения. Но большинство из них следует традициям так называемого *эдинбургского синтаксиса* (известного также как синтаксис DEC-10, сложившийся под влиянием реализации Prolog для компьютера DEC-10, которая оставила заметный след в истории развития этого языка [9], [122]). Кроме того, эдинбургский синтаксис лежит в основе международного стандарта ISO языка Prolog, ISO/IEC 13211-1 [43]. В настоящее время основные реализации Prolog главным образом совместимы с этим стандартом. В данной книге используется подмножество стандартного синтаксиса, если не считать некоторых небольших и незначительных различий. В тех редких случаях, когда допускаются такие различия, в соответствующем месте книги дается примечание на этот счет.

Синтаксис и значение программ Prolog

В этой главе...

2.1. Объекты данных	45
2.2. Согласование	52
2.3. Декларативное значение программ Prolog	56
2.4. Процедурное значение	58
2.5. Пример: обезьяна и банан	62
2.6. Порядок предложений и целей	66
2.7. Взаимосвязь между языком Prolog и логикой	73

В этой главе дано систематическое изложение синтаксиса и семантики основных понятий языка Prolog и представлены структурированные объекты данных. В ней рассматриваются следующие темы:

- простые объекты данных (атомы, числа, переменные);
- структурированные объекты;
- согласование как фундаментальная операция с объектами;
- декларативное (или непроцедурное) значение программы;
- процедурное значение программы;
- взаимосвязь между декларативным и процедурным значениями программы;
- модификация процедурного значения путем переупорядочения предложений и целей.

Большинство из этих тем уже рассматривалось в главе 1, а в данной главе их изложение становится более формальным и подробным.

2.1. Объекты данных

На рис. 2.1 представлена классификация объектов данных в языке Prolog. Система Prolog распознает тип объекта в программе по его синтаксической форме. Это возможно благодаря тому, что в синтаксисе языка Prolog определены разные формы для объектов данных каждого типа. В главе 1 уже рассматривался способ проведения различий между атомами и переменными: переменные начинаются с прописных букв, а атомы — со строчных букв. Системе Prolog не требуется сообщать какую-либо дополнительную информацию (наподобие объявления типа данных) для того, чтобы она распознала тип объекта.

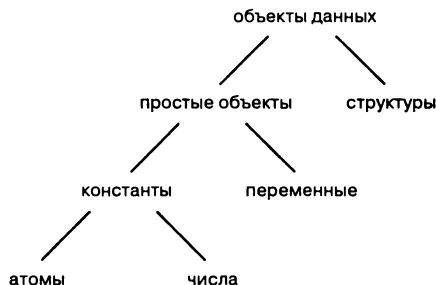


Рис. 2.1. Классификация объектов данных в языке Prolog

2.1.1. Атомы и числа

В главе 1 уже приводились некоторые простые примеры атомов и переменных. Но в целом они могут принимать более сложные формы, т.е. могут представлять собой строки, состоящие из следующих символов:

- прописные буквы A, B, ..., Z;
- строчные буквы a, b, ..., z;
- цифры 0, 1, 2, ..., 9;
- специальные символы, такие как "+", "-", "*", "/", "<", ">", "=", ":", ".", "&", "_", "~".

Атомы могут формироваться тремя перечисленными ниже способами.

1. Как строки букв, цифр и символов подчеркивания (" _ "), начинающиеся с прописной буквы:

```

anna
nil
x25
x_25
x_25AB
x_
x__y
alpha_beta_procedure
miss_Jones
sarah_jones

```

2. Как строки специальных символов:

```

<---->
=====>
...
.:.
::=

```

При использовании атомов в этой форме необходимо соблюдать осторожность, поскольку некоторые строки специальных символов уже имеют предопределенное значение; в качестве примера можно привести ":-".

3. Как строки символов, заключенных в одинарные кавычки. Такой формат является удобным, если требуется, например, применить атом, который начинается с прописной буквы. Заклучив его в кавычки, можно подчеркнуть его отличие от переменных:

```
'Tom'  
'South_America'  
'Sarah_Jones'
```

Числа, используемые в языке Prolog, подразделяются на целые числа и числа с плавающей точкой. Целые числа имеют простой синтаксис, как показано в следующих примерах:

```
1      1313      0      -97
```

Не все целые числа могут быть представлены в компьютере, поэтому диапазон целых чисел ограничен интервалом между некоторым наименьшим и наибольшим числами, которые допустимо использовать в конкретной реализации Prolog.

Предполагается, что для представления чисел с плавающей точкой применяется простой синтаксис, как показано в следующих примерах:

```
3.14      -0.0035      100.2
```

Обычно в программах на языке Prolog числа с плавающей точкой используются не очень часто. Причина этого состоит в том, что Prolog в основном предназначен для символьных, нечисловых вычислений. В символьных вычислениях часто применяются целые числа, например для подсчета количества элементов в списке, но необходимость в использовании чисел с плавающей точкой, как правило, возникает гораздо реже.

Кроме такого отсутствия необходимости использовать числа с плавающей точкой в типичных приложениях Prolog, есть еще одна причина, по которой следует избегать чисел с плавающей точкой. Как правило, необходимо стремиться к тому, чтобы смысл программ был как можно более очевидным. Но введение чисел с плавающей точкой иногда приводит к трудно диагностируемым нарушениям в работе программы из-за числовых ошибок, которые возникают при округлении во время выполнения арифметических операций. Например, при вычислении выражения

```
10000 + 0.0001 - 10000
```

может быть получен результат 0 вместо правильного результата 0.0001.

2.1.2. Переменные

Имена переменных представляют собой строки, состоящие из букв, цифр и символов подчеркивания. Они начинаются с прописной буквы или символа подчеркивания:

```
X  
Result  
Object2  
Participant_list  
ShoppingList  
_x23  
_23
```

Если переменная появляется в предложении только один раз, для нее не обязательно предусматривать имя. В этом случае можно использовать так называемую *анонимную* переменную, которая записывается как один символ подчеркивания. Например, рассмотрим следующее правило:

```
hasachild( X ) :- parent( X, _ ).
```

Это правило гласит: “Для всех *X*, *X* имеет ребенка, если *X* является родителем некоторого *Y*”. Итак, определено свойство *hasachild*, которое, как здесь подразумевается, не зависит от имени ребенка. Таким образом, переменная с обозначением ребенка — подходящее место для использования анонимной переменной. Следовательно, приведенное выше предложение может быть записано следующим образом:

```
hasachild( X ) :- parent( X, _ ).
```

Каждый раз при появлении в предложении символа подчеркивания он представляет новую анонимную переменную. Например, можно утверждать, что в мире есть

некто, имеющий ребенка, если существуют два таких объекта, один из которых является родителем другого:

```
somebody_has_child :- parent( _, _).
```

Это предложение эквивалентно следующему:

```
somebody_has_child :- parent( X, Y).
```

Но, безусловно, полностью отличается от следующего:

```
somebody_has_child :- parent( X, X).
```

Если анонимная переменная появляется в предложении вопроса, ее значение не выводится, когда система Prolog отвечает на вопрос. Если нас интересуют имена людей, имеющих детей, но не имена детей, то системе можно задать такой вопрос:

```
?- parent( X, _).
```

Лексическая область определения имен переменных представляет собой одно предложение. Это означает, например, что если имя `X15` встречается в двух предложениях, то оно обозначает две разные переменные. Но каждое вхождение `X15` в одном и том же предложении соответствует одной и той же переменной. Для констант ситуация будет иной: один и тот же атом всегда обозначает в любом предложении (а следовательно, и во всей программе) один и тот же объект.

2.1.3. Структуры

Структурированными объектами (или структурами) называются объекты, которые имеют несколько компонентов. Сами компоненты, в свою очередь, также могут быть структурами. Например, дата может рассматриваться как структура с тремя компонентами: число, месяц, год. Несмотря на то что структуры состоят из нескольких компонентов, они рассматриваются в программе как целостные объекты. Для соединения компонентов в целостный объект необходимо выбрать *функтор*. В данном примере подходящим функтором является `date`. Например, дату “1 мая 2001 года” можно записать следующим образом (рис. 2.2):

```
date( 1, may, 2001)
```



Рис. 2.2. Дата как пример структурированного объекта: а) представленного в виде дерева; б) рассматриваемого в той записи, которая применяется в языке Prolog

В этом примере все компоненты (два целых числа и один атом) являются константами. Компоненты могут также представлять собой переменные или другие структуры. Объект, который обозначает любое число мая, можно представить в виде следующей структуры:

```
date( Day, may, 2001)
```

Обратите внимание, что `Day` является переменной и может конкретизироваться значением любого объекта на каком-либо из следующих этапов выполнения программы.

Этот метод структурирования данных является простым и мощным. Он является одной из причин того, почему Prolog можно так естественно применять для решения проблем, связанных с символическими манипуляциями.

С точки зрения синтаксиса все объекты данных в языке Prolog представляют собой термы. Например, термами являются объекты

may

и

```
date( 1, may, 2001)
```

Все структурированные объекты можно представить графически в виде деревьев (см. рис. 2.2). Корнем дерева является функтор, а ветвями — компоненты. Если компонент представляет собой структуру, он становится поддеревом этого дерева, которое соответствует всему структурированному объекту.

В следующем примере показано, как могут использоваться структуры для представления некоторых простых геометрических объектов (рис. 2.3). Точка в двумерном пространстве определена двумя своими координатами; отрезок прямой определен двумя точками, а треугольник можно определить тремя точками. Предположим, что выбраны следующие функторы:

- point, для обозначения точек;
- seg, для обозначения отрезков прямых;
- triangle, для обозначения треугольников.

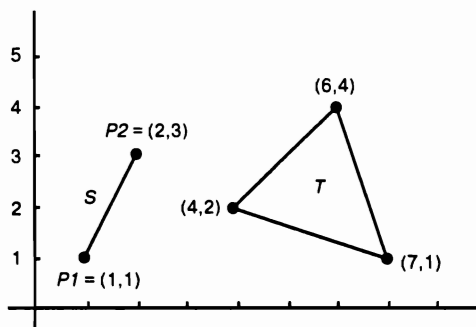


Рис. 2.3. Некоторые простые геометрические объекты

В этом случае объекты, изображенные на рис. 2.3, могут быть представлены следующим образом:

```
P1 = point(1,1)
P2 = point(2,3)
S = seg( P1, P2) = seg( point(1,1), point(2,3))
T = triangle( point(4,2), point(6,4), point(7,1))
```

Соответствующее древовидное представление этих объектов показано на рис. 2.4. Как правило, функтор, находящийся в корне дерева, называется *главным функтором терма*.

Если бы в той же программе были также точки в трехмерном пространстве, то для их представления можно было бы использовать другой функтор, скажем, point3:

```
point3( X, Y, Z)
```

Но допускается использовать одно и то же имя, в данном случае point, для точек в двух- и в трехмерном пространстве и, например, записывать:

```
point( X1, Y1) и point( X, Y, Z)
```

Если одно и то же имя встречается в программе в двух разных ролях, как в приведенном выше случае с функтором point, система Prolog определяет различие между ними по количеству параметров и интерпретирует само это имя как два функтора,

поскольку один из них имеет два параметра, а другой — три. Это связано с тем, что каждый функтор определяется следующими двумя признаками.

1. *Имя*, которое имеет такую же синтаксическую структуру, как и атомы.
2. *Аридность*, т.е. количество параметров.

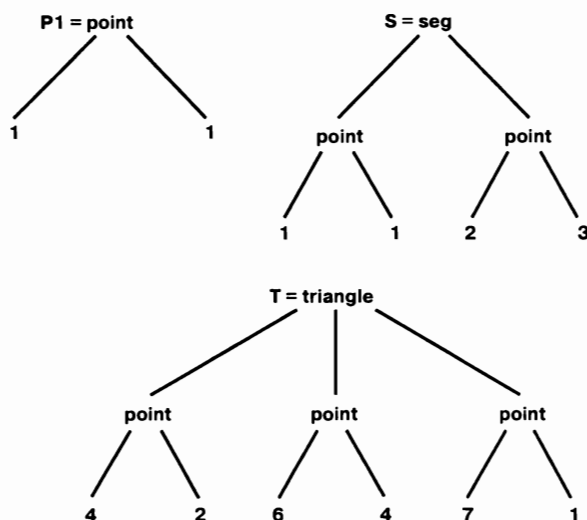


Рис. 2.4. Древовидное представление объектов, показанных на рис. 2.3

Как было описано выше, все структурированные объекты в языке Prolog представляют собой деревья, которые отображаются в программе с помощью термов. Рассмотрим еще два примера для иллюстрации того, как можно естественным образом представить с помощью термов Prolog сложные объекты данных. На рис. 2.5 показана древовидная структура, которая соответствует следующему арифметическому выражению:

$(a + b) * (c - 5)$

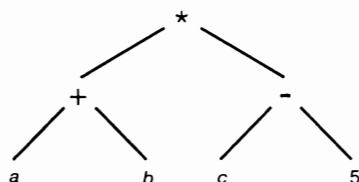


Рис. 2.5. Древовидная структура, соответствующая арифметическому выражению $(a + b) * (c - 5)$

В соответствии с синтаксисом термов, описанным выше, это выражение можно записать, используя символы “*”, “+” и “-” в качестве функторов, следующим образом:

$*(+(a, b), -(c, 5))$

Это, безусловно, допустимый терм Prolog, но обычно такая форма записи вряд ли является приемлемой. Люди, как правило, предпочитают использовать общеприня-

тую инфиксную систему обозначений, которая широко распространена в математике. В действительности язык Prolog также позволяет применять инфиксную систему обозначений таким образом, чтобы символы “*”, “+” и “-” записывались как инфиксные знаки операций. Подробные сведения о том, как программист может определить свои собственные операции, приведены в главе 3.

В качестве последнего примера рассмотрим некоторые простые электрические схемы (рис. 2.6). Справа на этом рисунке показаны древовидные представления соответствующих схем. Атомы r1, r2, r3 и r4 представляют собой имена резисторов, а функторы par и seq обозначают параллельные и последовательные соединения резисторов. Ниже перечислены соответствующие термы Prolog.

```
seq( r1, r2)
par( r1, r2)
par( r1, par( r2, r3))
par( r1, seq(par(r2,r3),r4))
```

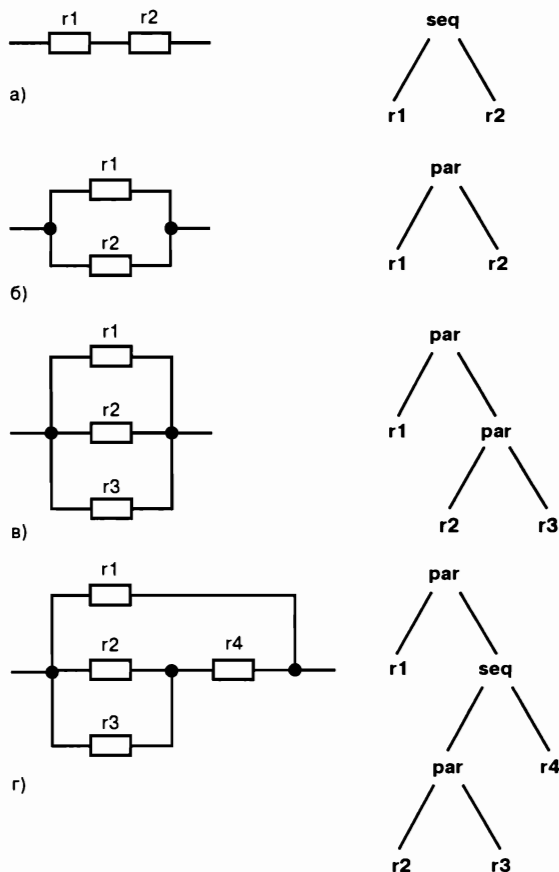


Рис. 2.6. Примеры простых электрических схем и их древовидных представлений: а) последовательное соединение резисторов r1 и r2; б) параллельное соединение двух резисторов; в) параллельное соединение трех резисторов; г) параллельное соединение резистора r1 и еще одной схемы

Упражнения

- 2.1. Какие из следующих объектов Prolog являются синтаксически правильными? К какому типу объектов они относятся (атом, число, переменная, структура)?
- а) Diana.
 - б) diana.
 - в) 'Diana'.
 - г) _diana.
 - д) 'Diana goes south'.
 - е) goes(diana, south).
 - ж) 45.
 - з) 5(X, Y).
 - и) +(north, west).
 - к) three(Black(Cats)).
- 2.2. Предложите формат представления для прямоугольников, квадратов и окружностей как структурированных объектов Prolog. Используйте подход, аналогичный продемонстрированному на рис. 2.4. Например, прямоугольник может быть представлен четырьмя точками (или, возможно, лишь тремя). Используя предложенные обозначения, подготовьте несколько примеров термов, которые представляют определенные объекты этих типов.

2.2. Согласование

В предыдущем разделе было показано, как могут использоваться термы для представления сложных объектов данных. Наиболее важной операцией с термами является согласование. Даже одно только согласование позволяет выполнять некоторые интересные вычисления.

Если даны два терма, то можно утверждать, что они согласуются, если выполнены следующие условия.

1. Они являются идентичными.
2. Переменные в обоих термах можно конкретизировать значениями объектов таким образом, чтобы после подстановки этих объектов вместо переменных термы стали идентичными.

Например, термы `date( D, M, 2001)` и `date(D1, may, Y1)` согласуются. Ниже показана одна конкретизация, при которой оба терма становятся идентичными.

- D конкретизируется значением D1.
- M конкретизируется значением may.
- Y1 конкретизируется значением 2001.

Этот вариант конкретизации можно записать более компактно в знакомой форме, в которой Prolog выводит результаты:

```
D = D1
M = may
Y1 = 2001
```

С другой стороны, термы `date( D, M, 2001)` и `date( D1, M1, 1444)` не согласуются; то же самое касается термов `date( X, Y, Z)` и `point( X, Y, Z)`.

Согласованием называется процесс, в котором в качестве входных данных берутся два терма и выполняется проверка того, являются ли они согласованными. Если термы не согласуются, это означает, что процесс согласования оканчивается неуда-

чей, а если они согласуются, то этот процесс завершается успешно и в нем осуществляется также конкретизация переменных в обоих термах такими значениями, что оба терма становятся идентичными.

Снова рассмотрим пример согласования двух дат. Запрос на выполнение этой операции можно передать системе Prolog в виде следующего вопроса с использованием знака операции “=”:

```
?- date( D, M, 2001) = date( D1, may, Y1).
```

Выше уже упоминалась конкретизация $D = D1$, $M = \text{may}$, $Y1 = 2001$, при которой достигается согласование. Но есть и другие варианты конкретизации, при которых оба терма становятся идентичными. Два из них приведены ниже.

```
D = 1
D1 = 1
M = may
Y1 = 2001
D = third
D1 = third
M = may
Y1 = 2001
```

Принято считать, что эти две конкретизации являются менее общими по сравнению с первой, поскольку они ограничивают значения переменных D и $D1$ более жестко, чем это необходимо. Для того чтобы оба терма в данном примере стали идентичными, требуется лишь предусмотреть применение одинаковых значений D и $D1$, хотя в качестве этого значения можно использовать что угодно. Согласование в языке Prolog всегда приводит к *наиболее общей конкретизации*. Таковой является конкретизация, которая ограничивает переменные в минимально возможной степени и тем самым оставляет максимально возможную свободу для дальнейшей конкретизации (на тот случай, если потребуются дальнейшее согласование). В качестве примера рассмотрим следующий вопрос:

```
?- date( D, M, 2001) = date( D1, may, Y1),
   date( D, M, 2001) = date( 15, M, Y).
```

Для достижения первой цели система Prolog конкретизирует переменные следующим образом:

```
D = D1
M = may
Y1 = 2001
```

После достижения второй цели конкретизация становится более определенной, как показано ниже.

```
D = 15
D1 = 15
M = may
Y1 = 2001
Y = 2001
```

Этот пример также показывает, что в ходе последовательного достижения цели переменные обычно конкретизируются все более определенными значениями.

Ниже приведены правила определения того, согласуются ли два терма, S и T .

1. Если S и T являются константами, то они согласуются, только если представляют собой одинаковый объект.
2. Если S представляет собой переменную, а T — нечто иное, то они согласуются и S конкретизируется значением T . И наоборот, если переменной является T , то T конкретизируется значением S .
3. Если S и T являются структурами, то они согласуются, только если выполняют следующие условия:

- а) S и T имеют одинаковый главный функтор;
 - б) все их соответствующие компоненты согласуются.
- Результирующая конкретизация определяется путем согласования компонентов.

Последнее из этих правил можно проиллюстрировать графически, рассматривая древовидное представление термов, как в примере, показанном на рис. 2.7. Процесс согласования начинается с корня (с главных функторов). Если оба функтора согласуются, процесс переходит к параметрам и осуществляется согласование пар соответствующих параметров. Итак, весь процесс согласования можно рассматривать как состоящий из показанной ниже последовательности (простых) операций согласования.

```
triangle = triangle,
point(1,1) = X,
A = point(4,Y),
point(2,3) = point(2,Z).
```

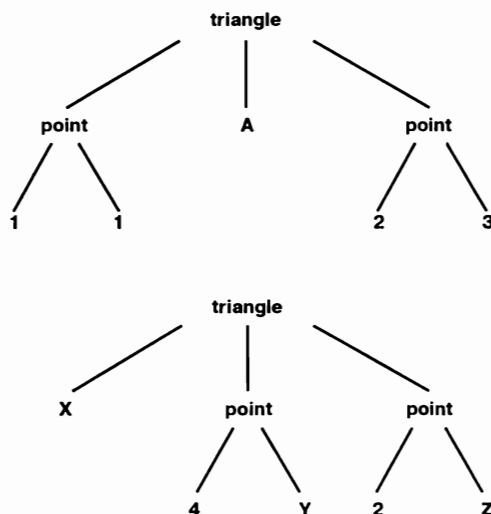


Рис. 2.7. Согласование термов запроса
`triangle( point (1,1), A, point (2,3)) =`
`triangle( X, point (4, Y), point (2, Z))`

Весь процесс согласования завершается успешно, поскольку успешными оказываются все операции согласования в этой последовательности. Результирующая конкретизация выглядит следующим образом:

```
X = point(1,1)
A = point(4,Y)
Z = 3
```

Следующий пример показывает, что даже одна только операция согласования может использоваться для выполнения некоторых интересных вычислений. Вернемся к примеру простых геометрических объектов (см. рис. 2.4) и определим фрагмент программы для распознавания горизонтальных и вертикальных отрезков прямых. *Вертикальность* является свойством отрезков, поэтому такое свойство можно формализовать в языке Prolog как унарное отношение. Пример, приведенный на рис. 2.8, позволяет понять, как следует формализовать это отношение. Отрезок является вертикальным, если координаты x его конечных точек равны; никакие иные ог-

раничения на отрезок прямой не накладываются. Свойство *горизонтальности* формулируется аналогичным образом, лишь меняются местами координаты x и y . Требуемое задание можно выполнить с помощью следующей программы, состоящей из двух фактов:

```
vertical( seg( point(X,Y), point(X,Y1) ).
horizontal( seg( point(X,Y), point(X1,Y) ) ).
```

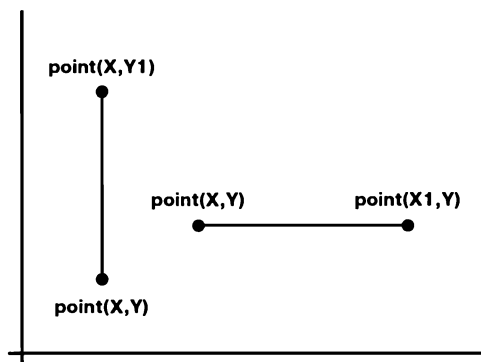


Рис. 2.8. Пример вертикального и горизонтального отрезков прямой

С этой программой может быть проведен следующий диалог:

```
?- vertical( seg( point(1,1), point(1,2) ) ).
yes
?- vertical( seg( point(1,1), point(2,Y) ) ).
no
?- horizontal( seg( point(1,1), point(2,Y) ) ).
Y = 1
```

На первый вопрос был получен ответ “yes”, поскольку цель вопроса согласуется с одним из фактов программы. Для ответа на второй вопрос не удалось найти какого-либо согласования. При ответе на третий вопрос переменной Y было присвоено значение 1 в результате согласования с фактом о горизонтальных отрезках.

Примером более общего вопроса к этой программе может служить вопрос о том, существуют ли какие-либо вертикальные отрезки, которые начинаются в точке $(2, 3)$?

```
?- vertical( seg(point(2,3),P) ).
P = point(2,Y)
```

Этот ответ означает, что таковыми являются любые отрезки прямой, оканчивающиеся в любой точке $(2, Y)$, иными словами, оканчивающиеся в любом месте на вертикальной прямой $x = 2$. Следует отметить, что фактически ответ системы Prolog может оказаться не таким изящным, как показано выше, но (в зависимости от используемой реализации Prolog) может выглядеть примерно так:

```
P = point(2, _136)
```

Тем не менее это различие является чисто внешним. В данном случае `_136` представляет собой переменную, которая не была конкретизирована, но `_136` — действительное имя переменной, которое было сформировано системой во время выполнения. Системе приходится формировать новые имена, чтобы обеспечить возможность переименовать пользовательские переменные в программе. Такая необходимость возникает по двум причинам: во-первых, в связи с тем, что одно и то же имя в разных предложениях обозначает различные переменные, и, во-вторых, из-за того, что при последовательных применениях одного и того же предложения каждый раз используется его “копия” с новым набором переменных.

Еще одним интересным вопросом к этой программе может служить вопрос о том, существуют ли отрезки, которые являются одновременно вертикальными и горизонтальными.

```
?- vertical( S), horizontal( S).  
S = seg( point(X,Y), point(X,Y) )
```

В этом положительном ответе системы Prolog на заданный вопрос фактически сказано, что любой отрезок, вырожденный в точку, имеет свойство быть вертикальным и горизонтальным одновременно. И в этом случае ответ был выработан просто в результате согласования. Как и в предыдущем случае, вместо имен переменных *X* и *Y* в ответе могут появиться некоторые имена, сформированные внутри программы.

Упражнения

2.3. Будут ли следующие операции согласования завершаться успешно или неудачно? Если они завершатся успешно, то каковы будут результаты конкретизации переменных?

а) `point( A, B) = point( 1, 2).`

б) `point( A, B) = point( X, Y, Z).`

в) `plus( 2, 2) = 4.`

г) `+( 2, D) = +( E, 2).`

д) `triangle( point(-1,0), P2, P3) = triangle( P1, point(1,0), point(0,Y)).`

В последнем примере результат конкретизации определяет семейство треугольников. Какое описание применимо к этому семейству?

2.4. Используя способ представления отрезков прямых, описанный в данном разделе, составьте терм, который представляет любой вертикальный отрезок на прямой линии, заданной уравнением $x = 5$.

2.5. Предположим, что прямоугольник представлен термом `rectangle( P1, P2, P3, P4)`, где переменными *P* обозначены вершины прямоугольника, заданные в определенном порядке. Определите отношение

`regular( R)`

которое является истинным, если *R* — прямоугольник с вертикальными и горизонтальными сторонами.

2.3. Декларативное значение программ Prolog

Как было показано в главе 1, могут рассматриваться два значения программ Prolog: декларативное и процедурное. В этом и следующем разделах приведены более формальные определения декларативного и процедурного значений программ в основном подмножестве языка Prolog. Но вначале снова проанализируем различие между этими двумя значениями. Рассмотрим следующее предложение:

`P :- Q, R.`

Здесь *P*, *Q* и *R* имеют синтаксис термов. Ниже приведены некоторые альтернативные декларативные прочтения этого предложения.

P является истинным, если *Q* и *R* истинны.
Из *Q* и *R* следует *P*.

Два альтернативных процедурных прочтения этого предложения приведены ниже.

Чтобы решить проблему *P*, вначале необходимо решить подпроблему *Q*,
а затем — подпроблему *R*.
Для достижения цели *P* вначале необходимо достичь *Q*, а затем — *R*.

Итак, различие между декларативными и процедурными прочтениями состоит в том, что последнее определяет не только логические соотношения между головой предложения и целями в его теле, но и задает порядок, в котором должны обрабатываться цели.

Рассмотрим формализованное определение декларативного значения.

Декларативное значение программ позволяет установить, является ли заданная цель истинной, а в случае положительного ответа — при каких значениях переменных она является истинной. Чтобы точно определить декларативное значение, необходимо ввести понятие экземпляра предложения. *Экземпляром предложения* S называется предложение S , в котором вместо каждой из его переменных подставлен некоторый терм. *Вариантом предложения* S называется такой экземпляр предложения S , где вместо каждой переменной подставлена другая переменная. Например, рассмотрим следующее предложение:

```
hasachild( X ) :- parent( X, Y ).
```

Ниже приведены два варианта этого предложения.

```
hasachild( A ) :- parent( A, B ).  
hasachild( X1 ) :- parent( X1, X2 ).
```

Экземплярами этого предложения являются следующие:

```
hasachild( peter ) :- parent( peter, Z ).  
hasachild( barry ) :- parent( barry, small(caroline)).
```

Если даны некоторая программа и цель G , то в декларативном значении неявно содержится приведенное ниже утверждение.

Цель G является истинной (т.е. достижимой, или логически следующей из программы), если и только если:

1. в программе имеется предложение S , такое, что
2. существует экземпляр I предложения S , такой, что
 - а) голова I идентична цели G и
 - б) все цели в теле I являются истинными.

Из этого определения можно вывести понятие вопроса Prolog следующим образом. Как правило, вопросом в системе Prolog является список целей, разделенных запятыми. Список целей является истинным, если все цели в списке являются истинными для одной и той же конкретизации переменных. Значения переменных становятся результатом наиболее общей конкретизации.

Поэтому запятая между целями обозначает конъюнкцию целей, при которой все они должны быть истинными. Но Prolog допускает также дизъюнкцию целей, при которой должна быть истинной лишь любая из целей. Дизъюнкция обозначается точкой с запятой. Например, предложение

```
P :- Q; R
```

имеет прочтение: P является истинным, если истинно Q или R . Поэтому значение этого предложения аналогично значению двух следующих предложений, вместе взятых:

```
P :- Q.  
P :- R.
```

Запятая связывает элементы предложения сильнее, чем точка с запятой. Поэтому предложение

```
P :- Q, R; S, T, U.
```

может рассматриваться следующим образом:

```
P :- ( Q, R ); ( S, T, U ).
```

и означает то же, что и предложения

```
P :- Q, R.  
P :- S, T, U.
```

Упражнения

2.6. Рассмотрим следующую программу:

```
f( 1, one).  
f( s(1), two).  
f( s(s(1)), three).  
f( s(s(s(X))), N) :-  
    f( X, N).
```

Как система Prolog ответит на следующие вопросы? Если это возможно, дайте несколько ответов, по меньшей мере два ответа.

- а) ?- f(s(1), A).
- б) ?- f(s(s(1)), two).
- в) ?- f(s(s(s(s(s(s(1)))))), C).
- г) ?- f(D, three).

2.7. Приведенная ниже программа утверждает, что два человека являются родственниками, если соблюдается по меньшей мере одно из следующих условий:

- а) один из них является предком другого,
- б) они имеют общего предка,
- в) они имеют общего потомка.

```
relatives( X, Y) :-  
    predecessor( X, Y).
```

```
relatives( X, Y) :-  
    predecessor( Y, X).
```

```
relatives( X, Y) :-          % X и Y имеют общего предка  
    predecessor( Z, X),  
    predecessor( Z, Y).
```

```
relatives( X, Y) :-          % X и Y имеют общего потомка  
    predecessor( X, Z),  
    predecessor( Y, Z).
```

Сократите эту программу, используя обозначение в виде точки с запятой.

2.8. Перепишите следующую программу без использования обозначения в виде точки с запятой.

```
translate( Number, Word) :-  
    Number = 1, Word = one;  
    Number = 2, Word = two;  
    Number = 3, Word = three.
```

2.4. Процедурное значение

Процедурное значение определяет, каким образом Prolog отвечает на вопросы. *Получить ответ на вопрос* означает попытаться достичь целей, заданных в списке. Их можно достичь, если переменные, которые встречаются в целях, могут быть конкретизированы таким образом, что цели логически следуют из программы. Поэтому процедурное значение Prolog определяет процедуру выполнения целей в списке применительно к заданной программе. Выражение “выполнить цель” означает попытаться ее достичь.

Обозначим процедуру выполнения целей как `execute`. Как показано на рис. 2.9, входы и выходы этой процедуры являются следующими:

- входы — программа и список целей;
- выходы — индикатор успеха/неудачи и конкретизация переменных.

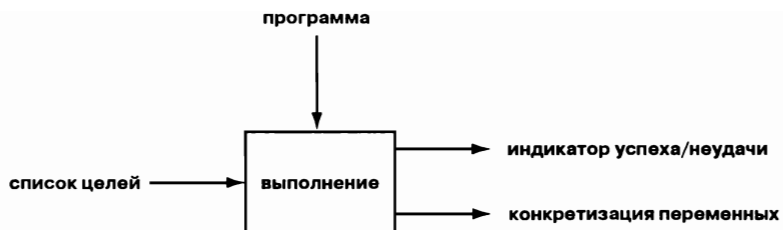


Рис. 2.9. Описание входов и выходов процедуры, которая выполняет список целей

Значение двух выходных результатов описано ниже.

1. Индикатор успеха/неудачи принимает значение “yes”, если цели являются достижимыми, и “no” в противном случае. Принято говорить, что “yes” указывает на успешное завершение, а “no” — на неудачное.
2. Конкретизация переменных вырабатывается только в случае успешного завершения; в случае неудачи конкретизация отсутствует.

В главе 1, в разделе 1.4, “Общие принципы поиска ответов на вопросы системой Prolog”, по сути было приведено неформальное описание того, что выполняет процедура `execute`. В остальной части этого раздела приведено более формальное и систематическое описание данного процесса, и его можно пропустить без серьезных опасений, что из-за этого будет затруднено понимание остального материала книги.

Конкретные операции процесса выполнения цели иллюстрируются на примере, приведенном в листинге 2.1. Прежде чем переходить к чтению следующего общего описания, следует ознакомиться с данным листингом.

Для выполнения следующего списка целей:

`G1, G2, ..., Gm`

процедура `execute` осуществляет описанные ниже действия.

- Если список целей пуст, то завершается успешно.
- Если список целей не пуст, то продолжает работу со (следующей) операции, называемой “SCANNING”.
- SCANNING. Сканировать предложения в программе от начала до конца до тех пор, пока не будет обнаружено первое предложение `C`, такое, что голова `C` согласуется с первой целью `G1`. Если такое предложение отсутствует, процедура оканчивается неудачей.

Если есть такое предложение `C` в форме

`H :- B1, ..., Bn.`

то процедура переименовывает переменные в `C` для получения варианта `C'` предложения `C`, такого, что `C'` и список `G1, ..., Gm` не имеют общих переменных. Допустим, что `C'` имеет вид

`H' :- B1', ..., Bn'.`

Процедура согласовывает цель `G1` и голову предложения `H'`; допустим, что результирующей конкретизацией переменных является `S`.

В списке целей `G1, G2, ..., Gm` процедура заменяет цель `G1` списком `B1', ..., Bn'`, что приводит к получению нового списка целей:

`B1', ..., Bn', G2, ..., Gm`

(Следует отметить, что если предложение С представляет собой факт, то $n = 0$ и новый список целей становится короче, чем первоначальный; такое сокращение списка целей может в конечном итоге привести к получению пустого списка и, тем самым, к успешному завершению.)

Затем процедура подставляет вместо переменных в новом списке целей новые значения, которые заданы в конкретизации S, что приводит к получению еще одного списка целей:

$B1'', \dots, Bn'', G2'', \dots, Gm''$

- После этого происходит переход к выполнению (рекурсивно, с помощью той же процедуры) этого нового списка целей. Если выполнение этого нового списка целей завершается успешно, то выполнение первоначального списка целей также завершается успешно. Если же выполнение нового списка целей оканчивается неудачей, то происходит отказ от этого нового списка целей и возврат через всю программу к операции SCANNING. Сканирование продолжается с предложения, которое непосредственно следует за предложением С (С — это предложение, которое использовалась перед этим), и осуществляется попытка достичь успешного завершения с помощью некоторого другого предложения.

Листинг 2.1. Пример, иллюстрирующий процедурное значение Prolog: трассировка выполнения процедуры `execute`

```
% Программа
big( bear).           % Предложение 1
big( elephant).       % Предложение 2
small( cat).          % Предложение 3

brown( bear).         % Предложение 4
black( cat).          % Предложение 5
gray( elephant).      % Предложение 6

dark( Z) :-           % Предложение 7 - все звери с черной шерстью
    black( Z).         % имеют темную окраску

dark( Z) :-           % Предложение 8 - все звери с коричневой шерстью
    brown( Z).         % имеют темную окраску
% Вопрос
? - dark( X), big( X). % Какой из зверей большой и имеет темную окраску?
% Трассировка выполнения
1) Начальный список целей: dark(X), big(X).
2) Сканировать программу сверху вниз, отыскивая предложение, голова которого
   согласуется с первой целью, dark(X). Обнаружено предложение 7:
dark(Z) :- black( Z).
Подставить вместо первой цели конкретизированное тело предложения 7, что
приводит к получению нового списка целей:
black( X), big( X)
3) Сканировать программу, чтобы найти согласование с целью black( X). Найдено
предложение 5: black(cat). Это предложение не имеет тела, поэтому список
целей после соответствующей конкретизации сокращается до
big( cat)
4) Сканировать программу, чтобы найти цель big( cat). Не обнаружено ни одного
предложения. Поэтому возвратиться к шагу (3) и отменить конкретизацию
X = cat. Теперь список целей снова принимает вид
black( X), big( X)
Продолжить сканирование программы ниже предложения 5. Не обнаружено ни одного
предложения. Поэтому вернуться к шагу (2) и продолжить сканирование ниже
предложения 7. Обнаружено предложение 8:
dark( Z) :- brown( Z).
Подставить brown( X) вместо первой цели в списке целей, что приводит
к получению списка
```


brown(X) , big(X) .

5) Сканировать программу, чтобы согласовать цель **brown(X)**, что приводит к обнаружению цели **brown(bear)**. Это предложение не имеет тела, поэтому список целей сокращается до

big(bear)

6) Сканировать программу, что приводит к обнаружению предложения **big(bear)**. Оно не имеет тела, поэтому список целей сокращается до пустого. Это указывает на успешное завершение, и соответствующая конкретизация переменной имеет вид

X = bear

Эта процедура записана более компактно в листинге 2.2 с использованием системы обозначений, аналогичной языку Pascal.

Здесь уместно привести несколько дополнительных замечаний, касающихся используемой формы представления процедуры **execute**. Прежде всего, в ней не было явно описано, как вырабатывается конечная результирующая конкретизация переменных. К успешному завершению привела именно конкретизация **S**, которая, возможно, могла быть в дальнейшем уточнена с помощью дополнительных конкретизаций, выполненных во вложенных рекурсивных вызовах процедуры **execute**.

При каждом неудачном завершении рекурсивного вызова **execute** выполнение возвращается к операции **SCANNING** и работа продолжается с того предложения **C** программы, которое использовалось перед этим в последнюю очередь. А поскольку использование предложения **C** не привело к успешному завершению, система Prolog вынуждена предпринять попытку перейти к обработке альтернативного предложения. При этом фактически происходит следующее: система Prolog отказывается от всей этой части неудачного выполнения и возвращается к той точке (к предложению **C**), с которой началась данная неудачная ветвь выполнения. После того как процедура возвращается к определенной точке, отменяются все конкретизации переменных, которые были выполнены после этой точки. Такая организация работы гарантирует, что Prolog систематически исследует все возможные альтернативные пути выполнения до тех пор, пока не будет найден тот из них, который в конечном итоге приведет к успеху, или пока не будет продемонстрировано, что все эти пути завершаются неудачей.

Как уже было сказано, даже после некоторого успешного завершения пользователь может вынудить систему вернуться к поиску дополнительных решений. В приведенном выше описании процедуры **execute** эта деталь была опущена.

Безусловно, в фактических реализациях Prolog необходимо добавить к процедуре **execute** несколько других усовершенствований. Одно из них состоит в сокращении объема сканирования предложений программы для повышения эффективности. Поэтому в практических реализациях Prolog осуществляется сканирование не всех предложений программы; рассматриваются только предложения, касающиеся данного отношения в текущей цели.

Упражнение

2.9. Рассмотрите программу, приведенную в листинге 2.1, и промоделируйте в стиле этого листинга выполнение системой Prolog процедуры поиска ответа на следующий вопрос:

```
?- big( X), dark( X).
```

Сравните полученную трассировку выполнения с приведенной в листинге 2.1, когда вопрос был по сути тем же самым, но цели были представлены в другом порядке:

```
?- dark( X), big( X).
```

В каком из этих двух случаев системе Prolog пришлось выполнить больше работы, прежде чем найти ответ?

Листинг 2.2. Процедура выполнения целей Prolog

```
procedure execute ( Program, GoalList, Success);
```

Входные параметры:

Program - список предложений

GoalList - список целей

Выходной параметр:

Success - истинностное значение; параметр Success становится равным true, если GoalList равен true применительно к Program

Локальные переменные:

Goal - цель

OtherGoals - список целей

Satisfied - истинностное значение

MatchOK - истинностное значение

Instant - конкретизация переменных

H, H', B1, B1', ..., Bn, Bn' - цели

Вспомогательные функции:

empty(L) - возвращает значение true, если L - пустой список

head(L) - возвращает первый элемент списка L

tail(L) - возвращает остальную часть списка L

append(L1,L2) - добавляет список L2 к концу списка L1

match(T1,T2,MatchOK,Instant) - предпринимает попытки согласовать термы T1 и T2; в случае успеха MatchOK становится равным true, а Instant содержит соответствующую конкретизацию переменных

substitute(Instant,Goals) - выполняет подстановку переменных в Goals в соответствии с конкретизацией Instant

```
begin
```

```
if empty(GoalList) then Success := true
```

```
else
```

```
begin
```

```
Goal := head(GoalList);
```

```
OtherGoals := tail(GoalList);
```

```
Satisfied := false;
```

```
while not Satisfied and "в программе имеются другие предложения" do
```

```
begin
```

```
Допустим, что следующим предложением в Program является
```

```
H :- B1, ..., Bn.
```

```
Сформировать вариант этого предложения
```

```
H' :- B1', ..., Bn'.
```

```
match(Goal,H',MatchOK,Instant);
```

```
if MatchOK then
```

```
begin
```

```
NewGoals := append([B1',... ,Bn'], OtherGoals);
```

```
NewGoals := substitute(Instant,NewGoals);
```

```
execute(Program,NewGoals,Satisfied)
```

```
end
```

```
end;
```

```
Success := Satisfied
```

```
end
```

```
end;
```

2.5. Пример: обезьяна и банан

Задача с обезьяной и бананом используется как простой пример решения проблемы. Приведенная в данном разделе программа Prolog, предназначенная для решения этой задачи, показывает, каким образом в подобных упражнениях могут использоваться механизмы согласования и перебора с возвратами. Вначале программа будет разработана непроцедурным способом, а затем подробно исследовано ее процедурное поведение. Предполагается, что эта программа должна быть компактной и наглядной.

В данном разделе используется следующий вариант задачи. Перед дверью, ведущей в комнату, находится обезьяна. В середине комнаты с потолка свисает банан. Обезьяна голодна и хочет получить банан, но не может дотянуться до него с пола. У окна комнаты стоит ящик, которым может пользоваться обезьяна. Обезьяна может выполнять следующие действия: ходить по полу, залезать на ящик, передвигать ящик (если она уже находится рядом с ящиком) и срывать банан, если она стоит на ящике непосредственно под бананом. Может ли обезьяна получить этот банан?

Одной из важных задач в программировании является поиск одного из представлений проблемы в терминах используемого языка программирования. В данном случае можно всегда рассматривать “мир, в котором существует обезьяна”, как находящийся в определенном состоянии, которое может изменяться во времени. Текущее состояние определяется положением объектов. Например, начальное состояние этого мира описано ниже.

1. Обезьяна находится у двери.
2. Обезьяна находится на полу.
3. Ящик стоит у окна.
4. Обезьяна не имеет банана.

Было бы удобно объединить все эти четыре фрагмента информации в один структурированный объект. Выберем в качестве функтора слово *state* (состояние), чтобы все четыре компонента описания состояния хранились вместе. На рис. 2.10 изображено начальное состояние, представленное как структурированный объект.

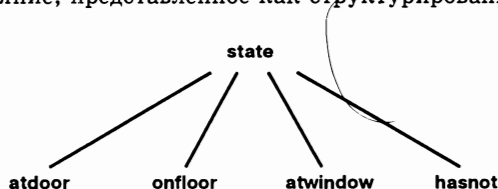


Рис. 2.10. Начальное состояние мира обезьяны, представленное как структурированный объект. Четырьмя его компонентами являются: положение обезьяны на плоскости, положение обезьяны в пространстве (на ящике или на полу), положение ящика, отсутствие или наличие у обезьяны банана

Данную проблему можно рассматривать как игру с одним участником. Формализуем правила этой игры. Во-первых, целью этой игры является ситуация, в которой обезьяна имеет банан; иными словами, любое состояние, в котором последним компонентом является информация о наличии банана:

```
state ( _, _, _, has)
```

Во-вторых, необходимо рассмотреть, каковы допустимые действия, после выполнения которых мир переходит из одного состояния в другое. Это — действия следующих четырех типов.

1. Схватить банан (*grasp*).
2. Залезть на ящик (*climb*).
3. Передвинуть ящик (*push*).
4. Перейти из одного места в другое (*walk*).

Не все действия возможны во всех возможных состояниях мира. Например, действие “схватить банан” возможно, только если обезьяна стоит на ящике непосредственно под бананом (который находится в середине комнаты) и еще не схватила ба-

нан. Эти правила можно формально представить на языке Prolog в виде следующего трехместного отношения, обозначенного как `move`:

```
move( State1, Move, State2)
```

Поэтому три параметра этого отношения определяют действие следующим образом:

```
State1  —————> State2
      Move
```

`State1` — это состояние до выполнения действия, `Move` — выполняемое действие и `State2` — состояние после выполнения действия.

Действие `grasp` с его обязательной предпосылкой, определяемой состоянием перед этим действием, может быть определено с помощью следующего предложения:

```
move( state( middle, onbox, middle, hasnot),    % Перед выполнением действия
      grasp,                                     % Действие
      state( middle, onbox, middle, has) ).      % После выполнения действия
```

Этот факт говорит о том, что после действия, в результате которого обезьяна получает банан, она остается на ящике в середине комнаты.

Аналогичным образом можно выразить тот факт, что обезьяна может передвигаться по полу из любого горизонтального положения `Pos1` в любое положение `Pos2`. Обезьяна может выполнять это действие независимо от положения ящика, а также от того, схватила она банан или нет. Все эти условия могут быть определены с помощью следующего факта Prolog:

```
move( state( Pos1, onfloor, Box, Has),
      walk( Pos1, Pos2),                                     % Перейти из Pos1 в Pos2
      state( Pos2, onfloor, Box, Has) ).
```

Обратите внимание, что это предложение говорит о многом, например о следующем:

- было выполнено действие “перейти из некоторой позиции `Pos1` в некоторую позицию `Pos2`”;
- обезьяна находится на полу до и после выполнения этого действия;
- ящик находится в некоторой точке `Box`, которая остается неизменной после этого действия;
- состояние “наличия банана” `Has` после этого действия остается неизменным.

Данное предложение фактически определяет целое множество возможных действий, поскольку оно применимо к любой ситуации, которая согласуется с указанным состоянием, предшествующим этому действию. Поэтому подобную спецификацию иногда называют также *схемой движения*. Такие схемы можно легко запрограммировать на языке Prolog с использованием переменных Prolog.

Другие два типа действий, `push` и `climb`, можно определить аналогичным образом.

Рассматриваемая программа должна отвечать на вопросы такого основного типа: может ли обезьяна в некотором начальном состоянии `State` получить банан? Такие вопросы можно формально представить в виде следующего предиката:

```
canget( State)
```

Здесь параметр `State` представляет собой одно из состояний мира обезьяны. Программа определения предиката `canget` может быть основана на двух приведенных ниже наблюдениях.

1. Для любого состояния, в котором обезьяна уже имеет банан, предикат `canget`, безусловно, должен быть истинным; в таком случае не требуются какие-либо действия. Это соответствует следующему факту Prolog:

```
canget( state( _, _, _, has) ).
```

2. В других случаях необходимо выполнить одно или несколько действий. Обезьяна может получить банан в любом состоянии `State1`, если есть некоторое действие `Move` для перехода из состояния `State1` в некоторое состояние `State2`, такое, что обезьяна затем может получить банан в состоянии `State2`

(выполнив от нуля и больше действий). Этот принцип проиллюстрирован на рис. 2.11. Предложение Prolog, которое соответствует этому правилу, приведено ниже.

```
canget( State1) :-
    move( State1, Move, State2),
    canget( State2).
```

На этом завершается разработка программы, которая показана в листинге 2.3.

Формулировка предиката `canget` является рекурсивной и аналогична формулировке отношения `predecessor`, описанного в главе 1 (сравните рис. 1.7 и 2.11). Такой принцип используется в языке Prolog снова и снова.

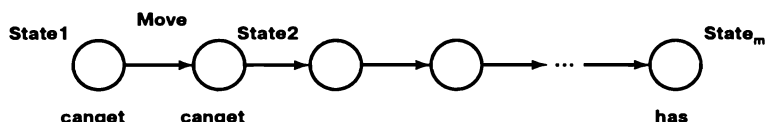


Рис. 2.11. Рекурсивная формулировка предиката `canget`

Листинг 2.3. Программа решения задачи с обезьяной и бананом

```
% move( State1, Move, State2) - выполнение действия Move в состоянии State1
% обеспечивает переход в состояние State2; состояние представлено термом:
% state( MonkeyHorizontal, MonkeyVertical, BoxPosition, HasBanana)
move( state( middle, onbox, middle, hasnot),      % До выполнения действия
    grasp,                                       % Схватить банан
    state( middle, onbox, middle, has) ).        % После выполнения действия
move( state( P, onfloor, P, H),
    climb,                                       % Забраться на ящик
    state( P, onbox, P, H) ).
move( state( P1, onfloor, P1, H),
    push( P1, P2),                             % Передвинуть ящик из позиции P1 в позицию P2
    state( P2, onfloor, P2, H) ).

move( state( P1, onfloor, B, H),
    walk( P1, P2),                             % Перейти из позиции P1 в позицию P2
    state( P2, onfloor, B, H) ).

% canget( State) - обезьяна может получить банан в состоянии State
canget( state( _, _, _, has) ). % Предложение can1 - обезьяна уже получила банан
canget( State1) :-              % Предложение can2 - обезьяна выполняет
    move( State1, Move, State2), % определенное действие, чтобы его получить
    canget( State2).             % Выполнить определенное действие
                                % Теперь получить банан
```

Итак, программа решения задачи с обезьяной и бананом разработана непроцедурным способом. Теперь рассмотрим ее процедурное поведение на примере следующего вопроса к программе:

```
?- canget( state( atdoor, onfloor, atwindow, hasnot)).
```

Система Prolog на этот вопрос отвечает “yes”. Процесс, осуществляемый системой Prolog для достижения этого ответа, проходит, согласно процедурной семантике Prolog, через последовательность списков целей. Он предусматривает выполнение определенного поиска правильных шагов среди возможных альтернативных шагов. В некоторый момент времени этот поиск может привести к выбору неправильного действия, которое ведет к тупиковой ветви. На этом этапе возобновить работу программы поможет возврат к предыдущему этапу. Такой процесс поиска иллюстрируется на рис. 2.12.

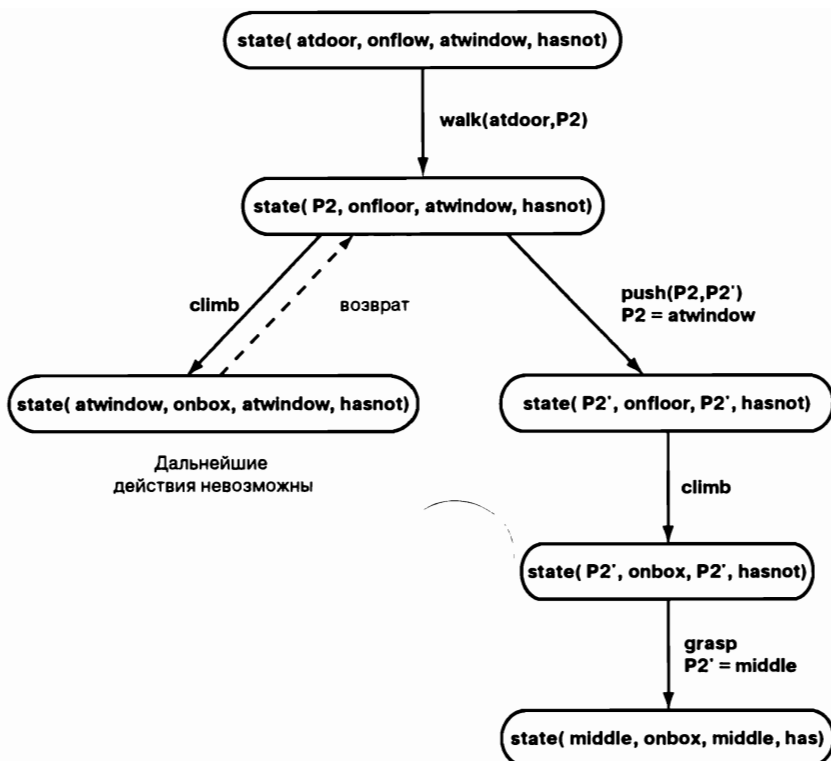


Рис. 2.12. Поиск обезьяной способа получения банана. Поиск начинается с верхнего узла и проходит в направлении вниз, как показано на рисунке. Попытки выполнения альтернативных действий предпринимаются в последовательности слева направо. Возврат к предыдущему этапу выполняется только один раз

Для ответа на этот вопрос системе Prolog пришлось вернуться к предыдущему этапу только один раз. Правильная последовательность действий была найдена почти сразу же. Причиной такой высокой эффективности программы послужил правильный выбор порядка, в котором в программе представлены предложения, касающиеся отношения move. Порядок в данном случае (к счастью) оказался вполне приемлемым. Но возможны и менее удачные варианты упорядочения. Согласно правилам этой игры, обезьяна вполне может также переходиться с места на место, не прикасаясь к ящику, или бесцельно передвигать ящик по комнате. Как показано в следующем разделе, при более тщательном исследовании обнаруживается, что в случае рассматриваемой программы порядок расположения предложений фактически имеет решающее значение.

2.6. Порядок предложений и целей

2.6.1. Опасность возникновения бесконечных циклов

Рассмотрим следующее предложение:

`p :- p.`

В этом предложении содержится утверждение о том, что “*p* является истинным, если *p* является истинным”. Это утверждение с декларативной точки зрения является полностью правильным, но с процедурной точки зрения — совершенно бесполезным. В действительности подобное предложение может привести к возникновению проблемы в системе Prolog. Рассмотрим следующий вопрос:

?- *p*.

С использованием приведенного выше предложения цель *p* заменяется такой же целью *p*, которая, в свою очередь, заменяется целью *p*, и т.д. В таком случае система Prolog входит в бесконечный цикл и не может обнаружить, что отсутствует какой-либо прогресс.

Этот пример показывает простой способ, который вынуждает систему Prolog войти в бесконечный цикл. Но аналогичные циклы могли бы возникать в некоторых из предыдущих примеров программ после изменения порядка расположения предложений или порядка расположения целей в предложениях. Рассмотрим несколько примеров.

В программе решения задачи с обезьяной и бананом предложения, касающиеся отношения *move*, были упорядочены следующим образом: *grasp*, *climb*, *push*, *walk* (возможно, для полноты следовало бы добавить предложение *unclimb* — слезть). В этих предложениях утверждается, что обезьяна имеет возможность схватить банан, залезть на ящик и т.д. Согласно процедурной семантике Prolog, порядок предложений указывает на то, что обезьяна предпочитает схватить банан, а не лезть на ящик, не двигать его и т.д. Такой порядок предпочтений фактически помогает обезьяне решить проблему. Но что могло бы произойти, если бы этот порядок был иным? Предположим, что предложение “*walk*” стоит на первом месте. На этот раз выполнение первоначальной цели предыдущего раздела:

```
?- canget( state( atdoor, onfloor, atwindow, hasnot) ).
```

привело бы к получению следующей трассировки выполнения. Первые четыре списка целей (с переменными, переименованными соответствующим образом) остаются такими же, как и прежде.

```
1) canget( state( atdoor, onfloor, atwindow, hasnot) )
```

Применяется второе предложение *canget('can2')*, что приводит к получению приведенного ниже результата.

```
2) move( state( atdoor, onfloor, atwindow, hasnot), M', S2' ),  
   canget( S2' )
```

В результате выполнения действия *walk(atdoor, P2')* будет получен следующий результат:

```
3) canget( state( P2', onfloor, atwindow, hasnot) )
```

После повторного применения предложения *can2* список целей принимает вид

```
4) move( state( P2', onfloor, atwindow, hasnot), M'', S2'' ),  
   canget( S2'' )
```

Теперь ситуация изменяется. В настоящее время первым предложением, голова которого согласуется с первой целью, приведенной выше, становится *walk* (а не *climb*, как перед этим). Соответствующая конкретизация такова:

```
S2'' = state( P2'', onfloor, atwindow, hasnot)
```

Поэтому список целей принимает вид

```
5) canget( state( P2'', onfloor, atwindow, hasnot) )
```

Применяя предложение *can2*, получим следующее:

```
6) move( state( P2'', onfloor, atwindow, hasnot), M''', S2''' ),  
   canget( S2''' )
```

И снова в первую очередь предпринимается попытка применить предложение *walk*, что приводит к получению следующего результата:

```
7) canget( state( P2''', onfloor, atwindow, hasnot) )
```

Теперь сравним цели 3), 5) и 7). Они являются одинаковыми, за исключением одной переменной; эта переменная последовательно принимает вид P' , P'' и P''' . Как известно, успех достижения цели не зависит от конкретных имен переменных в цели. Это означает, что, начиная со списка целей 3), трассировка выполнения не обнаруживает никакого прогресса. Фактически можно видеть, что просто повторно применяются два предложения, `can2` и `walk`. Обезьяна ходит по комнате, даже не пытаясь использовать ящик. Поскольку отсутствует какой-либо прогресс, эти действия (теоретически) могут продолжаться до бесконечности; система Prolog не получает информации о том, что нет смысла продолжать в том же духе.

Как показывает этот пример, иногда система Prolog пытается решить проблему таким способом, но решение так не достигается, несмотря на то, что оно существует. Подобные ситуации не являются в программировании на языке Prolog чем-то необычным. Но бесконечные циклы не являются также чем-то исключительным и в других языках программирования. Необычным по сравнению с другими языками программирования является то, что программа Prolog может быть правильной с декларативной точки зрения, но вместе с тем неправильной с процедурной точки зрения, в том смысле, что она не способна выработать ответ на вопрос. В таких случаях система Prolog может оказаться неспособной достичь цели, поскольку она пытается выработать ответ, выбирая неверный путь.

В таком случае возникает вполне резонный вопрос о том, нельзя ли внести в программу какие-либо более существенные изменения, чтобы коренным образом исключить любую угрозу образования циклов, или приходится всегда полагаться на правильное упорядочение предложений и целей. Как оказалось, программы, особенно большие, были бы слишком ненадежными, если бы в них приходилось рассчитывать только на наличие некоторого приемлемого упорядочения. Поэтому предусмотрено несколько других методов, которые исключают возможность возникновения бесконечных циклов, и они являются гораздо более общими и надежными, чем сам метод упорядочения. Эти методы будут регулярно использоваться в остальной части данной книги, особенно в тех главах, в которых рассматриваются темы выбора путей, решения задач и поиска информации.

2.6.2. Модификация программы путем переупорядочения предложений и целей

Даже в простейших примерах программ, приведенных в главе 1, была скрыта опасность поведения, связанного с заикливанием. Например, в главе 1 была приведена следующая программа, которая определяет отношение `predecessor`:

```
predecessor( Parent, Child) :-  
    parent( Parent, Child).  
  
predecessor( Predecessor, Successor) :-  
    parent( Predecessor, Child),  
    predecessor( Child, Successor).
```

Проанализируем некоторые модификации этой программы. Безусловно, что все эти модификации должны иметь одно и то же декларативное значение, но разное процедурное значение. Согласно декларативной семантике Prolog, без изменения декларативного значения этой программы в нее можно внести следующие изменения.

1. Изменить порядок предложений в программе.
2. Изменить порядок целей в телах предложений.

Процедура `predecessor` состоит из двух предложений, а одно из них имеет две цели в своем теле. Поэтому могут быть сформированы четыре варианта этой программы, имеющих одно и то же декларативное значение. Эти четыре варианта могут быть получены следующим образом.

1. В результате перестановки местами обоих предложений.

2. В результате перестановки местами целей при каждом выбранном порядке предложений.

Соответствующие четыре процедуры, `pred1`, `pred2`, `pred3` и `pred4`, приведены в листинге 2.4.

Листинг 2.4. Четыре версии программы `predecessor`

```
% Четыре версии программы predecessor
% Первоначальная версия
pred1( X, Z) :-
    parent( X, Z).

pred1( X, Z) :-
    parent( X, Y),
    pred1( Y, Z).

% Версия А: поменять местами предложения первоначальной версии
pred2( X, Z) :-
    parent( X, Y),
    pred2( Y, Z).

pred2( X, Z) :-
    parent( X, Z).

% Версия Б: поменять местами цели во втором предложении первоначальной версии
pred3( X, Z) :-
    parent( X, Z).

pred3( X, Z) :-
    pred3( X, Y),
    parent( Y, Z).

% Версия В: поменять местами цели и предложения первоначальной версии
pred4( X, Z) :-
    pred4( X, Y),
    parent( Y, Z).

pred4( X, Z) :-
    parent( X, Z).
```

В поведении этих четырех процедур, эквивалентных с декларативной точки зрения, наблюдаются важные различия. Для демонстрации этого рассмотрим отношение `parent` (см. рис. 1.1 в главе 1). Итак, что произойдет при обработке вопроса о том, является ли Том предком Пэт, с использованием четырех вариантов отношения `predecessor`, как показано ниже.

```
?- pred1( tom, pat).
yes
?- pred2( tom, pat).
yes
?- pred3( tom, pat).
yes
?- pred4( tom, pat).
```

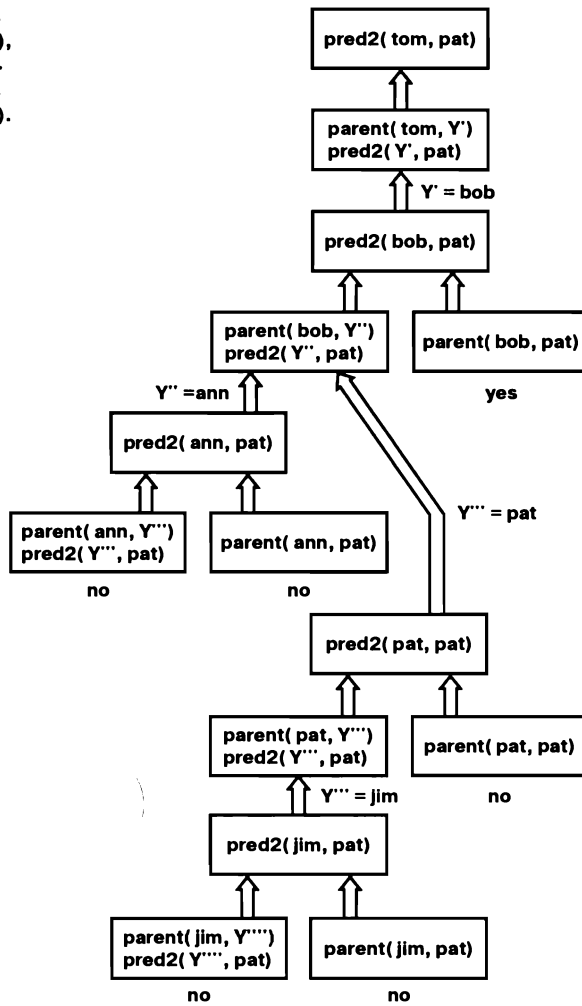
В последнем случае система Prolog не может найти ответ. Это выражается в том, что система Prolog выводит на терминал примерно такое сообщение, как "More core needed" (Нехватка оперативной памяти) или "Stack overflow" (Переполнение стека).

Трассировка выполнения программы `pred1` (называемой в главе 1 как `predecessor`), полученная при обработке указанного вопроса, показана на рис. 1.10 в главе 1. На рис. 2.13 приведены соответствующие трассировки для `pred2`, `pred3` и `pred4`. На рис. 2.13, в наглядно показано, что нельзя надеяться на успешное завершение работы программы `pred4`, а на рис. 2.13, а видно, что программа `pred2` является довольно неэффективной по сравнению с `pred1`, поскольку она выполняет намного больше операций поиска и возврата в генеалогическом древе.

```

pred2(X, Z) :-
  parent(X, Y),
  pred2(Y, Z).
pred2(X, Z) :-
  parent(X, Z).

```

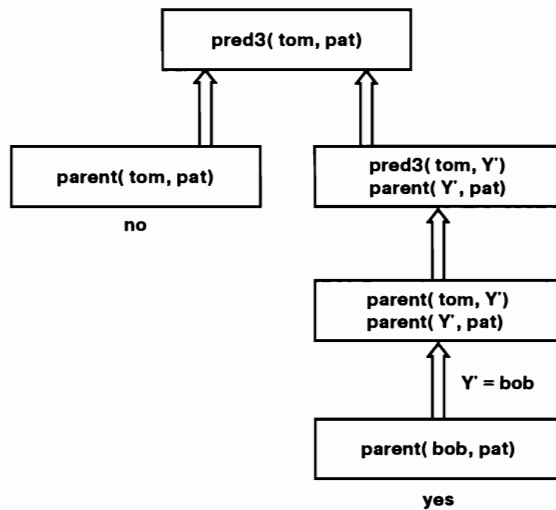


a)

```

pred3( X, Z) :-
  parent( X, Z).
pred3( X, Z) :-
  pred3( X, Y),
  parent(Y, Z).

```

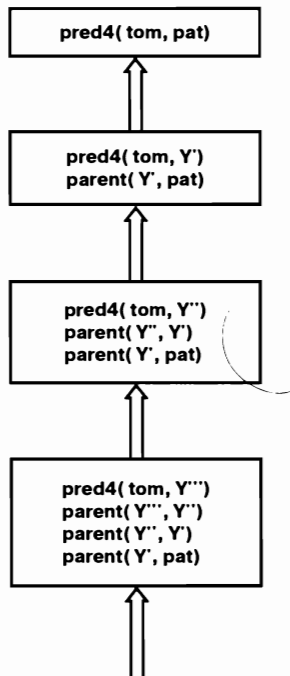


6)

```

pred4( X, Z) :-
  pred4( X, Y),
  parent( Y, Z).
pred4( X, Z) :-
  parent( X, Z).

```



в)

Рис. 2.13. Поведение трех вариантов отношения predecessor при поиске ответа на вопрос о том, является ли Том предком Пэт

Это сравнение должно напоминать нам об общем проверенном на практике эвристическом правиле в области решения задач: обычно лучше всего вначале пытаться проверить простейшую идею. В данном случае все версии отношения predecessor основаны на следующих идеях.

1. Наиболее простая идея состоит в том, что нужно проверить, соответствуют ли отношению `parent` два параметра отношения `predecessor`.
2. Более сложная идея состоит в том, чтобы найти кого-то, кто связывает друг с другом обоих людей (того, кто связан с ними отношениями, с одной стороны, `parent`, а с другой стороны, `predecessor`).

Из всех этих четырех вариантов отношения `predecessor` только в варианте `pred1` в первую очередь осуществляются простейшие действия. В отличие от этого, в варианте `pred4` в первую очередь всегда предпринимается попытка выполнить сложные действия, а варианты `pred2` и `pred3` лежат между этими двумя крайностями. Поэтому даже без подробного изучения трассировок выполнения следует предпочесть вариант `pred1` просто на основании правила, согласно которому вначале следует пытаться выполнить простейшие действия. Это правило и в целом может служить полезным руководящим указанием в программировании.

Рассматриваемые четыре варианта процедуры `predecessor` можно также сравнить на основании изучения того, на какие типы вопросов позволяют находить ответы конкретные варианты программы и на какие типы вопросов они не дают ответов. Как оказалось, и вариант `pred1`, и вариант `pred2` позволяют достичь ответа на вопросы любого типа о предках; вариант `pred4` ни при каких условиях не позволяет достичь ответа, а вариант `pred3` иногда позволяет найти ответ, а иногда нет. Ниже приведен один пример вопроса, на который вариант `pred3` не позволяет найти ответ.

```
?- pred3( liz, jim).
```

Этот вопрос снова вводит систему в бесконечную последовательность рекурсивных вызовов. Поэтому вариант `pred3` также нельзя считать правильным с процедурной точки зрения.

2.6.3. Объединение декларативного и процедурного представлений

В предыдущем разделе было показано, что порядок целей и предложений действительно играет важную роль. Кроме того, есть такие программы, которые являются правильными с декларативной точки зрения, но на практике не работают. Подобные несоответствия между декларативным и процедурным значениями могут оказаться весьма неприятными. В таком случае напрашивается вопрос, почему бы не забыть о существовании декларативного значения? Эту дискуссию можно довести до абсурда, рассмотрев такое предложение, как

```
predecessor( X, Z) :- predecessor( X, Z).
```

которое является правильным с декларативной точки зрения, но полностью бесполезным в работающей программе.

Причина, по которой не следует забывать о декларативном значении, состоит в том, что прогресс в технологии программирования обычно достигается путем ухода от процедурных деталей в направлении к декларативным аспектам, которые обычно проще поддаются формальному описанию и являются более легкими для понимания. Вся тяжесть проработки процедурных деталей должна быть возложена на саму систему, а не на программиста. Система Prolog может оказать помощь в решении этой задачи, но, как было описано в этом разделе, она помогает лишь частично: иногда эта система сама прорабатывает процедурные детали должным образом, а иногда неспособна это сделать. Многие программисты придерживаются взглядов, что лучше иметь в программе хоть какое-то декларативное значение, чем вообще никакого (последнее относится к большинству других языков программирования). Такой принцип имеет большую практическую значимость, поскольку чаще всего довольно легко заставить программу успешно работать, добившись от нее, чтобы она стала правильной с декларативной точки зрения. Поэтому полезный практический подход, который часто позволяет достичь результатов, состоит в том, что вначале необходимо

сосредоточиться на декларативных аспектах проблемы, затем проверить полученную программу, а если произойдет ее отказ, связанный с процедурным аспектом, попытаться переупорядочить предложения и цели, определив более приемлемую их последовательность.

2.7. Взаимосвязь между языком Prolog и логикой

Язык Prolog имеет непосредственное отношение к математической логике, поэтому его синтаксис и значение программ можно определить более кратко с использованием логических обозначений. И действительно, синтаксис языка Prolog часто определяют именно таким образом. Но подобное введение в Prolog предполагает, что читатель знаком с некоторыми понятиями математической логики. С другой стороны, безусловно, что знакомство с этими понятиями не требуется для изучения и использования языка Prolog как инструментального средства программирования, а в этом и состоит назначение данной книги. Для тех читателей, кто особенно заинтересован в изучении взаимосвязи между языком Prolog и логикой, ниже приведены некоторые основные указания, касающиеся определения этого языка в терминах математической логики, наряду с некоторыми подходящими источниками информации.

Синтаксис Prolog — это синтаксис предложений (формул) логики предикатов первого порядка, записанных в так называемой *форме предложений* (в конъюнктивной нормальной форме, в которой кванторы не записываются явно) и дополнительно ограниченных *хорновскими предложениями* (формулами логики предикатов первого порядка, которые имеют самое большое один положительный литерал, называемыми также хорновскими дизъюнктами). В [32] опубликована программа Prolog, которая преобразует формулы исчисления предикатов первого порядка в форму предложений. Процедурное значение программы Prolog основано на принципе резолюции, применимом для автоматического доказательства теорем, который был представлен Робинсоном в его классической статье [131]. В языке Prolog для доказательства теоремы резолюции используется специальная стратегия, называемая SLD. Введение в проблематику исчисления предикатов первого порядка и доказательства теорем на основе резолюций можно найти в нескольких книгах, посвященных общим вопросам искусственного интеллекта [58]; [60]; [126]; [133]; см. также [51]). Математические проблемы, касающиеся свойств процедурного значения Prolog по отношению к логике, проанализированы в [89].

Операция согласования в языке Prolog соответствует тому действию, которое в логике называют *унификацией*. Но мы избегаем использования термина “унификация”, поскольку в большинстве систем Prolog по требованиям повышения эффективности согласование реализовано таким способом, что оно не совсем точно соответствует унификации (см. упр. 2.10). Но с практической точки зрения согласование вполне можно рассматривать как операцию, аналогичную унификации. Тем не менее для правильной унификации требуется проведение так называемой *проверки вхождения* (occurs check), при которой определяется, входит ли конкретная переменная в указанный терм. Если бы такая проверка вхождения применялась при согласовании, эта операция стала бы неэффективной.

Упражнение

2.10. Что произойдет, если системе Prolog будет задан следующий вопрос:

$$?- X = f(X).$$

Должен ли этот запрос на согласование завершиться успехом или неудачей? В соответствии с определением операции унификации в логике он должен окончиться неудачей, но что должно произойти в соответствии с определением операции согласования, приведенным в разделе 2.2? Объясните, почему во многих реализациях Prolog на данный вопрос дается такой ответ:

$$X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(\dots$$

Резюме

В первых двух главах рассматривалась основная часть языка Prolog, называемая также “чистым языком Prolog”. К этой части определения языка применяется термин “чистый”, поскольку она тесно связана с формальной логикой. А далее в этой книге (главы 3–7) рассматриваются расширения, которые направлены на то, чтобы можно было лучше приспособить этот язык к некоторым практическим потребностям. В данной главе рассматривались перечисленные ниже важные вопросы.

- *Простыми объектами* в языке Prolog являются атомы, переменные и числа. *Структурированные объекты* (или *структуры*) используются для представления объектов, имеющих несколько компонентов.
- Структуры создаются с помощью *функторов*. Каждый функтор определяется своим именем и арностью.
- *Тип объекта* распознается исключительно по его синтаксической форме.
- *Лексической областью определения* переменных является одно предложение. Поэтому под переменными с одним и тем же именем в двух предложениях подразумеваются две различные переменные.
- Структуры могут быть естественным образом представлены как *деревья*. Prolog может рассматриваться как язык для обработки древовидных информационных структур.
- При проведении *операции согласования* берутся два терма и предпринимается попытка сделать их идентичными, конкретизируя переменные в обоих термах.
- В случае успешного выполнения операции согласования ее результатом становится *наиболее общая конкретизация переменных*.
- *Декларативная семантика* языка Prolog определяет, является ли цель истинной по отношению к данной программе, а если установлено, что она истинна, то определяет, какая конкретизация переменных является истинной.
- Запятая между целями обозначает *конъюнкцию* целей, а точка с запятой между целями — *дизъюнкцию* целей.
- *Процедурная семантика* языка Prolog определяет процедуру достижения списка целей в контексте данной программы. Процедура выводит сведения об истинности или ложности целей в списке и соответствующие конкретизации переменных. Процедура автоматически выполняет возвраты для исследования альтернатив.
- *Декларативное значение* программ в “чистом языке Prolog” не зависит от порядка предложений и порядка целей в предложениях.
- *Процедурное значение* зависит от порядка целей и предложений. Поэтому такой порядок может повлиять на эффективность программы; неприемлемый порядок может даже привести к бесконечным рекурсивным вызовам.
- Изменение порядка предложений и целей в программе, правильной с декларативной точки зрения, позволяет повысить эффективность программы и сохранить при этом ее декларативную правильность. Такое *переупорядочение* является одним из методов предотвращения возможности возникновения бесконечных циклов.
- Кроме переупорядочения, существуют другие более общие методы *предотвращения бесконечных циклов* и, тем самым, обеспечения процедурной надежности программ.
- В данной главе рассматривались следующие понятия:
 - объекты данных: атом, число, переменная, структура;
 - терм;

- функтор, арность функтора;
- главный функтор терма;
- согласование термов;
- наиболее общая конкретизация;
- декларативная семантика;
- экземпляр предложения, вариант предложения;
- процедурная семантика;
- выполнение целей.

Списки, операции, арифметические выражения

В этой главе...

3.1. Представление списков	76
3.2. Некоторые операции со списками	78
3.3. Запись в операторной форме	87
3.4. Арифметические выражения	92

В данной главе рассматривается специальная система обозначений для списков, которые представляют собой одни из наиболее простых и полезных структур, а также приведены некоторые программы для выполнения специальных операций со списками. Кроме того, здесь рассматриваются простые арифметические выражения и связанные с ними средства записи в операторной форме, которые часто способствуют повышению удобства чтения программ. Основной синтаксис Prolog, описанный в главе 2, после введения этих трех дополнений становится удобной инфраструктурой для написания интересных программ.

3.1. Представление списков

Список — это простая структура данных, широко используемая в нечисловом программировании. *Список* представляет собой последовательность, состоящую из любого количества элементов, таких как `ann`, `tennis`, `tom`, `skiing`. Подобный список может быть записан в языке Prolog следующим образом:

```
[ ann, tennis, tom, skiing]
```

Но это, тем не менее, лишь внешнее представление списка. Как уже было показано в главе 2, все структурированные объекты в языке Prolog представляют собой деревья. Списки не являются исключением из этого правила.

При изучении возможных способов представления списка в виде стандартного объекта Prolog необходимо учитывать два случая: список может быть либо пустым, либо непустым. В первом случае список записывается просто в виде атома Prolog, `[]`. Во втором случае список можно рассматривать как состоящий из следующих компонентов.

1. Первый компонент, называемый *головой списка*.
2. Остальная часть списка, называемая *хвостом*.

В приведенном выше примере списка

```
[ ann, tennis, tom, skiing]
```

головой является `ann`, а хвостом — следующий список:


```
[ tennis, tom, skiing]
```

Как правило, в качестве головы может быть выбран любой объект Prolog (например, дерево или переменная), а хвост должен представлять собой список. Затем голова и хвост объединяются в некоторую структуру с помощью специального функтора списка:

```
.( Head, Tail)
```

Поскольку Tail, в свою очередь, является списком, он может либо быть пустым, либо иметь собственную голову и хвост. Поэтому для представления списков любой длины в программе Prolog не требуются какие-либо дополнительные понятия, кроме понятий функтора, атома и терма. В таком случае приведенный выше пример списка представляется в виде следующего терма:

```
.( ann, .( tennis, .( tom, .( skiing, [])) ) )
```

Соответствующая древовидная структура показана на рис. 3.1. Следует отметить, что в приведенном выше терме присутствует пустой список. Это связано с тем, что предпоследний хвост представляет собой одноэлементный список:

```
[ skiing]
```

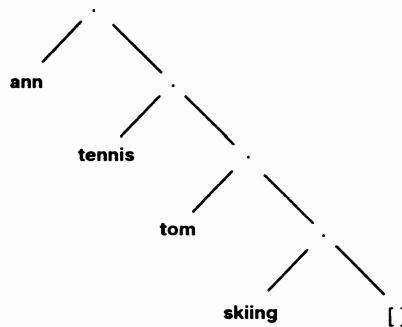


Рис. 3.1. Древовидное представление списка [ann, tennis, tom, skiing]

Этот список в качестве хвоста имеет пустой список:

```
[skiing] = .(skiing, [])
```

Данный пример свидетельствует о том, что общий принцип структуризации объектов данных в языке Prolog применим и для создания списков любой длины. Кроме того, как показывает этот пример, упрощенная система обозначений с помощью точек, которая может потребовать использования весьма глубокого вложения субтермов в части с обозначением хвоста, иногда приводит к созданию довольно громоздких и запутанных выражений. Именно по этой причине в языке Prolog предусмотрена более наглядная система обозначений для списков, с помощью которой списки можно записывать в виде последовательности элементов, заключенной в квадратные скобки. В программе могут использоваться обе системы обозначений, но обычно запись с применением квадратных скобок является более предпочтительной. Но следует учитывать, что это лишь соглашение, применяемое для упрощения внешнего представления, и что внутренним представлением списков являются бинарные деревья. При выводе на внешнее устройство подобные термы автоматически преобразуются в более наглядную форму. Поэтому возможны следующие диалоги с системой Prolog:

```
?- List1 = [a,b,c],
List2 = .( a, .( b, .( c, [])) ).
List1 = [a,b,c]
List2 = [a,b,c]
?- Hobbies1 = .(tennis, .( music, [])),
```

```
Hobbies2 = [ skiing, food],
L = [ ann, Hobbies1, tom, Hobbies2].
Hobbies1 = [ tennis, music]
Hobbies2 = [ skiing, food]
L = [ ann, [ tennis, music], tom, [ skiing, food] ]
```

Этот пример напоминает также, что элементы списка могут представлять собой объекты любого рода; в частности, они также могут быть списками.

На практике часто возникает необходимость рассматривать весь хвост списка как единый объект. Например, предположим, что определен такой список:

```
L = [a,b,c]
```

Поэтому можно записать следующее:

```
Tail = [b,c] и L = .( a, Tail)
```

Чтобы можно было представить это выражение на основе системы обозначений для списков с применением квадратных скобок, в языке Prolog предусмотрено еще одно дополнение к списковой записи: вертикальная черта, которая разделяет голову и хвост, как показано в следующем примере:

```
L = [a | Tail]
```

Обозначение с помощью вертикальной черты фактически является более общим, поскольку в список можно ввести любое количество элементов, за ними указать символ “|” и после этого привести список оставшихся элементов. Поэтому все следующие альтернативные способы записи приведенного выше списка являются допустимыми:

```
[a,b,c] = [a | [b,c] ] = [a,b | [c] ] = [a,b,c | []]
```

Из этого следуют приведенные ниже выводы.

- *Список* — это структура данных, которая может либо быть пустой, либо состоять из двух частей: *головы* и *хвоста*. Сам хвост также должен быть списком.
- Списки обрабатываются в языке Prolog как частный случай бинарных деревьев. Для повышения удобства чтения в языке Prolog для списков предусмотрена специальная система обозначений, поэтому являются допустимыми списки, оформленные следующим образом:

```
[ Item1, Item2, ...]
```

или

```
[ Head | Tail]
```

или

```
[ Item1, Item2, ... | Others]
```

3.2. Некоторые операции со списками

Списки могут использоваться для представления множеств, но с учетом различия между ними, поскольку порядок элементов в множестве не играет роли, а в списке он является значимым; кроме того, один и тот же объект может неоднократно повторяться в списке, но не в множестве. Тем не менее наиболее широко применяемые операции со списками являются аналогичными операциям с множествами. В частности, к ним относятся следующие:

- проверка того, является ли некоторый объект элементом списка, что соответствует проверке принадлежности к множеству;
- конкатенация двух списков, в результате чего формируется третий список; такую операцию можно считать соответствующей объединению множеств;
- добавление нового объекта в список или удаление некоторого объекта из него.

В оставшейся части этого раздела приведены программы для выполнения этих и некоторых других операций со списками.

3.2.1. Проверка принадлежности к списку

Разработаем отношение для проверки принадлежности к списку, которое имеет следующую общую форму:

```
member( X, L)
```

где X — объект, а L — список. Цель `member( X, L)` является истинной, если X присутствует в L . Например, цель

```
member( b, [a,b,c])
```

является истинной, а

```
member( b, [a,[b,c]])
```

не является истинной, но

```
member( [b,c], [a,[b,c]])
```

является истинной. Программа для проверки отношения принадлежности к списку может быть основана на приведенных ниже рассуждениях.

X входит в состав L , если истинно одно из следующих утверждений:

- 1) X является головой L или
- 2) X входит в состав хвоста L .

Эту программу можно записать в виде двух предложений; первое представляет собой простой факт, а второе — правило:

```
member( X, [X | Tail]).  
member( X, [Head | Tail]) :-  
    member( X, Tail).
```

3.2.2. Конкатенация

Для конкатенации списков определим следующее отношение:

```
conc( L1, L2, L3)
```

где $L1$ и $L2$ — два списка, а $L3$ — их конкатенация. Например:

```
conc( [a,b], [c,d], [a,b,c,d])
```

является истинным, но

```
conc( [a,b], [c,d], [a,b,a,c,d])
```

является ложным. При определении отношения `conc` снова необходимо рассмотреть два описанных ниже случая, в зависимости от первого параметра, $L1$.

1. Если первый параметр является пустым списком, то второй и третий параметры должны представлять собой одинаковые списки (назовем их L); это выражается следующим фактом Prolog:

```
conc([], L, L).
```

2. Если первый параметр `conc` — непустой список, то он имеет голову и хвост и должен выглядеть примерно так:

```
[X | L1]
```

На рис. 3.2 иллюстрируется операция конкатенации списка $[X | L1]$ и некоторого списка $L2$. Результатом конкатенации становится список $[X | L3]$, где $L3$ — конкатенация $L1$ и $L2$. В языке Prolog этот результат можно записать следующим образом:

```
conc( [X | L1], L2, [X | L3]) :-  
    conc( L1, L2, L3).
```

Теперь эта программа может использоваться для конкатенации заданных списков, например:

```
?- conc( [a,b,c], [1,2,3], L).  
L = [a,b,c,1,2,3]  
?- conc([a,[b,c],d], [a,[],b], L).  
L = [a, [b,c],d, a, [],b]
```

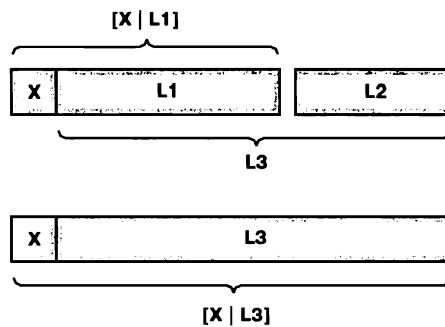


Рис. 3.2. Конкатенация списков

Хотя программа `conc` выглядит довольно простой, она допускает разностороннее использование с помощью многих других способов. Например, `conc` можно заставить работать в обратном направлении, для декомпозиции заданного списка на два отдельных списка, как показано ниже.

```
?- conc( L1, L2, [a,b,c] ).
L1 = []
L2 = [a,b,c];
L1 = [a]
L2 = [b,c];
L1 = [a,b]
L2 = [c];
L1 = [a,b,c]
L2 = [];
```

Возможны четыре варианта декомпозиции списка $[a, b, c]$, и все они были найдены программой `conc` с помощью перебора с возвратами.

Кроме того, эту программу можно использовать для поиска определенного элемента в списке. Например, с ее помощью можно найти все месяцы, которые предшествуют, и месяцы, которые следуют за указанным месяцем, как показывает приведенная ниже цель.

```
?- conc( Before, [may | After],
[jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec] ).
Before = [jan, feb, mar, apr]
After = [jun, jul, aug, sep, oct, nov, dec].
```

Кроме того, можно найти месяцы, которые непосредственно предшествуют и непосредственно следуют, допустим, за маем месяцем, введя следующий вопрос:

```
?- conc( _, [Month1, may, Month2 | _],
[jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec] ).
Month1 = apr
Month2 = jun
```

Более того, с ее помощью можно, например, удалить из некоторого списка $L1$ все, что следует за тремя последовательными вхождениями элементов z в списке $L1$, наряду с этими тремя z , например:

```
?- L1 = [a,b,z,z,z,c,z,z,z,d,e],
conc( L2, [z,z,z | _], L1 ).
L1 = [a,b,z,z,z,c,z,z,z,d,e]
L2 = [a,b,z,z,c]
```

Выше уже было запрограммировано отношение для проверки принадлежности к списку. Но такое же отношение можно более изящно запрограммировать с помощью программы `conc`, введя следующее предложение:

```
member1( X, L ) :-
conc( L1, [X | L2], L ).
```

Это предложение фактически говорит о следующем: X входит в состав списка L , если L можно разложить на два списка таким образом, что X является головой второго из них. Безусловно, `member1` определяет такое же отношение, что и `member`. Другое имя этому отношению было присвоено лишь для того, чтобы можно было отличить друг от друга эти две реализации. Обратите внимание на то, что приведенное выше предложение можно записать с использованием анонимных переменных следующим образом:

```
member1( X, L) :-
  conc( _, [X | _], L).
```

Любопытно сравнить между собой обе реализации отношения проверки принадлежности, `member` и `member1`. Первая из них имеет довольно простое процедурное значение, которое состоит в следующем:

- Для проверки того, входит ли некоторый элемент X в состав некоторого списка L :
- 1) вначале следует проверить, не равна ли элементу X голова списка L , а затем
 - 2) проверить, не входит ли элемент X в состав хвоста списка L .

С другой стороны, декларативное прочтение отношения `member1` является простым, но его процедурное значение не столь очевидно. Один из любопытных экспериментов состоит в том, чтобы определить, как фактически отношение `member1` применяется при вычислении ответа на некоторый вопрос. Определенное представление об этом можно получить на примере трассировки выполнения; рассмотрим следующий вопрос:

```
?- member1( b, [a,b,c]).
```

Соответствующая трассировка приведена на рис. 3.3. На основании этой трассировки можно сделать вывод, что отношение `member1` действует аналогично отношению `member`. Оно сканирует список элемент за элементом до тех пор, пока не будет найден искомый элемент или исчерпан список.

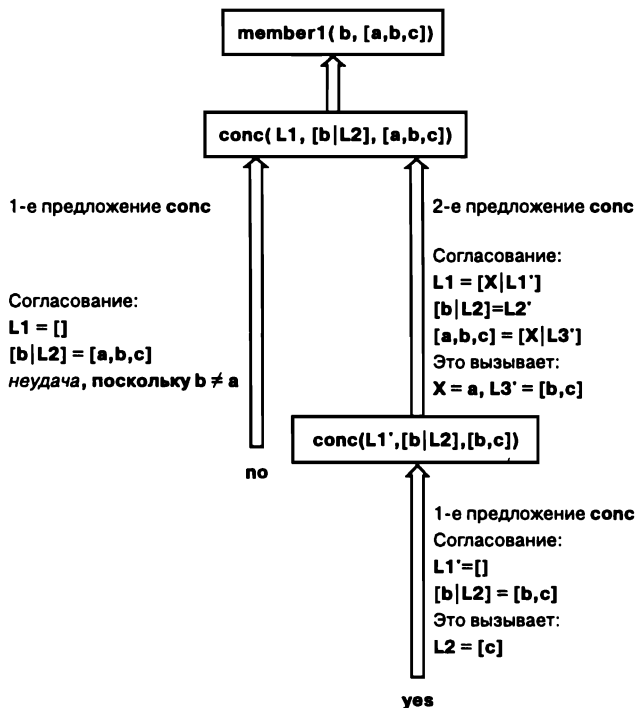


Рис. 3.3. Пример того, что процедура `member1` находит элемент в данном списке путем последовательного поиска в этом списке

Упражнения

3.1. С помощью программы `conc` выполните приведенные ниже задания.

- Составьте цель для удаления последних трех элементов из списка `L` и создания другого списка, `L1`. Подсказка: `L` представляет собой конкатенацию `L1` и трехэлементного списка.
- Составьте цель для удаления первых трех и последних трех элементов из списка `L` и создания списка `L2`.

3.2. Определите отношение

```
last( Item, List)
```

таким образом, чтобы `Item` был последним элементом списка `List`. Разработайте две версии: а) с использованием отношения `conc`; б) без использования отношения `conc`.

3.2.3. Добавление элемента

Чтобы добавить новый элемент в список, проще всего поместить этот элемент перед списком, чтобы он стал новой головой этого списка. Если `X` — новый элемент и `L` — список, к которому добавляется `X`, то результирующий список принимает вид

```
[X|L]
```

Поэтому для добавления нового элемента в начало списка фактически не требуется какая-либо специальная процедура. Тем не менее, если будет необходимо явно определить такую процедуру, ее можно записать как следующий факт:

```
add( X, L, [X | L] ).
```

3.2.4. Удаление элемента

Удаление элемента `X` из списка `L` можно запрограммировать как следующее отношение:

```
del( X, L, L1)
```

Здесь `L1` равно списку `L` с удаленным элементом `X`. Отношение `del` можно определить аналогично отношению проверки принадлежности. При этом необходимо снова рассмотреть приведенные ниже два случая.

- Если `X` является головой списка, результатом удаления становится хвост списка.
- Если `X` находится в хвосте, то удаление выполняется в хвосте.

```
del( X, [X | Tail], Tail ).
del( X, [Y | Tail], [Y | Tail1] ) :-
    del( X, Tail, Tail1 ).
```

Подобно отношению `member`, отношение `del` также является недетерминированным. Это означает, что если в списке имеется несколько вхождений элемента `X`, отношение `del` будет способно удалить любое из них с помощью перебора с возвратами, но последовательность удаления элементов при каждом вызове не определена. Безусловно, при каждом альтернативном выполнении будет удаляться только одно вхождение `X`, а другие останутся нетронутыми, например:

```
?- del( a, [a,b,a,a], L ).
L = [b,a,a];
L = [a,b,a];
L = [a,b,a];
no
```

Выполнение отношения `del` оканчивается неудачей, если список не содержит элемента, подлежащего удалению.

Кроме того, отношение `del` может использоваться для выполнения обратной операции — для добавления элемента к списку путем вставки нового элемента в любом месте списка. Например, если требуется вставить элемент `a` в любое место списка `[1,2,3]`, то эту операцию можно выполнить, задав вопрос о том, каковым является список `L`, такой, что после удаления из него элемента `a` будет получен список `[1,2,3]`, как показано ниже.

```
?- del( a, L, [1,2,3]).
L = [a,1,2,3];
L = [1,a,2,3];
L = [1,2,a,3];
L = [1,2,3,a];
no
```

В целом операцию вставки элемента `X` в любом месте некоторого списка `List` для получения списка `BiggerList` можно определить с помощью следующего предложения:

```
insert( X, List, BiggerList) :-
    del( X, BiggerList, List).
```

Отношение `member1` представляет собой пример изящной реализации отношения проверки принадлежности с использованием `conc`. Для проверки принадлежности может также применяться `del`. Идея здесь проста: некоторый элемент `X` входит в состав списка `List`, если он может быть удален из этого списка, как показано ниже.

```
member2( X, List) :-
    del( X, List, _).
```

3.2.5. Подсписок

Теперь рассмотрим отношение `sublist`. Это отношение имеет два параметра — список `L` и список `S`, такой, что он встречается в `L` в качестве его подсписка, поэтому предикат

```
sublist( [c,d,e], [a,b,c,d,e,f])
```

является истинным, а предикат

```
sublist( [c,e], [a,b,c,d,e,f])
```

является ложным. Программа Prolog для отношения `sublist` может быть основана на той же идее, что и `member1`, но на этот раз отношение является более общим (рис. 3.4). Соответствующим образом, это отношение может быть сформулировано, как показано ниже.

`S` является подсписком `L`, если выполняются следующие условия:

- 1) `L` может быть разложен на два списка, `L1` и `L2`, и
- 2) `L2` может быть разложен на два списка, `S` и некоторый список `L3`.

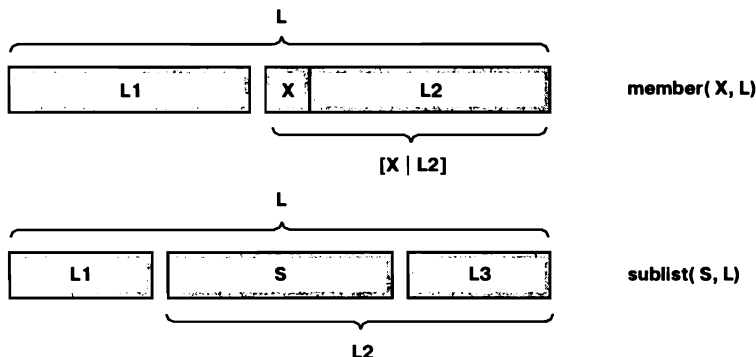


Рис. 3.4. Отношения `member` и `sublist`

Как было показано выше, для разбиения списков может использоваться отношение `conc`. Поэтому приведенную выше формулировку можно выразить на языке Prolog следующим образом:

```
sublist( S, L) :-
    conc( L1, L2, L),
    conc( S, L3, L2).
```

Безусловно, процедура `sublist` допускает разнообразное применение несколькими способами. Несмотря на то что она была разработана для проверки того, встречается ли некоторый список в качестве подсписка в другом списке, ее можно также использовать, например, для поиска всех подписков данного списка, например, следующим образом:

```
?- sublist( S, [a,b,c] ).
S = [];
S = [a];
S = [a,b];
S = [a,b,c];
S = [b];
...
```

3.2.6. Перестановки

Иногда возникает необходимость найти все возможные перестановки элементов в заданном списке. Для этого определим отношение `permutation` с двумя параметрами. Параметры представляют собой два списка, таких, что один из них является перестановкой другого. Это отношение предназначено для выработки перестановок списка по методу перебора с возвратами с помощью процедуры `permutation`, как в следующем примере:

```
?- permutation( [a,b,c], P).
P = [a,b,c];
P = [a,c,b];
P = [b,a,c];
...
```

Программа для отношения `permutation` может быть снова сформирована исходя из анализа двух описанных ниже случаев, в зависимости от первого списка.

1. Если первый список пуст, второй список также должен быть пустым.
2. Если первый список не пуст, то он имеет форму $[X \mid L]$ и перестановку такого списка можно сформировать, как показано на рис. 3.5, — вначале выполнить перестановку L , получив $L1$, а затем вставить X в любую позицию списка $L1$.

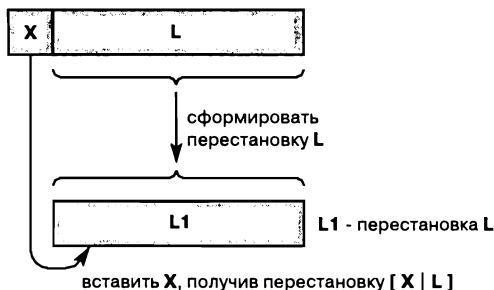


Рис. 3.5. Один из способов формирования перестановки списка $[X \mid L]$

Ниже приведены два предложения Prolog, которые соответствуют этим двум случаям.


```
permutation( [], []).
permutation( [X | L], P) :-
    permutation( L, L1),
    insert( X, L1, P).
```

Один из альтернативных подходов к разработке этой программы может предусматривать удаление одного из элементов (X) из первого списка, перестановку оставшегося списка с получением списка P, а затем добавление X перед списком P. Соответствующая программа выглядит следующим образом:

```
permutation2( [], []).
permutation2( L, [X | P]) :-
    del( X, L, L1),
    permutation2( L1, P).
```

Целесообразно провести некоторые эксперименты с этими программами перестановки. Обычный способ вызова их на выполнение выглядит примерно так:

```
?- permutation( [ red, blue, green], P).
```

В соответствии с поставленной задачей должны быть получены все шесть перестановок следующим образом:

```
P = [ red, blue, green];
P = [ red, green, blue];
P = [ blue, red, green];
P = [ blue, green, red];
P = [ green, red, blue];
P = [ green, blue, red];
no
```

Еще одна попытка использовать отношение `permutation` может состоять в следующем:

```
?- permutation( L, [a,b,c]).
```

Теперь первая версия этого отношения, `permutation`, успешно конкретизирует переменную `L` всеми шестью значениями перестановок. А если после этого пользователь затребует дополнительные решения, программа так и не ответит “no”, поскольку войдет в бесконечный цикл, пытаясь найти еще одну перестановку, притом что таковая отсутствует. Вторая версия, `permutation2`, в этом случае найдет только первую (идентичную) перестановку, а затем сразу же войдет в бесконечный цикл. Поэтому при использовании этих программ получения перестановок необходимо соблюдать осторожность.

Упражнения

3.3. Определите два предиката

`evenlength( List)` и `oddlength( List)`

такие, что они становятся истинными, если их параметром является список, который, соответственно, содержит четное или нечетное количество элементов (имеет четную или нечетную длину). Например, список `[ a, b, c, d]` соответствует предикату `evenlength`, тогда как `[ a, b, c]` — предикату `oddlength`.

3.4. Определите отношение

`reverse( List, ReversedList)`

которое изменяет порядок следования элементов в списках на противоположный, например `reverse( [ a, b, c, d], [ d, c, b, a])`.

3.5. Определите предикат `palindrome ( List)`. Список является *палиндромом*, если он читается одинаково в прямом и обратном направлении, например `[m,a,d,a,m]`.

3.6. Определите отношение

```
shift( List1, List2)
```

такое, что List2 представляет собой List1 после “кругового сдвига” на один элемент влево. Например, вопрос

```
?- shift( [1,2,3,4,5], L1),  
shift( L1, L2).
```

приводит к получению следующего ответа:

```
L1 = [2,3,4,5,1]  
L2 = [3,4,5,1,2]
```

3.7. Определите отношение

```
translate( List1, List2)
```

для преобразования списка цифр от 0 до 9 в список соответствующих слов, например, следующим образом:

```
translate( [3,5,1,3], [three,five,one,three])
```

Используйте следующие факты в качестве вспомогательного отношения:

```
means( 0, zero).  
means( 1, one).  
means( 2, two).  
...
```

3.8. Определите отношение

```
subset( Set, Subset)
```

где Set и Subset — два списка, представляющие два множества. Необходимо предусмотреть, чтобы это отношение можно было использовать не только для проверки принадлежности подмножества к множеству, но и для получения всех возможных подмножеств данного множества, например, следующим образом:

```
?- subset( [a,b,c], S).  
S = [a,b,c];  
S = [a,b];  
S = [a,c];  
S = [a];  
S = [b,c];  
S = [b];  
...
```

3.9. Определите отношение

```
dividelist( List, List1, List2)
```

таким образом, чтобы элементы списка List распределялись между списками List1 и List2, причем List1 и List2 имели примерно одинаковую длину, например `dividelist ([ a, b, c, d, e], [ a, c, e], [ b, d])`.

3.10. Переопределите программу с обезьяной и бананом, приведенную в главе 2, как отношение

```
canget( State, Actions)
```

чтобы она не просто отвечала “yes” или “no”, но и вырабатывала последовательность действий обезьяны, представленную как список шагов, например:

```
Actions = [ walk(door>window), push(window,middle), climb, grasp]
```

3.11. Определите отношение

```
flatten( List, FlatList)
```

где List может представлять собой список списков, а FlatList является списком List, “линеаризованным” таким образом, чтобы элементы подсписков (или подподсписков) List были реорганизованы в виде одного линейного списка, например:

```
?- flatten( [a,b,[c,d],[],[[e]]],f), L).  
L = [a,b,c,d,e,f]
```

3.3. Запись в операторной форме

В математике широко используются выражения, аналогичные приведенному ниже.

$2 * a + b * c$

где $+$ и $*$ представляют собой знаки операций сложения и умножения, а 2 , a , b — операнды. В частности, операции $+$ и $*$ называются инфиксными, поскольку знаки этих операций находятся между двумя операндами. Подобные выражения могут быть представлены в виде деревьев, как показано на рис. 3.6, а также записаны с помощью термов языка Prolog с использованием знаков $+$ и $*$ в качестве функторов:

$+(*(2,a), *(b,c))$

Поскольку обычно удобнее записывать такие выражения в привычной, инфиксной форме с использованием знаков операций, в языке Prolog предусмотрена возможность задавать выражение в операторной записи, поэтому в программе Prolog допускается вводить подобные выражения в следующем виде:

$2*a + b*c$

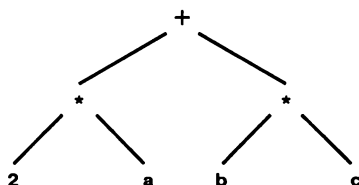


Рис. 3.6. Древовидное представление выражения $2*a + b*c$

Но следует учитывать, что это только внешнее представление соответствующего объекта, которое автоматически преобразуется системой в обычную форму термов Prolog, а при выводе подобный терм будет снова представлен для пользователя в его внешней, инфиксной форме.

Таким образом, применение знаков операции в языке Prolog представляет собой альтернативный способ записи выражений. Если в программе встретится конструкция $a + b$, система Prolog обработает ее точно так же, как если бы она была записана в форме $+(a, b)$. Но для того чтобы система Prolog могла правильно трактовать такие выражения, как $a + b*c$, она должна иметь информацию о том, что знак $*$ связывает сильнее, чем $+$. Формально это выражается в том, что $+$ имеет более высокий приоритет, чем $*$, поэтому правильная интерпретация выражений зависит от приоритета знаков операций. Например, выражение $a + b*c$ можно было бы в принципе трактовать либо как

$+(a, *(b,c))$

либо как

$*(+(a,b), c)$

Общее правило состоит в том, что в выражении без скобок знак операции с наивысшим приоритетом является главным функтором терма. Если выражения, содержащие знаки операции $+$ и $*$, должны обрабатываться в системе в соответствии с общепринятыми соглашениями, то знак $+$ должен иметь более высокий приоритет, чем знак $*$. В этом случае выражение $a + b*c$ означает то же, что и $a + (b*c)$. Если же требуется другая интерпретация, то ее необходимо явно обозначить с помощью круглых скобок, например, как $(a + b)*c$.

Программист может определять свои собственные знаки операции. Поэтому, например, в программе можно определить атомы `has` и `supports` как инфиксные знаки операции, а затем записывать в программе факты наподобие следующих:

```
peter has information.  
floor supports table.
```

Эти факты полностью эквивалентны следующим:

```
has( peter, information).  
supports( floor, table).
```

Программист может определять новые операции, вставляя в программу предложения специального типа, иногда называемые *директивами* (directive), которые действуют как определения операций. Определение операции должно находиться в программе перед любым выражением, содержащим эту операцию. В данном примере знак операции `has` должен быть правильно определен с помощью следующей директивы:

```
:- op( 600, xfx, has).
```

Такая конструкция сообщает системе Prolog, что предусмотрено использование атома `has` в качестве знака операции, которая имеет приоритет 600 и относится к типу `“xfx”`; этот тип характеризует некоторые инфиксные операции. Такая форма спецификатора `“xfx”` указывает, что данный знак операции, обозначенный как `“f”`, должен находиться между двумя операндами, обозначенными как `“x”`.

Следует отметить, что определения операций не задают каких-либо манипуляций или действий. В принципе, со знаком операции не связаны никакие операции над данными (за исключением очень редких случаев). Знаки операции, как и функторы, обычно используются только для объединения объектов в структуры, а не для активизации действий над данными, хотя на первый взгляд кажется, что слово *“операция”* предполагает выполнение какого-либо действия.

Знаки операции, называемые также *операторами*, представляют собой атомы. Приоритет операции не должен выходить из определенного диапазона, который зависит от реализации. В дальнейшем предполагается, что этот диапазон находится в пределах 1–1200.

Типы операций подразделяются на три группы, которые обозначаются спецификаторами типа, такими как `xfx`. Эти три группы перечислены ниже.

1. Инфиксные операции трех типов:

```
xfx xfy yfx
```

2. Префиксные операции двух типов:

```
fx fy
```

3. Постфиксные операции двух типов:

```
xf yf
```

Спецификаторы выбираются таким образом, чтобы они отражали структуру выражения, где `“f”` представляет знак операции, а `“x”` и `“y”` — операнды. Присутствие обозначения `“f”` между операндами указывает на то, что операция является инфиксной. Спецификаторы префиксных и постфиксных операций имеют только один операнд, который, соответственно, следует или предшествует знаку операции.

Между обозначениями `“x”` и `“y”` имеется определенное различие. Для описания этого различия необходимо ввести понятие *приоритета операнда*. Если операнд заключен в круглые скобки или представляет собой неструктурированный объект, то его приоритет равен 0; если операнд представляет собой структуру, то его приоритет равен приоритету его главного функтора. Знак `“x”` обозначает операнд, приоритет которого должен быть строго меньше, чем у операции, а знак `“y”` представляет операнд, приоритет которого меньше или равен приоритету операции.

Эти правила позволяют устранять неоднозначность при анализе выражений, состоящих из нескольких операций с одинаковым приоритетом. Например, выражение

```
a - b - c
```

обычно интерпретируется как $(a - b) - c$, а не как $a - (b - c)$. Для обеспечения такой правильной интерпретации тип операции “-” должен быть определен как `yfx`. На рис. 3.7 показано, почему вторая интерпретация при этом будет исключена.

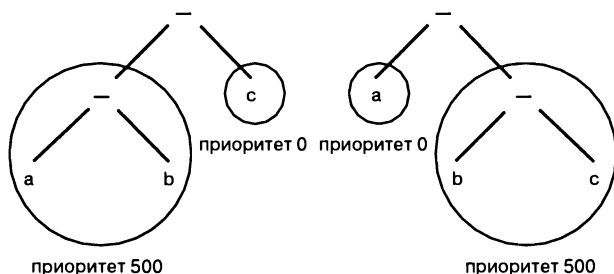


Рис. 3.7. Две интерпретации выражения $a - b - c$, в которых предполагается, что операция “-” имеет приоритет 500. Если операция “-” относится к типу `yfx`, то вторая интерпретация является недействительной, поскольку приоритет выражения $b - c$ не меньше приоритета операции “-”

В качестве еще одного примера рассмотрим префиксную операцию `not`. Если операция `not` определена как `fy`, то выражение

`not not p`

является допустимым, но если операция `not` определена как `fx`, то это выражение недопустимо, поскольку операндом первой операции `not` является выражение `not p`, имеющее такой же приоритет, как и сама операция `not`. В последнем случае данное выражение должно записываться с круглыми скобками, следующим образом:

`not( not p)`

Для удобства некоторые операции определены в системе Prolog заранее, таким образом, чтобы их можно было сразу же использовать, поэтому для них не требуется вводить определение. Каковы эти операции и какими являются их приоритеты, зависит от реализации Prolog. Мы предполагаем, что этот набор “стандартных” операций соответствует определениям, заданным с помощью предложений, которые приведены в листинге 3.1. Операции в этом листинге представляют собой подмножество операций, которые определены в стандарте Prolog, с добавлением операции `not`. Кроме того, как показано в листинге 3.1, в одном предложении может быть объявлено несколько операций, если все они имеют одинаковый приоритет и относятся к одному и тому же типу. В таком случае знаки операций записываются в виде списка.

Использование операций позволяет намного повысить удобство программ для чтения. В качестве примера предположим, что разрабатывается программа для обработки логических выражений. В такой программе может потребоваться, например, ввести одну из теорем эквивалентности де Моргана (de Morgan), которая с помощью математических обозначений может быть записана следующим образом:

$\sim(A \ \& \ B) \iff \sim A \vee \sim B$

Один из способов формулировки этого утверждения в языке Prolog может предусматривать использование следующего предложения:

`equivalence( not( and( A, B)), or( not( A), not( B) ) ).`

Листинг 3.1. Множество заранее определенных операций

```
:- op( 1200, xfx, [:-,-->] ).
:- op( 1200, fx, [:-,?-] ).
:- op( 1100, xfy, ';' ).
```

```

:- op( 1050, xfy, ->).
:- op( 1000, xfy, ',').
:- op( 900, fy, [ not, '\+' ] ).
:- op( 700, xfx, [ =, \=, ==, \==, =.. ] ).
:- op( 700, xfx, [ is, ==, =\=, <, =<, >, >=, @<, @=<, @>, @>= ] ).
:- op( 500, yfx, [ +, - ] ).
:- op( 400, yfx, [ *, /, //, mod ] ).
:- op( 200, xfx, ** ).
:- op( 200, xfy, ^ ).
:- op( 200, fy, ~ ).

```

Но обычно в программировании следует стремиться к тому, чтобы сохранялось максимально возможное сходство между первоначальной системой обозначений, применяемой в проблемной области, и системой обозначений, используемой в программе. В данном примере эту задачу можно выполнить почти полностью с использованием операций. Подходящий набор операций для данного назначения можно определить следующим образом:

```

:- op( 800, xfx, <==> ).
:- op( 700, xfy, v ).
:- op( 600, xfy, & ).
:- op( 500, fy, ~ ).

```

Теперь теореме де Моргана можно записать как следующий факт:

```
~(A&B) <==> ~A v ~B.
```

В соответствии с приведенной выше спецификацией операций этот терм будет интерпретироваться, как показано на рис. 3.8.

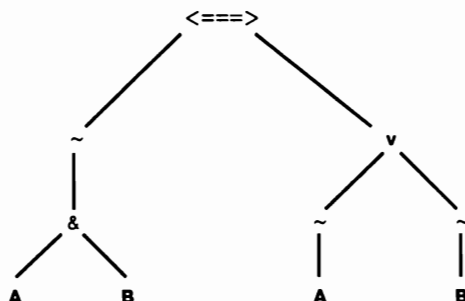


Рис. 3.8. Интерпретация терма $\sim(A \& B)$
 $\<==> \sim A \vee \sim B$

На основании изложенного можно сделать приведенные ниже заключения.

- Удобство программ для чтения часто можно повысить с помощью записи в операторной форме. Операции могут быть инфиксными, префиксными или постфиксными.
- В принципе, со знаками операций не связаны какие-либо операции над данными, если не считать некоторых частных случаев. Определения операций не определяют конкретные действия; они лишь вводят новые обозначения. Знаки операций, как и функторы, применяются только для соединения компонентов структур.
- Программист может определять свои собственные операции. В определении каждой операции необходимо указать ее знак, приоритет и тип.
- Приоритет — это целое число, не выходящее за пределы определенного диапазона, обычно 1–1200. Знак операции с наивысшим приоритетом в выражении

является главным функтором выражения. Операции с более низким приоритетом связывают операнды сильнее, чем операции с более высоким приоритетом.

- Тип операции зависит, во-первых, от положения знака операции по отношению к ее операндам и, во-вторых, от приоритета операндов в сравнении с приоритетом самой операции. В спецификаторе типа `xfy` знак `x` обозначает операнд, приоритет которого строго меньше по сравнению с операцией, а `y` обозначает операнд, приоритет которого меньше или равен приоритету этой операции.

Упражнения

3.12. Предположим, что заданы следующие определения операций:

```
:- op( 300, xfx, plays).  
:- op( 200, xfy, and).
```

В этом случае следующие два термина являются синтаксически правильными объектами:

```
Term1 = jimmy plays football and squash  
Term2 = susan plays tennis and basketball and volleyball
```

Как эти термины интерпретируются системой Prolog? Каковы их главные функторы и какова их структура?

3.13. Предложите подходящие определения операций (“was”, “of”, “the”), чтобы иметь возможность записывать примерно такие предложения:

```
diana was the secretary of the department.
```

а затем задавать системе Prolog следующие вопросы:

```
?- Who was the secretary of the department.
```

```
Who = diana
```

```
?- diana was What.
```

```
What = the secretary of the department
```

3.14. Рассмотрим следующую программу:

```
t( 0+1, 1+0).  
t( X+0+1, X+1+0).  
t( X+1+1, Z) :-  
t( X+1, X1),  
t( X+1, Z).
```

Как эта программа ответит на следующие вопросы, если “+” — это инфиксная операция типа `yfx` (как обычно):

а) ?- `t(0+1, A)`.

б) ?- `t( 0+1+1, B)`.

в) ?- `t( 1+0+1+1+1, C)`.

г) ?- `t( D, 1+1+1+0)`.

3.15. В предыдущем разделе отношения, касающиеся списков, записывались следующим образом:

```
member( Element, List),  
conc( List1, List2, List3),  
del( Element, List, NewList), ...
```

Предположим, что необходимо предусмотреть возможность записывать эти отношения таким образом:

```
Element in List,  
concatenating List1 and List2 gives List3,  
deleting Element from List gives NewList, ...
```

Определите операции “in”, “concatenating”, “and” и т.д., чтобы иметь такую возможность. Кроме того, переопределите соответствующие процедуры.

3.4. Арифметические выражения

Некоторые из заранее определенных операций могут использоваться в качестве основных арифметических операций. К ним относятся перечисленные ниже.

- +. Сложение
- -. Вычитание.
- *. Умножение.
- /. Деление.
- **. Возведение в степень.
- //. Целочисленное деление.
- mod. Деление по модулю, вычисление остатка от целочисленного деления.

Обратите внимание на то, что это и есть тот исключительный случай, в котором оператор может фактически вызвать на выполнение операцию. Но даже в таких случаях для осуществления арифметических действий требуется дополнительное указание. Например, следующий вопрос представляет собой наивную попытку потребовать выполнения арифметического вычисления:

```
?- X = 1 + 2.
```

Система Prolog “послушно” ответит

```
X = 1 + 2
```

а не $X = 3$, как следовало ожидать. Причина этого проста — выражение $1 + 2$ просто обозначает терм Prolog, в котором знак + является функтором, а 1 и 2 — его параметрами. В приведенной выше цели нет ничего, что могло бы вынудить систему Prolog фактически активизировать операцию сложения. Для преодоления этой проблемы предусмотрена специальная предопределенная операция is. Операция is вынуждает систему выполнить вычисление, поэтому правильный способ вызова арифметической операции состоит в следующем:

```
?- X is 1 + 2.
```

В этом случае будет получен ответ:

```
X = 3
```

При этом операция сложения была выполнена с помощью специальной процедуры, которая связана с операцией is. Подобные процедуры принято называть *встроенными процедурами*.

В различных реализациях Prolog могут применяться немного разные обозначения для арифметических операций. Например, знак операции “/” может обозначать целочисленное деление или деление действительных чисел. В данной книге знак “/” обозначает деление действительных чисел, знак “//” — целочисленное деление, а “mod” — вычисление остатка от деления. В соответствии с этим на вопрос

```
?- X is 5/2,  
Y is 5//2,  
Z is 5 mod 2.
```

будет получен следующий ответ:

```
X = 2.5  
Y = 2  
Z = 1
```

Левым операндом операции is служит простой объект, а правым операндом — арифметическое выражение, состоящее из арифметических операций, чисел и переменных. Поскольку операция is вынуждает систему выполнить вычисление, то все переменные в выражении ко времени выполнения данной цели уже должны быть конкретизированы числовыми значениями. Приоритет заранее определенных арифметических операций (см. листинг 3.1) является таковым, что ассоциативность па-

раметров с операциями соответствует обычным правилам математики. Для обозначения других ассоциаций могут применяться круглые скобки. Обратите внимание, что операции $+$, $-$, $*$, $/$ и div определены как yfx , а это означает, что их вычисление выполняется слева направо; например, терм

```
x is 5 - 2 - 1
```

интерпретируется следующим образом:

```
x is (5 - 2) - 1
```

Кроме того, в реализациях Prolog обычно предусмотрены такие стандартные функции, как $\text{sin}(X)$, $\text{cos}(X)$, $\text{atan}(X)$, $\text{log}(X)$, $\text{exp}(X)$ и т.д. Эти функции могут находиться справа от знака операции is .

Кроме того, при выполнении арифметических операций возникает необходимость сравнивать числовые значения. Например, с помощью следующей цели может быть выполнена проверка того, больше ли произведение чисел 277 и 37 по сравнению с числом 10000:

```
?- 277 * 37 > 10000.  
yes
```

Обратите внимание на то, что операция $>$, как и операция is , вынуждает систему вычислять арифметические выражения.

Предположим, что в программе имеется отношение born , которое устанавливает связь между именами людей и годами их рождения. В таком случае можно выполнить выборку имен людей, родившихся в период между 1980 и 1990 годами включительно, с помощью следующего вопроса:

```
?- born( Name, Year),  
   Year >= 1980,  
   Year <= 1990.
```

Ниже перечислены операции сравнения.

- $X > Y$. X больше Y .
- $X < Y$. X меньше Y .
- $X \geq Y$. X больше или равен Y .
- $X \leq Y$. X меньше или равен Y .
- $X =:= Y$. Значения X и Y равны.
- $X \neq Y$. Значения X и Y не равны.

Следует отметить, что операция сопоставления $=$ и операция проверки на равенство $=:=$ отличаются друг от друга; например, они по-разному ведут себя в целях $X = Y$ и $X =:= Y$. Первая цель вызывает согласование объектов X и Y , а если объекты X и Y согласуются, может приводить к конкретизации некоторых переменных в терминах X и Y . В этом случае вычисление не выполняется. С другой стороны, операция $X =:= Y$ вызывает вычисление арифметических выражений, но не может стать причиной какой-либо конкретизации переменных. Это отличие можно проиллюстрировать на следующих примерах:

```
?- 1 + 2 =:= 2 + 1.  
yes  
?- 1 + 2 = 2 + 1.  
no  
?- 1 + A = B + 2.  
A = 2  
B = 1
```

Рассмотрим более подробно использование арифметических операций на двух простых примерах. В первом из них решается задача вычисления наибольшего общего делителя, а во втором — подсчета элементов в списке.

Если даны два положительных целых числа, X и Y , то их наибольший общий делитель, D , можно определить с учетом перечисленных ниже трех случаев.

1. Если X и Y равны, то D равен X .
2. Если $X < Y$, то D равен наибольшему общему делителю X и разности $Y - X$.
3. Если $Y < X$, то для нахождения наибольшего общего делителя применяется такое же правило, как и в случае 2), после перестановки местами X и Y .

Можно легко показать на примере, что эти три правила действительно позволяют решить поставленную задачу. Например, если заданы значения $X = 20$ и $Y = 25$, то выполнение приведенных выше правил приводит к получению значения $D = 5$ после ряда вычитаний.

Эти правила можно сформулировать в виде программы Prolog, определив отношение с тремя параметрами, например, следующим образом:

```
gcd( X, Y, D)
```

После этого три правила можно представить в виде трех предложений следующим образом:

```
gcd( X, X, X).
gcd( X, Y, D) :-
    X < Y,
    Y1 is Y - X,
    gcd( X, Y1, D).
gcd( X, Y, D) :-
    Y < X,
    gcd( Y, X, D).
```

Безусловно, последнюю цель в третьем предложении можно равнозначно заменить такими двумя целями:

```
X1 is X - Y,
gcd( X1, Y, D)
```

Во втором примере требуется выполнить подсчет, а для этого обычно необходимо предусмотреть некоторые арифметические операции. Примером такой задачи является определение длины списка; иными словами, подсчет количества элементов в списке. Определим такую процедуру

```
length( List, N)
```

которая подсчитывает количество элементов в списке `List` и конкретизирует `N` значением полученного числа. Как и в случае предыдущих отношений, касающихся списков, целесообразно рассмотреть два описанных ниже случая.

1. Если список пуст, то его длина равна 0.
2. Если список не пуст, то `List = [Head | Tail]`; в таком случае его длина равна 1 плюс длина хвоста `Tail`.

Эти два случая соответствуют следующей программе:

```
length([], 0).
length( [_ | Tail], N) :-
    length( Tail, N1),
    N is 1 + N1.
```

В качестве примера применения программы `length` рассмотрим следующий вопрос:

```
?- length( [a,b,[c,d],e], N).
N = 4
```

Обратите внимание на то, что во втором предложении программы `length` две цели, применяемые в теле, нельзя менять местами. Причина этого состоит в том, что переменная `N1` должна быть конкретизирована для того, чтобы появилась возможность обработать цель:

```
N is 1 + N1
```

Вместе со встроенной процедурой `is` в программу было введено отношение, возможность выполнения которого зависит от порядка обработки, и поэтому процедурные соображения выступили на первый план.

Любопытно понаблюдать за тем, что произойдет при попытке создать программу для отношения `length` без использования операции `is`. Пример такой попытки показан ниже.

```
length1([],0).
length1( [_ | Tail], N) :-
    length1(Tail, N1),
    N = 1 + N1.
```

Теперь после ввода цели

```
?- length1([a,b,[c,d],e], N).
```

будет получен следующий ответ:

```
N = 1+(1+(1+(1+0))).
```

Системе не была передана на выполнение операция, которая явно вынуждает произвести операцию сложения, и поэтому такая операция так и не была осуществлена. Но, в отличие от `length`, вариант `length1` допускает перестановку целей во втором предложении, следующим образом:

```
length1( [_ | Tail], N) :-
    N = 1 + N1,
    length1(Tail, N1).
```

Эта версия `length1` вырабатывает такой же результат, как и первоначальная версия. Кроме того, для нее можно использовать более короткую запись следующим образом:

```
length1( [_ | Tail], 1 + N) :-
    length1(Tail, N).
```

которая также обеспечивает получение аналогичного результата. Но программу `length1` все равно можно использовать для определения количества элементов в списке, как показано ниже.

```
?- length1([a,b,c], N), Length is N.
N = 1+(1+(1+0))
Length = 3
```

Наконец, следует отметить, что предикат `length` часто реализуется в качестве встроенного предиката. На основании изложенного можно сделать выводы, которые приведены ниже.

- Для выполнения арифметических операций можно использовать встроенные процедуры.
- Выполнение арифметических операций необходимо явно затребовать с помощью встроенной процедуры `is`. С такими предопределенными операциями, как `+`, `-`, `*`, `/`, `div` и `mod`, также связаны встроенные процедуры.
- Ко времени вычисления арифметического выражения все его параметры должны быть уже конкретизированы числовыми значениями.
- Значения арифметических выражений можно сравнивать с помощью таких операций, как `<`, `=` и т.д. Эти операции вынуждают систему выполнять вычисления их параметров.

Упражнения

3.16. Определите отношение

```
max( X, Y, Max)
```

такое, что `Max` является наибольшим из двух чисел, `X` и `Y`.

3.17. Определите предикат

```
maxlist( List, Max)
```

такой, что `Max` является максимальным числом в списке чисел `List`.

3.18. Определите предикат

```
sumlist( List, Sum)
```

такой, что Sum — сумма заданного списка чисел List.

3.19. Определите предикат

```
ordered( List)
```

который является истинным, если List — упорядоченный список чисел, например `ordered( [1,5,6,6,9,12])`.

3.20. Определите предикат

```
subsum( Set, Sum, SubSet)
```

такой, что Set является списком чисел, SubSet — подмножеством этих чисел, а сумма чисел в SubSet равна Sum, например:

```
?- subsum( [1,2,5,3,2], 5, Sub).  
Sub = [1,2,2];  
Sub = [2,3];  
Sub = [5];  
...
```

3.21. Определите процедуру

```
between( N1, N2, X)
```

которая для двух заданных целых чисел N1 и N2 с помощью перебора с возвратами определяет все целые числа X, соответствующие ограничению $N1 \leq X \leq N2$.

3.22. Определите операторы “if”, “then”, “else” и “:=” таким образом, чтобы следующий терм стал действительным:

```
if X > Y then Z := X else Z := Y
```

Выберите приоритеты таким образом, чтобы “if” был главным функтором. Затем определите отношение “if”, чтобы оно выполняло функции своего рода небольшого интерпретатора для операторов “if-then-else” в следующей форме:

```
if Val1 > Val2 then Var := Val3 else Var := Val4
```

где Val1, Val2, Val3 и Val4 являются числами (или переменными, конкретизированными значениями чисел), а Var представляет собой переменную. Отношение “if” должно иметь следующую интерпретацию: если значение Val1 больше, чем значение Val2, то Var конкретизируется значением Val3, в противном случае — значением Val4. Ниже приведен пример использования такой интерпретации.

```
?- X = 2, Y = 3,  
   Val2 is 2*X,  
   Val4 is 4*X,  
   if Y > Val2 then Z := Y else Z := Val4,  
   if Z > 5 then W := 1 else W := 0.  
X = 2  
Y = 3  
Z = 8  
W = 1  
Val2 = 4  
Val4 = 8
```

Резюме

- *Список* — это широко используемая структура. Он может либо быть пустым, либо состоять из головы и хвоста, который также является списком. В языке Prolog для списков предусмотрен специальный формат записи.
- К числу обычных *операций со списками*, программы для выполнения которых рассматриваются в данной главе, относятся следующие: проверка принадлеж-

ности к списку, конкатенация, добавление элемента, удаление элемента, выделение подсписка.

- *Запись в операторной форме* позволяет программисту приспосабливать синтаксис программ к конкретным потребностям. Применение операций предоставляет возможность намного повысить удобство программ для чтения.
- Новые операции можно определить с помощью директивы *ор*, указав знак операции, ее тип и приоритет.
- В принципе, с операциями не связаны какие-либо действия; *знаки операций* представляют собой синтаксические обозначения, позволяющие применять альтернативный синтаксис для термов.
- *Арифметические действия* выполняются с помощью *встроенных процедур*. Система принуждается к выполнению арифметического выражения с помощью процедуры *is* и операций (предикатов) сравнения: $<$, $=$ и т.д.
- В этой главе введены следующие понятия:
 - список, голова списка, хвост списка;
 - списочное обозначение;
 - операторы, запись в операторной форме;
 - инфиксные, префиксные и постфиксные операторы;
 - приоритет оператора;
 - арифметические встроенные процедуры.

Использование структур: примеры программ

В этой главе...

4.1. Выборка структурированной информации из базы данных	98
4.2. Методы абстрагирования данных	102
4.3. Моделирование недетерминированного конечного автомата	103
4.4. Консультант бюро путешествий	107
4.5. Задача с восемью ферзями	111

Структуры данных, наряду со средствами согласования и перебора с возвратами, а также арифметическими выражениями, являются мощными инструментами программирования. Настоящая глава посвящена формированию навыков использования этих инструментальных средств с помощью примеров программ, которые обеспечивают выборку структурированной информации из базы данных, моделируют недетерминированный конечный автомат, применяются для планирования путешествий и позволяют решить задачу расстановки восьми ферзей на шахматной доске. В данной главе также показано, как принцип абстрагирования данных может быть реализован в программе Prolog. Примеры программ, приведенные в этой главе, можно изучать выборочно.

4.1. Выборка структурированной информации из базы данных

В данном примере демонстрируются методы представления структурированных объектов данных и манипулирования ими. В нем также показано, что язык Prolog хорошо подходит для оформления запросов к базе данных.

Любую базу данных можно естественным образом представить на языке Prolog в виде множества фактов. Например, базу данных о семьях можно представить таким образом, что для описания каждой семьи будет применяться одно предложение. На рис. 4.1 показано, каким способом может быть структурирована информация о каждой семье. Каждая семья имеет три составляющих: муж, жена и дети. Поскольку в разных семьях имеется различное количество детей, информация о детях представлена в виде списка, который позволяет включать в него любое количество элементов. Информация о каждом человеке, в свою очередь, представлена в виде структуры, состоящей из четырех компонентов: имя, фамилия, дата рождения, место работы. Информация о месте работы может содержать слово “unemployed” (безработный) или указывать организацию, в которой работает данное лицо, и заработную плату. Ин-

формацию о семье, представленную на рис. 4.1, можно записать в базу данных с помощью следующего предложения:

```
family(
person( tom, fox, date(7,may,1960), works(bbc, 15200)),
person( ann, fox, date(9,may,1961), unemployed),
[ person( pat, fox, date(5,may,1983), unemployed),
person( jim, fox, date(5,may, 1983), unemployed) ]).
```

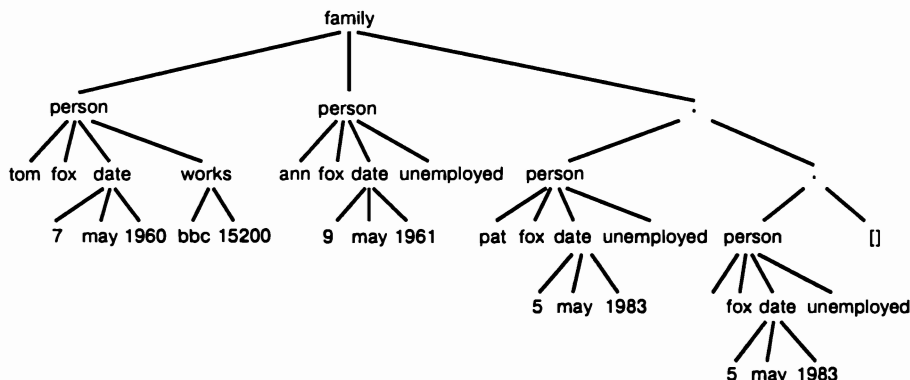


Рис. 4.1. Пример структурного представления информации о семье

Таким образом, рассматриваемая база данных может состоять из последовательности фактов, подобных этому, в которых описаны все семьи, представляющие интерес для данной программы.

Prolog фактически является языком, который очень хорошо приспособлен для выборки требуемой информации из подобной базы данных. Замечательным свойством этого языка является то, что он позволяет ссылаться на объекты, фактически не задавая все компоненты этих объектов. Prolog предоставляет возможность указывать структуру интересующих нас объектов и оставлять конкретные компоненты в этой структуре не заданными или заданными лишь частично. Некоторые примеры этого приведены на рис. 4.2. В частности, можно указать все семьи Армстронг с помощью следующего термина:

```
family( person( _, armstrong, _, _), _, _ )
```

Символы подчеркивания обозначают различные анонимные переменные; при их использовании можно не беспокоиться о том, как определить их значения. Кроме того, можно обозначить все семьи с тремя детьми с помощью следующего термина:

```
family( _, _, [_,_,_] )
```

Чтобы найти всех замужних женщин, имеющих по меньшей мере трех детей, можно сформировать такой вопрос:

```
?- family( _, person( Name, Surname, _, _), [_,_,_] ).
```

Из этих примеров следует вывод, что интересующие нас объекты можно определять не только по их содержимому, но и путем указания их структуры. Достаточно обозначить структуру этих объектов и оставить их параметры в виде незаполненных ячеек.

Кроме того, существует возможность предусмотреть набор процедур, которые будут служить в качестве утилит, позволяющих повысить удобство взаимодействия с базой данных. Подобные вспомогательные процедуры могут стать частью пользовательского интерфейса. Некоторые полезные вспомогательные процедуры для рассматриваемой базы данных приведены ниже.

```

husband( X ) :- % X - муж
    family(X, _, _).

wife( X ) :- % X - жена
    family( _, X, _).

child( X ) :- % X - ребенок
    family( _, _, Children),
    member( X, Children). % X задан в списке Children

exists( Person ) :- % Любой человек, описанный в базе данных
    husband( Person)
    ;
    wife( Person)
    ;
    child( Person).

dateofbirth( person( _, _, Date, _), Date).

salary( person( _, _, works( _, S)), S ). % Зарплата работающего лица

salary( person( _, _, unemployed), 0). % Зарплата безработного лица

```

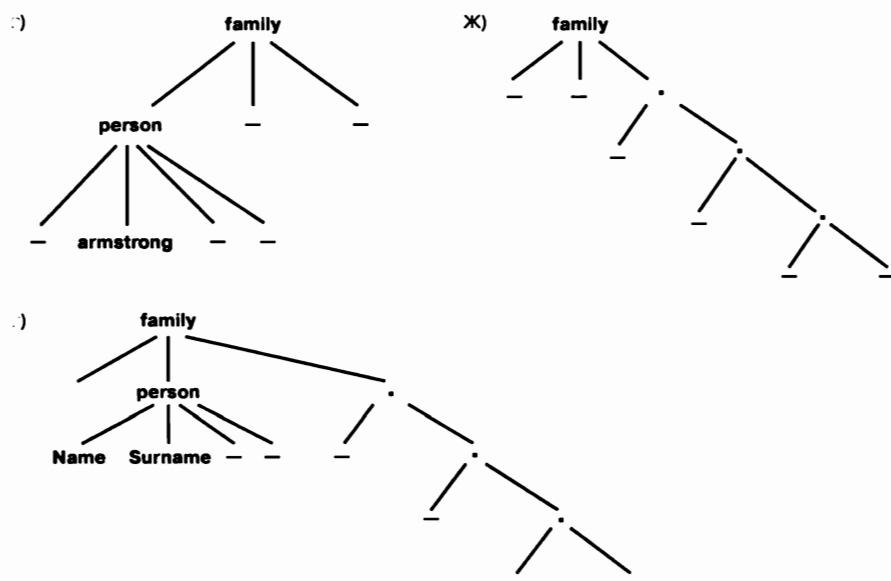


Рис. 4.2. Примеры определения объектов по их структурным свойствам: а) любая семья Армстронг; б) любая семья, в которой имеются точно три ребенка; в) любая семья, имеющая по меньшей мере трех детей. В структуре в) предусмотрена возможность выборки имени и фамилии жены путем конкретизации переменных Name и Surname

Такие утилиты можно, например, использовать в приведенных ниже запросах к базе данных.

- Найти имена всех людей, представленных в базе данных:
`?- exists( person( Name, Surname, _, _)).`
- Найти информацию обо всех детях, родившихся в 2000 году:
`?- child(X),
 dateofbirth(X, date(_, _, 2000)).`

■ **Найти имена всех работающих жен:**

```
?- wife( person( Name, Surname, _, works( _, _ ) ) ).
```

■ **Найти имена всех безработных людей, которые родились до 1973 года:**

```
?- exists( person( Name, Surname, date( _, _, Year), unemployed)),  
Year < 1973.
```

■ **Найти имена людей, родившихся до 1960 года, которые имеют заработную плату меньше 8000:**

```
?- exists( Person,  
dateofbirth( Person, date( _, _, Year)),  
Year < 1960,  
salary( Person, Salary),  
Salary < 8000.
```

■ **Найти фамилии отцов семейства, имеющих не меньше трех детей:**

```
?- family( person( _, Name, _, _), _, [_ , _ , _]).
```

Чтобы рассчитать общий доход семьи, удобно определить сумму значений заработной платы людей, представленных в списке, в виде следующего отношения с двумя параметрами:

```
total( List_of_people, Sum_of_their_salaries)
```

Это отношение может быть представлено с помощью приведенной ниже программы.

```
total( [],0). % Пустой список людей  
total( [Person | List], Sum) :-  
    salary( Person, S), % S - заработок первого человека в списке  
    total( List, Rest), % Rest - сумма заработков остальных людей в списке  
    Sum is S + Rest.
```

В этом случае общий доход любой семьи можно определить с помощью следующего вопроса:

```
?- family( Husband, Wife, Children),  
total( [Husband, Wife | Children], Income).
```

Предположим, что для подсчета количества элементов в списке может применяться отношение `length`, как определено в разделе 3.4. В таком случае можно найти все семьи, имеющие доход на члена семьи меньше 2000, с помощью следующего вопроса:

```
?- family( Husband, Wife, Children),  
total( [Husband, Wife | Children], Income),  
length( [Husband, Wife | Children], N), % N - количество членов семьи  
Income/N < 2000.
```

Упражнения

4.1. Напишите запросы для поиска следующей информации в базе данных о семьях:

- а) найти фамилии мужей в семьях, не имеющих детей;
- б) найти всех работающих детей;
- в) найти фамилии мужей в семьях с работающими женами и безработными мужьями;
- г) найти всех детей, родители которых отличаются по возрасту по меньшей мере на 15 лет.

4.2. Определите следующее отношение:

```
twins( Child1, Child2)
```

для поиска информации о близнецах в базе данных о семьях.

4.2. Методы абстрагирования данных

Абстрагирование данных можно рассматривать как процесс организации различных фрагментов информации в виде удобных для работы компонентов (возможно, по принципу создания иерархии) и, тем самым, структуризации информации в некоторой концептуально значимой форме. Каждый подобный информационный компонент должен быть легко доступным в программе. В идеальном случае все подробные сведения о реализации, такие как структура, должны быть невидимыми для пользователя информационной структуры; в таком случае программист сможет сосредоточиться на самих объектах и отношениях между ними. Подобный процесс предназначен для того, чтобы можно было обеспечить для программиста возможность использовать саму информацию, не думая о том, как фактически представлена эта информация.

Рассмотрим один из способов осуществления данного принципа в языке Prolog. Для этого снова вернемся к примеру с информацией о семье из предыдущего раздела. Информация о каждой семье представляет собой коллекцию различных фрагментов информации. Все эти фрагменты собраны в такие естественные информационные компоненты, как данные об отдельном лице или о семье, чтобы их можно было рассматривать как целостные объекты. Снова предположим, что информация о семье структурирована, как показано на рис. 4.1. В предыдущем разделе сведения о каждой семье были представлены в виде предложения Prolog. А в этом разделе данные о семье будут представлены в виде структурированного объекта, как показано ниже.

```
FoxFamily = family( person( tom, fox, _,_ ),_,_ )
```

Определим некоторые отношения, с помощью которых пользователь сможет обращаться к отдельным компонентам информации о семье без учета сведений, представленных на рис. 4.1. Подобные отношения можно назвать *селективными*, поскольку они позволяют извлекать из базы данных определенные компоненты. В качестве имени такого селективного отношения (или просто *селектора*) применяется имя выбираемого с его помощью компонента. Каждое отношение будет иметь два параметра: объект, содержащий компонент, и сам компонент:

```
selector_relation( Object, Component_selected)
```

Ниже приведены некоторые селекторы для структуры с описанием семьи.

```
husband( family( Husband, _,_ ), Husband ).
wife( family( _, Wife, _ ), Wife ).
children( family( _,_, ChildList ), ChildList ).
```

Кроме того, можно определить селекторы для первого и второго по счету ребенка следующим образом:

```
firstchild( Family, First ) :-
    children( Family, [First | _] ).
secondchild( Family, Second ) :-
    children( Family, [_ , Second | _] ).
```

Это определение можно уточнить, чтобы иметь возможность выбирать *n*-го ребенка, таким образом:

```
nthchild( N, Family, Child ) :-
    children( Family, ChildList ),
    nth_member( N, ChildList, Child ). % n-й элемент списка
```

Еще одним интересным объектом является отдельное лицо. Ниже приведены некоторые селекторы, которые соответствуют структуре, приведенной на рис. 4.1.

```
firstname( person( Name,_,_,_ ), Name ).
surname( person( _,Surname,_,_ ), Surname ).
born( person( _,_,Date,_,_ ), Date ).
```

Преимущество селективных отношений состоит в том, что после их определения можно забыть о том, с помощью какого конкретного способа представлена та или иная структурированная информация. Для создания и манипулирования этой ин-

формацией достаточно знать имена селективных отношений и использовать их в остальной части программы. В случае более сложных вариантов представления их использовать проще, чем всегда явно ссылаться на сами представления. В частности, в рассматриваемом примере семьи пользователю не требуется знать, что информация о детях представлена в виде списка. Предположим, например, что необходимо сформировать утверждение о том, что Том Фокс и Джим Фокс принадлежат к одной и той же семье и что Джим — второй ребенок Тома. С использованием приведенных выше селективных отношений можно определить два лица, скажем, `Person1` и `Person2`, и саму семью. Такую задачу позволяет выполнить следующий список целей:

```
firstname( Person1, tom), surname( Person1, fox), % Person1 - Том Фокс
firstname( Person2, jim), surname( Person2, fox), % Person2 - Джим Фокс
husband( Family, Person1),
secondchild( Family, Person2)
```

В результате конкретизация переменных `Person1`, `Person2` и `Family` будет выполнена следующим образом:

```
Person1 = person( tom, fox, _, _ )
Person2 = person( jim, fox, _, _ )
Family = family( person(tom,fox,_,_), _, [_ ,person(jim,fox) | _])
```

Использование селективных отношений позволяет также проще модифицировать программы. Достаточно представить себе, что возникла необходимость повысить эффективность программы путем выбора иного способа представления данных. После этого достаточно просто изменить определения селективных отношений, и остальная часть программы будет работать в неизменном виде с этим новым представлением.

Упражнение

4.3. Дополните определение отношения `nthchild`, сформировав следующее отношение:

```
nth_member( N, List, X)
```

которое принимает истинное значение, если `X` является n -м элементом списка `List`.

4.3. Моделирование недетерминированного конечного автомата

В этом примере показано, как абстрактную математическую конструкцию можно перевести на язык Prolog. Кроме того, данный пример показывает, что полученная в результате программа эмулятора конечного автомата может оказаться гораздо более гибкой, чем предполагалось первоначально.

Недетерминированным конечным автоматом называется абстрактная машина, которая считывает в качестве входных данных строки символов и принимает решение о том, следует ли принять или отвергнуть ту или иную входную строку. Конечный автомат имеет целый ряд состояний и всегда находится в одном из состояний. Он способен изменять свое состояние, переходя из текущего в другое возможное состояние. Внутреннюю структуру конечного автомата можно представить с помощью графа переходов, подобного приведенному на рис. 4.3. В данном случае узлы графа s_1 , s_2 , s_3 и s_4 обозначают состояния автомата. Начиная с начального состояния (в этом примере — s_1), автомат переходит из одного состояния в другое, считывая входную строку. Направления переходов зависят от текущего входного символа; эти символы показаны в виде меток на дугах в графе переходов.

Переход происходит после чтения каждого входного символа. Следует отметить, что переходы могут быть недетерминированными. Например, если автомат, показанный на рис. 4.3, находится в состоянии s_1 и текущим входным символом является `a`,

он может перейти либо в s_1 , либо в s_2 . Некоторые дуги отмечены меткой `null`, которая означает *пустой символ*. Такие дуги соответствуют так называемым *скрытым переходам* автомата. Подобные переходы называются *скрытыми*, поскольку они происходят без чтения входных данных, а наблюдатель, рассматривающий автомат как “черный ящик”, не способен обнаружить, что произошел какой-либо переход.

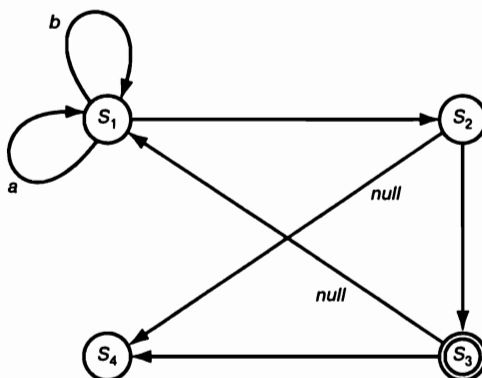


Рис. 4.3. Пример недетерминированного конечного автомата

Состояние s_3 обозначено двойным кружком, который указывает на то, что это — *конечное состояние*. Считается, что автомат принимает входную строку, если в графе имеется путь перехода, такой, что

1. он начинается с начального состояния;
2. он оканчивается конечным состоянием;
3. метки дуг вдоль пути полностью соответствуют входной строке.

Задача принятия решения о том, какой из возможных переходов должен быть выполнен в тот или иной момент времени, полностью возложена на конечный автомат. В частности, автомат может выбрать вариант действий, предусматривающий или не предусматривающий выполнение скрытого перехода, если этот переход возможен в текущем состоянии. Но абстрактные недетерминированные машины такого рода имеют одно так называемое *магическое свойство* (т.е. свойство, не обусловленное их структурой): если имеется выбор, они всегда выбирают “правильный” переход, ведущий к принятию входной строки, если таковой существует. Например, автомат, показанный на рис. 4.3, принимает строки `ab` и `aabaab`, но отвергает строки `abb` и `abba`. Можно легко понять, что этот автомат принимает любую строку, которая оканчивается на `ab`, и отвергает все прочие строки.

В языке Prolog конечный автомат может быть задан с помощью трех описанных ниже отношений.

1. Унарное отношение `final`, которое определяет конечное состояние автомата.
2. Отношение `trans` с тремя параметрами, которое определяет переходы между состояниями таким образом:

```
trans(S1, X, S2)
```

Это означает, что переход из состояния s_1 в состояние s_2 возможен, если считан текущий входной символ X .

3. Бинарное отношение

```
silent(S1, S2)
```

которое означает, что возможен скрытый переход из состояния s_1 в состояние s_2 .

Для конечного автомата, показанного на рис. 4.3, эти три отношения заданы следующим образом:

```
final( S3).
trans( S1, a, S1).
trans( S1, a, S2).
trans( S1, b, S1).
trans( S2, b, S3).
trans( S3, b, S4).
silent( S2, S4).
silent( S3, S1).
```

В дальнейшем входные строки будут представлены в виде списков Prolog. Например, строка `aab` будет представлена как `[a,a,b]`. Программа эмулятора конечного автомата, в которую введено описание этого автомата, должна обрабатывать заданную входную строку и принимать решение о том, следует ли принять или отвергнуть эту строку. По определению недетерминированный автомат принимает заданную строку (начиная с начального состояния), если после чтения всей входной строки автомат может (по всей вероятности) находиться в своем конечном состоянии. Программа эмулятора разрабатывается как бинарное отношение `accepts`, которое определяет, может ли быть принята некоторая строка из текущего состояния. Таким образом, предикат

```
accepts( State, String)
```

принимает истинное значение, если автомат, начиная с состояния `State`, рассматриваемого как начальное состояние, воспринимает строку `String`. Отношение `accepts` может быть определено тремя предложениями, которые соответствуют трем описанным ниже случаям.

1. В состоянии `State` принята пустая строка, `[]`, притом что `State` является конечным состоянием.
2. В состоянии `State` принята непустая строка, если чтение первого символа строки может перевести автомат в некоторое состояние, `State1`, и остальная часть строки принята в состоянии `State1`, как показано на рис. 4.4, а.
3. В состоянии `State` принята строка, если автомат может выполнить скрытый переход из состояния `State` в состояние `State1`, а затем принять (всю) входную строку в состоянии `State1`, как показано на рис. 4.4, б.

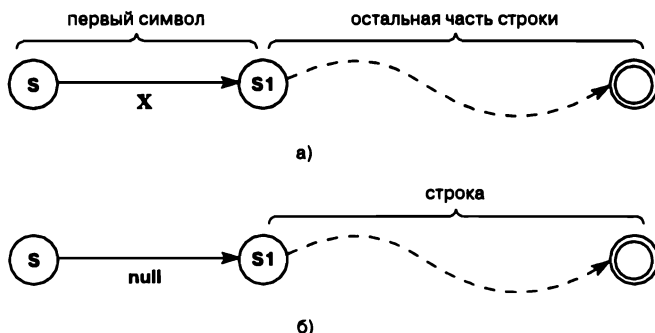


Рис. 4.4. Варианты принятия строки: а) после чтения ее первого символа `X`; б) после выполнения скрытого перехода

Эти правила можно перевести на язык Prolog следующим образом:

```
accepts( State, [ ] ) :- % Принять пустую строку
    final( State).
```

```

accepts( State, [X | Rest] ) :- % Принять после чтения первого символа
    trans( State, X, Statel),
    accepts( Statel, Rest).

accepts( State, String) :- % Принять после выполнения скрытого перехода
    silent( State, Statel),
    accepts( Statel, String).

```

Этой программе можно, например, задать вопрос о возможности принятия строки aaab, следующим образом:

```

?- accepts( S1, [a,a,a,b]).
yes

```

Как уже было показано, программы Prolog часто способны решать более общие проблемы, чем те, для решения которых они первоначально были разработаны. В данном случае можно также задать эмулятору приведенный ниже вопрос о том, в каком первоначальном состоянии должен находиться данный конечный автомат, чтобы он мог принять строку ab.

```

?- accepts( S, [a,b]).
S = S1;
S = S3

```

Любопытно отметить, что можно также задать вопрос, каковы все строки с длиной 3, которые могут быть приняты в состоянии s_1 .

```

?- accepts( S1, [X1,X2,X3]).
X1 = a
X2 = a
X3 = b;
X1 = b
X2 = a
X3 = b;
no

```

Если желательно, чтобы приемлемые входные строки можно было ввести в виде списков, то данный вопрос можно сформулировать следующим образом:

```

?- String = [_,_,_], accepts( S1, String).
String = [a,a,b];
String = [b,a,b];
no

```

С этой программой можно проводить дальнейшие эксперименты, задавая даже еще более общие вопросы, например, о том, из каких состояний автомат принимает входные строки с длиной 7.

Для последующего экспериментирования может потребоваться внесение изменений в структуру автомата путем модификации отношений `final`, `trans` и `silent`. Автомат, приведенный на рис. 4.3, не содержит каких-либо циклических скрытых путей (путей, состоящих только из скрытых переходов). А если в автомат на рис. 4.3 будет введен новый переход:

```

silent( S1, S3)

```

то в нем появится *скрытый цикл*. Но теперь работа эмулятора может нарушиться, например, при поиске ответа на следующий вопрос:

```

?- accepts( S1, [a]).

```

Этот вопрос вынудит эмулятор до бесконечности переходить по циклу в состояние s_1 , притом что программа будет постоянно искать какой-то способ перейти в конечное состояние.

Упражнения

- 4.4. Почему не может произойти заикливание в эмуляторе первоначального автомата, показанного на рис. 4.3, который характеризуется тем, что в графе переходов отсутствует скрытый цикл?

4.5. Заикливание при выполнении отношения `accepts` можно предотвратить, например, путем подсчета количества переходов, сделанных до сих пор. В таком случае появляется возможность выдвинуть требование, чтобы эмулятор отыскивал только пути с некоторой ограниченной длиной. Модифицируйте отношение `accepts` таким образом. Подсказка: введите третий параметр — максимальное количество допустимых переходов, как показано ниже.

```
accepts( State, String, MaxMoves)
```

4.4. Консультант бюро путешествий

В данном разделе показано, как разработать программу, которая дает консультации по планированию авиаперелетов. Эта программа является довольно простым консультантом, но обладает способностью отвечать на некоторые важные вопросы, наподобие приведенных ниже.

- В какие дни недели имеется прямой вечерний рейс из Любляны в Лондон?
- Как долететь из Любляны в Эдинбург в четверг?
- Мне нужно посетить Милан, Люблян и Цюрих, отправившись из Лондона во вторник и вернувшись в Лондон в пятницу. В какой последовательности мне следует посещать эти города, чтобы не приходилось совершать больше одного перелета в каждый день путешествия?

Эта программа должна быть сосредоточена вокруг базы данных, содержащей информацию об авиарейсах. Такая информация представлена в виде следующего отношения с тремя параметрами:

```
timetable( Place1, Place2, ListOfFlights)
```

где `ListOfFlights` — список структурированных элементов в следующей форме:

```
DepartureTime / ArrivalTime / FlightNumber / ListOfDays
```

В данном случае знак операции `“/”` лишь соединяет друг с другом компоненты структуры и, безусловно, не означает арифметического деления. Переменная `ListOfDay` может представлять собой либо список дней недели, либо атом `alldays` с обозначением ежедневного рейса. Одно из предложений отношения `timetable` может, например, выглядеть таким образом:

```
timetable( london, edinburgh,  
[ 9:40 / 10:50 / ba4733 / alldays,  
19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).
```

Значения времени представлены как структурированные объекты с двумя компонентами (часы и минуты), которые соединены знаком операции `“:”`.

Главная проблема состоит в том, что необходимо иметь возможность точно определять маршруты между двумя указанными городами в указанный день недели. Это требование можно учесть в программе с помощью следующего отношения с четырьмя параметрами:

```
route( Place1, Place2, Day, Route)
```

где `Route` — последовательность авиаперелетов, которая удовлетворяет следующим критериям.

1. Начальной точкой маршрута является `Place1`.
2. Конечная точка маршрута — `Place2`.
3. Все авиаперелеты должны происходить в один и тот же день недели, `Day`.
4. Все авиаперелеты, заданные в маршруте `Route`, должны осуществляться с помощью авиарейсов, которые определены в отношении `timetable`.
5. Должно быть предусмотрено достаточное время для пересадки с одного авиарейса на другой.

Маршрут представлен в виде списка структурированных объектов, имеющих следующую форму:

From / To / FlightNumber / Departure_time

Кроме того, предусмотрено использование перечисленных ниже вспомогательных предикатов.

1) `flight( Place1, Place2, Day, FlightNum, DepTime, ArrTime)`

Этот предикат указывает, что существует авиарейс `FlightNum`, на котором может быть осуществлен авиаперелет из города `Place1` в город `Place2` в день недели `Day`, с указанным временем отправления и прибытия.

2) `deptime( Route, Time)`

Временем отправления по маршруту `Route` является `Time`.

3) `transfer( Time1, Time2)`

Между значениями времени `Time1` и `Time2` должен быть предусмотрен промежуток времени не меньше 40 минут, которого будет достаточно для пересадки с одного рейса на другой.

Проблема поиска маршрута напоминает задачу моделирования недетерминированного конечного автомата, рассматриваемую в предыдущем разделе. Между этими двумя проблемами имеются указанные ниже аналогии.

- Состояния конечного автомата соответствуют пребыванию в тех или иных городах.
- Переходы между двумя состояниями соответствуют перелету из одного города в другой.
- Отношение `transition` конечного автомата соответствует отношению `timetable`.
- Эмулятор конечного автомата находит в графе переходов путь от начального состояния к конечному; программа планирования путешествий находит маршрут между начальной и конечной точками маршрута.

Поэтому нет ничего удивительного в том, что отношение `route` может быть определено по аналогии с отношением `accepts`, за исключением того, что в этом случае отсутствуют скрытые переходы. При этом могут рассматриваться два перечисленных ниже случая.

1. Выполнение задачи с помощью прямого рейса. Если существует прямой рейс между городами `Place1` и `Place2`, то маршрут состоит из этого рейса:

```
route( Place1, Place2, Day, [ Place1 / Place2 / Fnum / Dep ] ) :-  
    flight( Place1, Place2, Day, Fnum, Dep, An).
```

2. Выполнение задачи с помощью рейсов с пересадкой. Маршрут между городами `P1` и `P2` состоит из первого рейса, от города `P1` до некоторого промежуточного города `P3`, за которым следует рейс из города `P3` в город `P2`. Кроме того, должно быть достаточно времени между прибытием первого рейса и отправлением второго для выполнения пересадки.

```
route( P1, P2, Day, [ P1 / P3 / Fnum1 / Dep1 | RestRoute ] ) :-  
    route( P3, P2, Day, RestRoute ),  
    flight( P1, P3, Day, Fnum1, Dep1, Arr1 ),  
    deptime( RestRoute, Dep2 ),  
    transfer( Arr1, Dep2 ).
```

Программы для вспомогательных отношений `flight`, `transfer` и `deptime` можно легко разработать, и они включены в полную программу планирования путешествий, приведенную в листинге 4.1. Кроме того, в нее в качестве примера включена база данных о расписании авиаперелетов.

Листинг 4.1. Планировщик маршрутов, состоящих из отдельных авиаперелетов, и вымышленное расписание авиарейсов

% ПЛАНИРОВЩИК МАРШРУТОВ ПОЛЕТА

```
:- op( 50, xfy, :).
```

```
% route( Place1, Place2, Day, Route):
```

```
% Route - последовательность полетов, начинающихся в городе Place1 и  
заканчивающихся в городе Place2, проводимых в день Day
```

```
route( P1, P2, Day, [ P1 / P2 / Fnum / Deptime ] ) :- % Прямой рейс  
flight( P1, P2, Day, Fnum, Deptime, _).
```

```
% Рейс с пересадками
```

```
route( P1, P2, Day, [ (P1 / P3 / Fnum1 / Dep1) | RestRoute] ) :-  
route( P3, P2, Day, RestRoute),  
flight( P1, P3, Day, Fnum1, Dep1, Arr1),  
deptime( RestRoute, Dep2), % Время отправления по маршруту  
transfer( Arr1, Dep2). % Достаточное время для пересадки
```

```
flight( Place1, Place2, Day, Fnum, Deptime, Arrtime) :-  
timetable( Place1, Place2, Flightlist),  
member( Deptime / Arrtime / Fnum / Daylist, Flightlist),  
flyday( Day, Daylist).
```

```
flyday( Day, Daylist) :-  
member( Day, Daylist).
```

```
flyday( Day, alldays) :-  
member( Day, [mo,tu,we,th,fr,sa,su] ).
```

```
deptime( [ P1 / P2 / Fnum / Dep | _], Dep).
```

```
transfer( Hours1:Mins1, Hours2:Mins2) :-  
60 * (Hours2 - Hours1) + Mins2 - Mins1 >= 40.
```

```
member( X, [X | L] ).
```

```
member( X, [Y | L] ) :-  
member( X, L).
```

% БАЗА ДАННЫХ О ПОЛЕТАХ

```
timetable( edinburgh, london,  
[ 9:40 / 10:50 / ba4733 / alldays,  
13:40 / 14:50 / ba4773 / alldays,  
19:40 / 20:50 / ba4833 / [mo,tu,we,th,fr,su] ] ).
```

```
timetable( london, edinburgh,  
[ 9:40 / 10:50 / ba4732 / alldays,  
11:40 / 12:50 / ba4752 / alldays,  
18:40 / 19:50 / ba4822 / [mo,tu,we,th,fr] ] ).
```

```
timetable( london, ljubljana,  
[ 13:20 / 16:20 / jp212 / [mo,tu,we,fr,su],  
16:30 / 19:30 / ba473 / [mo,we,th,sa] ] ).
```

```
timetable( london, zurich,  
[ 9:10 / 11:45 / ba614 / alldays,  
14:45 / 17:20 / sr805 / alldays ] ).
```

```
timetable( london, milan,  
[ 8:30 / 11:20 / ba510 / alldays,  
11:00 / 13:50 / az459 / alldays ] ).
```

```

timetable( ljubljana, zurich,
  [ 11:30 / 12:40 / jp322 / [tu,th] ] ).

timetable( ljubljana, london,
  [ 11:10 / 12:20 / jp211 / [mo,tu,we,fr,su],
    20:30 / 21:30 / ba472 / [mo,we,th,sa] ] ).

timetable( milan, london,
  [ 9:10 / 10:00 / az458 / alldays,
    12:20 / 13:10 / ba511 / alldays ] ).

timetable( milan, zurich,
  [ 9:25 / 10:15 / sr621 / alldays,
    12:45 / 13:35 / sr623 / alldays ] ).

timetable( zurich, ljubljana,
  [ 13:30 / 14:40 / jp323 / [tu,th] ] ).

timetable( zurich, london,
  [ 9:00 / 9:40 / ba613 / [mo,tu,we,th,fr,sa],
    16:10 / 16:55 / sr806 / [mo,tu,we,th,fr,su] ] ).

timetable( zurich, milan,
  [ 7:55 / 8:45 / sr620 / alldays ] ).

query3(City1,City2,City3,FN1,FN2,FN3,FN4) :-
  permutation( [milan,ljubljana,zurich],[City1,City2,City3]),
  flight( london, City1, tu, FN1, Dep1, Arr1),
  flight( City1, City2, we, FN2, Dep2, Arr2),
  flight( City2, City3, th, FN3, Dep3, Arr3),
  flight( City3, london, fr, FN4, Dep4, Arr4).

conc([], L, L).

conc([X|L1],L2, [X|L3]) :-
  conc(L1,L2,L3).

permutation( [], [] ).

permutation( L, [X | P] ) :-
  del( X, L, L1),
  permutation( L1, P).

del( X, [X|L], L).

del( X, [Y|L], [Y|L1] ) :-
  del( X, L, L1).

```

Рассматриваемый планировщик маршрутов является чрезвычайно простым и способен заниматься исследованием даже таких маршрутов, которые, безусловно, никуда не ведут. Тем не менее он успешно справляется со своей работой, если база данных об авиарейсах невелика. А при наличии действительно крупной базы данных требуется более интеллектуальное планирование, которое позволило бы справиться с большим количеством потенциальных рассматриваемых маршрутов.

Ниже приведены некоторые примеры вопросов к программе.

- В какие дни недели существует прямой вечерний рейс из Любляны в Лондон?

```

?- flight( ljubljana, london, Day, _, DeptHour:_, _), DeptHour >= 18.
Day = mo;
Day = we;
...

```

- Как можно добраться из Любляны в Эдинбург в четверг?

```

?- route( ljubljana, edinburgh, th, R).
R = [ ljubljana / zurich / jp322 / 11:30, zurich / london / sr806 /

```

```
16:10,  
london / edinburgh / ba4822 / 18:40]
```

- Мне нужно посетить Милан, Любляну и Цюрих, отправившись из Лондона во вторник и вернувшись в Лондон в пятницу. В какой последовательности мне следует посетить эти города, чтобы не приходилось совершать больше одного полета в каждый день путешествия?

Этот вопрос немного сложнее. Его можно сформулировать с использованием отношения `permutation`, программа которого приведена в главе 3. Пользователь фактически просит найти такую перестановку, которая позволяет выбрать последовательность посещения городов Милан, Любляна и Цюрих, чтобы соответствующие перелеты были возможны в подряд идущие дни:

```
?- permutation( [milan, ljubljana, zurich], [City1, City2, City3]),  
flight( london, City1, tu, FN1, _, _),  
flight( City1, City2, we, FN2, _, _),  
flight( City2, City3, th, FN3, _, _),  
flight( City3, london, fr, FN4, _, _).  
City1 = milan  
City2 = zurich  
City3 = ljubljana  
FN1 = ba510  
FN2 = sr621  
FN3 = jp323  
FN4 = jp211
```

Наконец, отметим, что эта программа восприимчива к бесконечным циклам, которые могут возникать, например, при поиске ответа на вопрос, связанный с определением маршрута, информация о котором отсутствует в расписании:

```
?- route ( moscow, edinburgh, mo, R).
```

Поэтому лучше всего обеспечивать безопасность вопросов, ограничивая длину маршрута. Для этого может использоваться следующий обычный прием для ограничения с помощью отношения `conc`:

```
?- conc( R, _, [_,_,_,_]), route( moscow, edinburgh, mo, R).  
no
```

Цель `conc` ограничивает список `R` длиной 4, а также вынуждает в первую очередь рассматривать при поиске кратчайшие маршруты.

4.5. Задача с восемью ферзями

Эта задача состоит в определении того, как расставить восемь ферзей на пустой шахматной доске таким образом, чтобы ни один ферзь не нападал на другого. Для решения этой задачи может быть составлена программа в виде унарного предиката:

```
solution( Pos)
```

который принимает истинное значение, если и только если `Pos` представляет собой позицию на шахматной доске, в которой все восемь ферзей не нападают друг на друга. Кроме того, любопытно сравнить различные идеи по составлению программы для решения данной задачи. Поэтому в данном разделе представлены три программы, основанные на разных способах представления этой проблемы.

4.5.1. Программа 1

Вначале рассмотрим вариант, в котором решение исходит из представления позиции на доске. Вполне обоснованно можно предположить, что позицию допустимо представить как список из восьми элементов, каждый из которых соответствует одному ферзю. Каждый элемент в списке обозначает клетку доски, на которой стоит соответствующий ферзь. Кроме того, каждая клетка на доске может быть обозначена

парой координат (X и Y), где каждая координата представляет собой целое число от 1 до 8. В программе эта пара может быть записана следующим образом:

X/Y

Здесь, безусловно, знак операции “/” не обозначает операцию деления, а просто позволяет объединить обе координаты одной клетки. На рис. 4.5 показано одно из решений задачи с восемью ферзями и его представление в виде списка.

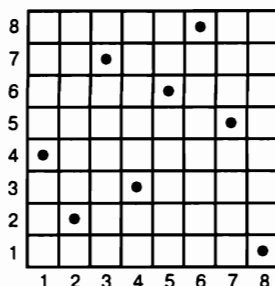


Рис. 4.5. Решение задачи с восемью ферзями. Эту позицию можно представить в виде списка [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1]

После выбора такого способа представления задачу можно свести к поиску списка, представленного в форме

[X1/Y1, X2/Y2, X3/Y3, ..., X8/Y8]

который удовлетворяет требованию о том, что ни один ферзь не нападает на другого. Рассматриваемая процедура solution должна обеспечивать поиск соответствующей конкретизации переменных X1, Y1, X2, Y2, ..., X8, Y8. Но, поскольку известно, что все ферзи должны стоять на разных столбцах доски для предотвращения нападений по вертикали, можно сразу же ограничить соответствующий выбор и, тем самым, упростить задачу поиска. Таким образом, можно зафиксировать координаты X, чтобы список с решением определялся по следующему, более конкретному образцу:

[1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]

Мы заинтересованы в поиске решения задачи на доске с размерами 8×8. Но в программировании ключом к решению часто является рассмотрение более общей проблемы. Как это ни парадоксально, часто возникает такая ситуация, что решение более общей проблемы проще сформулировать по сравнению с более узкой, первоначальной проблемой. После этого первоначальная проблема решается как частный случай более общей проблемы.

Творческая часть решения проблемы состоит в поиске приемлемого обобщения первоначальной проблемы. В данном случае хорошая идея состоит в том, что количество ферзей (количество столбцов в списке) можно обобщить и принять равным не 8, а любому числу, включая нуль. В таком случае требование к отношению solution можно сформулировать, рассматривая два приведенных ниже случая.

Случай 1. Список ферзей пуст. Пустой список, безусловно, является решением проблемы, поскольку при отсутствии ферзей отсутствует и возможность для нападения.

Случай 2. Список ферзей не пуст. При этом он должен выглядеть примерно так: [X/Y | Others].

В случае 2 первый ферзь стоит на некоторой клетке X/Y , а остальные ферзи распределены по клеткам, обозначенным списком `Others`. Если этот список представляет собой одно из решений, то должны соблюдаться приведенные ниже условия.

1. Между ферзями, перечисленными в списке `Others`, не должно быть возможно обоюдное нападение; это означает, что сам список `Others` также представляет собой решение.
2. X и Y должны быть целыми числами от 1 до 8.
3. Ферзь, стоящий на клетке X/Y , не должен нападать ни на один из ферзей в списке `Others`.

Для разработки программы, соответствующей первому условию, можно использовать само отношение `solution`. Второе условие можно определить следующим образом: переменная Y должна входить в состав списка целых чисел от 1 до 8, т.е. списка `[1,2,3,4,5,6,7,8]`. С другой стороны, не следует беспокоиться о выборе значения X , поскольку список с решением должен соответствовать образцу, в котором координаты X уже определены. Поэтому гарантируется, что переменная X будет иметь подходящее значение от 1 до 8. Третье условие можно реализовать еще в одном отношении, `noattack`. Таким образом, все эти утверждения можно записать на языке Prolog следующим образом:

```
solution( [X/Y | Others] ) :-  
    solution( Others ),  
    member( Y, [1,2,3,4,5,6,7,8] ),  
    noattack( X/Y, Others ).
```

Теперь осталось только определить отношение `noattack`:

```
noattack( Q, Qlist )
```

Ситуацию, связанную с применением этого отношения, можно снова разделить на два рассматриваемых ниже случая.

1. Если список `Qlist` пуст, то это отношение, безусловно, является истинным, поскольку отсутствует ферзь, который мог бы подвергнуться нападению.
2. Если список `Qlist` не пуст, то он имеет форму `[Q1 | Qlist1]` и должны быть удовлетворены следующие два условия:
 - а) ферзь, который стоит на клетке Q , не должен нападать на ферзя, стоящего на клетке $Q1$;
 - б) ферзь, стоящий на клетке Q , не должен нападать ни на одного из ферзей в списке `Qlist1`.

Для указания на то, что ферзь, стоящий на некоторой клетке, не нападает на ферзя, стоящего на другой клетке, достаточно сформулировать следующее простое условие: эти две клетки не должны находиться на одной и той же строке, на одном и том же столбце или на одной и той же диагонали. Применяемый образец решения гарантирует, что все ферзи находятся на разных столбцах, поэтому остается только явно определить следующее:

- координаты Y ферзей являются разными;
- ферзи не находятся на одной и той же восходящей (если двигаться слева направо) или нисходящей диагонали; это означает, что расстояние между клетками, на которых стоят ферзи, в направлении X не должно быть равно расстоянию между клетками в направлении Y .

Законченная программа приведена в листинге 4.2. Для упрощения ее использования был введен список с образцом. Выборку этого списка можно выполнить в вопросе, заданном для выработки решений. Поэтому теперь программе можно задать следующий вопрос:

```
?- template( S ), solution( S ).
```

и программа начнет вырабатывать решения следующим образом:

```
S = [ 1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1];
S = [ 1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1];
S = [ 1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1];
...
```

Листинг 4.2. Программа 1 для решения задачи с восемью ферзями

```
% solution( BoardPosition), если
%   BoardPosition - список ненападающих ферзей

solution( [] ).

solution( [X/Y | Others] ) :-
    solution( Others),           % Первый ферзь в клетке X/Y;
    member( Y, [1,2,3,4,5,6,7,8] ), % Others - список прочих ферзей
    noattack( X/Y, Others).      % Первый ферзь не нападает на других

noattack( _, [] ).              % Объект для нападения отсутствует

noattack( X/Y, [X1/Y1 | Others] ) :-
    Y =\= Y1,                   % Разные координаты Y
    Y1-Y =\= X1-X,              % Разные диагонали
    Y1-Y =\= X-X1,
    noattack( X/Y, Others).

member( Item, [Item | Rest] ).

member( Item, [First | Rest] ) :-
    member( Item, Rest).

% Образец решения

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

Упражнение

4.6. При поиске решения программа, приведенная в листинге 4.2, проверяет альтернативные значения для координат Y ферзей. В каком месте программы определяется порядок этих альтернатив? Как можно легко модифицировать эту программу для изменения такого порядка? Проведите эксперименты с использованием разных способов определения порядка альтернатив для изучения того, как от этого изменяется быстродействие программы.

4.5.2. Программа 2

При использовании представления позиции на доске при разработке программы 1 каждое решение имело следующую форму:

```
[ 1/Y1, 2/Y2, 3/Y3, ..., 8/Y8]
```

поскольку ферзи размещались на подряд идущих вертикальных рядах клеток доски. Но если бы данные о координатах X были опущены, это не привело бы к потере какой-либо информации. Поэтому может использоваться более экономичное представление позиции на доске, при котором сохраняются только координаты Y ферзей, следующим образом:

```
[ Y1, Y2, Y3, ..., Y8]
```

Для предотвращения нападений по горизонтали ни одна пара ферзей не должна находиться на одном и том же горизонтальном ряду клеток доски. Такое условие налагает ограничение на выбор координат Y. Ферзи должны занимать все горизонталь-

ные ряды 1, 2, ..., 8. Остается только сделать выбор правильного порядка этих восьми чисел. Поэтому каждое решение представляет собой одну из перестановок списка: [1, 2, 3, 4, 5, 6, 7, 8]

Такая перестановка, *S*, представляет собой решение, если все ферзи находятся в безопасности. Поэтому можно записать следующее правило:

```
solution( S ) :-
    permutation( [1,2,3,4,5,6,7,8], S ),
    safe( S ).
```

Программа для отношения *permutation* была разработана в главе 3, поэтому остается только определить отношение *safe*. Его определение можно разделить на два описанных ниже случая.

1. *S* — пустой список. Такой вариант, безусловно, является безопасным, поскольку отсутствует угроза нападения со стороны какого-либо из ферзей.
2. *S* — непустой список в форме [Queen | Others]. Эта ситуация является безопасной, если безопасным является список *Others* и ферзь *Queen* не нападает на каких-либо ферзей в списке *Others*.

В языке Prolog это определение может быть выражено следующим образом:

```
safe( []).
safe( [Queen | Others] ) :-
    safe( Others ),
    noattack( Queen, Others ).
```

Приведенное здесь отношение *noattack* определить немного сложнее. Сложность состоит в том, что позиции ферзей определены только координатами *Y*, а координаты *X* явно не представлены. Эту проблему можно решить благодаря небольшому обобщению отношения *noattack*, как показано на рис. 4.6. Следующая цель:

```
noattack( Queen, Others )
```

предназначена для обеспечения того, чтобы ферзь *Queen* не нападал на ферзей *Others*, если расстояние *X* между *Queen* и *Others* равно 1. В данном случае необходимо обобщить понятие расстояния *X* между *Queen* и *Others*. Поэтому введем это расстояние в качестве третьего параметра в отношение *noattack* следующим образом:

```
noattack( Queen, Others, Xdist )
```

Соответствующим образом цель *noattack* в отношении *safe* необходимо модифицировать следующим образом:

```
noattack( Queen, Others, 1 )
```

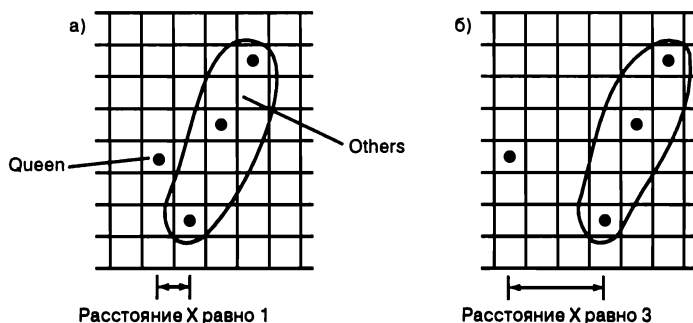


Рис. 4.6. Способ обобщения отношения *noattack*: а) расстояние *X* между *Queen* и *Others* равно 1; б) расстояние *X* между *Queen* и *Others* равно 3

Теперь отношение noattack можно сформулировать в соответствии с двумя рассматриваемыми случаями, в зависимости от списка Others: если список Others пуст, то отсутствует противник, на которого можно было бы напасть, и поэтому, безусловно, отсутствует нападение; если список Others не пуст, то ферзь Queen не должен нападать на первого ферзя в списке Others (который находится на расстоянии Xdist столбцов от Queen), а также на ферзя, находящегося в хвосте списка Others, который расположен на расстоянии Xdist + 1. Эти рассуждения приводят к созданию программы, показанной в листинге 4.3.

Листинг 4.3. Программа 2 для решения задачи с восемью ферзями

```
% solution( Queens), если
%   Queens - список координат Y восьми ненападающих ферзей

solution( Queens) :-
    permutation( [1,2,3,4,5,6,7,8], Queens),
    safe( Queens).

permutation( [], [] ).

permutation( [Head | Tail], PermList) :-
    permutation( Tail, PermTail),
    del( Head, PermList, PermTail).    % Вставить голову Head в список Tail,
                                     % подвергшийся перестановке

% del( Item, List, NewList) - список NewList получен путем удаления элемента
%                               Item из списка List

del( Item, [Item | List], List).

del( Item, [First | List], [First | List1] ) :-
    del( Item, List, List1).

% safe( Queens), если
%   Queens - список координат Y восьми ненападающих ферзей

safe( [] ).

safe( [Queen | Others] ) :-
    safe( Others),
    noattack( Queen, Others, 1).

noattack( _, [], _).

noattack( Y, [Y1 | Ylist], Xdist) :-
    Y1-Y =\= Xdist,
    Y-Y1 =\= Xdist,
    Dist1 is Xdist + 1,
    noattack( Y, Ylist, Dist1).
```

4.5.3. Программа 3

Третья программа решения задачи с восемью ферзями основана на следующих рассуждениях. Каждый ферзь должен находиться на некоторой клетке доски, т.е. в определенном вертикальном ряду, горизонтальном ряду, на восходящей и нисходящей диагонали. Для обеспечения того, чтобы все ферзи были в безопасности, их необходимо поместить на такие клетки, которые находятся в отличных от других вертикальных рядах, горизонтальных рядах, на разных восходящих и нисходящих диагоналях. Поэтому вполне естественно перейти к рассмотрению более развитого способа представления, с четырьмя координатами, как показано ниже.

- х. Вертикальный ряд.
- у. Горизонтальный ряд.
- u. Восходящая диагональ.
- v. Нисходящая диагональ.

Эти координаты не являются независимыми: в частности, значения u и v зависят от значений x и y (как показано на рис. 4.7). Такую зависимость можно представить с помощью следующих уравнений:

$$u = x - y$$

$$v = x + y$$

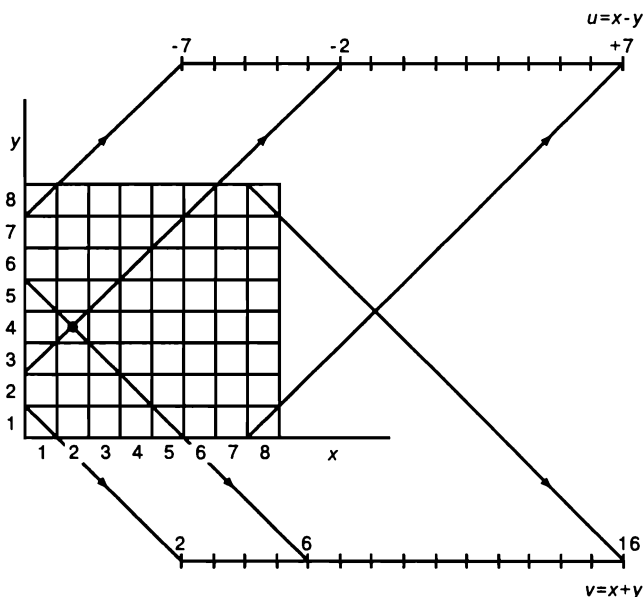


Рис. 4.7. Соотношение между координатами, заданными с помощью номеров вертикальных и горизонтальных рядов, а также восходящих и нисходящих диагоналей. Клетка, обозначенная точкой, имеет следующие координаты: $x = 2$, $y = 4$, $u = 2 - 4 = -2$, $v = 2 + 4 = 6$

Области определения для всех четырех измерений показаны ниже.

$Dx = [1, 2, 3, 4, 5, 6, 7, 8]$
 $Dy = [1, 2, 3, 4, 5, 6, 7, 8]$
 $Du = [-7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]$
 $Dv = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]$

Теперь задачу с восемью ферзями можно сформулировать следующим образом: выбрать восемь четырехэлементных кортежей (X, Y, U, V) из соответствующих областей определения (X из Dx , Y из Dy и т.д.), никогда не используя дважды одни и те же элементы из любой области определений. Безусловно, после выбора X и Y значения U и V становятся вполне определенными. Поэтому решение, грубо говоря, может состоять в следующем: учитывая все четыре области определения, выбрать позицию первого ферзя, удалить соответствующие элементы из этих четырех областей определения, а затем использовать остальную часть областей определения для размещения оставшихся ферзей. Программа, основанная на этой идее, приведена в листинге 4.4.

Позиция на доске снова представлена в виде списка координат Y. Основным отношением в этой программе является следующее отношение:

```
sol( Ylist, Dx, Dy, Du, Dv)
```

которое конкретизирует координаты Y ферзей (в списке Ylist) при условии, что эти ферзи помещены на подряд идущих вертикальных рядах, номера которых взяты из области определения Dx. Все координаты Y и все соответствующие координаты U и V берутся из списков Dy, Du и Dv. Процедура верхнего уровня, solution, может быть вызвана с помощью следующего вопроса:

```
?- solution(S).
```

Ввод этого вопроса влечет за собой вызов отношения sol с полными областями определения, которые соответствуют пространству состояний задачи с восемью ферзями.

Листинг 4.4. Программа 3 для решения задачи с восемью ферзями

```
% solution( Ylist), если
%   Ylist - список координат Y восьми ненападающих ферзей

solution( Ylist) :-
    sol( Ylist,                                     % Координаты Y ферзей
        [1,2,3,4,5,6,7,8],                         % Область определения координат X
        [1,2,3,4,5,6,7,8],                         % Область определения координат Y
        [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],    % Восходящие диагонали
        [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16] ).  % Нисходящие диагонали

sol( [], [], Dy, Du, Dv).

sol( [Y | Ylist], [X | Dx1], Dy, Du, Dv) :-
    del( Y, Dy, Dyl),                               % Выбор координаты Y
    U is X-Y,                                       % Соответствующая восходящая диагональ
    del( U, Du, Dul),                               % Удаление данных о ней
    V is X+Y,                                       % Соответствующая нисходящая диагональ
    del( V, Dv, Dvl),                               % Удаление данных о ней
    sol( Ylist, Dx1, Dyl, Dul, Dvl).               % Использование оставшихся значений

del( Item, [Item | List], List).

del( Item, [First | List], [First | List1] ) :-
    del( Item, List, List1).
```

Процедура sol является универсальной в том смысле, что она может использоваться для решения задачи с N ферзями (на шахматной доске с размерами N×N). Для этого достаточно только правильно задать области определения Dx, Dy и т.д.

Целесообразно механизировать выработку значений этих областей определения. Для этого требуется процедура

```
gen( N1, N2, List)
```

которая после получения двух заданных целых чисел, N1 и N2, вырабатывает следующий список:

```
List = [ N1, N1 + 1, N1 + 2, ..., N2 - 1, N2]
```

Такая процедура приведена ниже.

```
gen( N, N, [N]).
gen( N1, N2, [N1 | List]) :-
    N1 < N2, M is N1 + 1,
    gen( M, N2, List).
```

Отношение верхнего уровня, solution, необходимо обобщить соответствующим образом, определив его в форме:

```
solution( N, S)
```

где N — размер доски и S — решение, представленное в виде списка координат Y для N ферзей. Обобщенное отношение `solution` выглядит следующим образом:

```
solution( N, S) :-  
    gen( 1, N, Dxy), % Dxy - область определения для X и Y  
    Nu1 is 1 - N, Nu2 is N - 1,  
    gen(Nu1, Nu2, Du),  
    Nv2 is N + N,  
    gen( 2, Nv2, Dv),  
    sol( S, Dxy, Dxy, Du, Dv).
```

Например, решение задачи с 12 ферзями может быть получено с помощью следующего вопроса:

```
?- solution( 12, S).  
S = [1,3,5,8,10,12,6,11,2,7,9,4]
```

4.5.4. Заключительные замечания

Приведенные выше три решения задачи с восемью ферзями показывают, что к одной и той же проблеме часто можно подходить с разных сторон. Кроме того, в этих решениях использовались различные представления данных. Иногда такое представление было более экономичным, а иногда — более явным и частично избыточным. Недостатком более экономичного представления является то, что при его использовании всегда требуется вновь восстанавливать изъятую из него информацию, когда в ней возникает необходимость.

В некоторых случаях ключом к успешному решению задачи явилось ее обобщение. Как это ни парадоксально, но при рассмотрении более общей проблемы решение часто становится проще. Такой принцип обобщения является своего рода стандартным приемом, который может часто применяться на практике.

Из этих трех программ третья является наиболее ярким примером того, как подойти к решению общей проблемы формирования структуры из заданного множества элементов с учетом ограничений.

Возникает резонный вопрос о том, какая из этих трех программ является наиболее эффективной. С этой точки зрения программа 2 обладает намного более низкими характеристиками по сравнению с двумя другими программами, которые примерно равнозначны друг другу. Причина этого состоит в том, что в программе 2, основанной на использовании перестановок, формируется полный набор перестановок, а две другие программы обладают способностью распознавать и отбрасывать небезопасные перестановки еще тогда, когда они сформированы лишь частично. В программе 3 исключены некоторые арифметические вычисления, поскольку необходимость в них, по сути, отсутствует благодаря избыточному представлению координат на доске, используемому в данной программе.

Упражнение

4.7. Предположим, что клетки шахматной доски представлены парами их координат в форме X/Y , где X и Y находятся в пределах 1–8.

- а) Определите отношение `jump( Square1, Square2)`, соответствующее ходу конем на шахматной доске. Предположим, что переменная `Square1` всегда конкретизируется значениями координат некоторой клетки, а `Square2` может оставаться не конкретизированной, например, следующим образом:

```
?- jump( 1/1, S).  
S = 3/2;  
S = 2/3;  
но
```

- б) Определите отношение `knightpath( Path)`, где `Path` — список клеток, который представляет один из допустимых путей прохождения коня по пустой шахматной доске.
- в) Используя отношение `knightpath`, запишите вопрос для поиска любого пути движения коня, состоящего из четырех ходов от клетки 2/1 до противоположного края доски ($Y = 8$), который проходит через клетку 5/4 после второго хода.

Резюме

Примеры, приведенные в этой главе, демонстрируют некоторые перечисленные ниже важные особенности и характерные свойства программирования на языке Prolog.

- Любая база данных может быть естественным образом представлена как *множество фактов* Prolog.
- *Механизмы запроса и согласования* системы Prolog могут разнообразными способами применяться для выборки структурированной информации из базы данных. Кроме того, могут быть легко определены вспомогательные процедуры для дальнейшего упрощения взаимодействия с определенной базой данных.
- *Абстрагирование данных* может рассматриваться как метод программирования, который позволяет упростить использование сложных структур данных и вносит свой вклад в обеспечение создания более понятных программ. В языке Prolog можно легко реализовать фундаментальные принципы абстрагирования данных.
- Такие абстрактные математические конструкции, как *конечные автоматы*, часто можно непосредственно преобразовать в исполняемые определения Prolog.
- Как показывает пример задачи с восемью ферзями, к решению одной и той же проблемы можно подойти разными способами, определяя различные представления этой проблемы. *Введение избыточности* в представление часто позволяет уменьшить объем вычислений. В этом проявляется один из компромиссов между требованиями к пространству и времени.
- Важным шагом в поиске решения часто становится *обобщение проблемы*. Как это ни парадоксально, но при рассмотрении более общей проблемы формулировка решения может упроститься.

Управление перебором с возвратами

В этой главе...

5.1. Предотвращение перебора с возвратами	121
5.2. Примеры использования оператора отсечения	125
5.3. Отрицание как недостижение цели	129
5.4. Проблемы, связанные с использованием отрицания и оператора отсечения	132

Как было описано выше, программист может управлять выполнением программы, изменяя порядок следования предложений и целей, а в этой главе рассматривается еще одно средство управления программой, называемое *оператором отсечения*, которое позволяет предотвратить перебор с возвратами. Кроме того, оператор отсечения расширяет выразительные возможности языка Prolog и предоставляет возможность определить одну из разновидностей отрицания, называемого *отрицанием вследствие недостижения цели* и связанного с *предположением о замкнутости мира*.

5.1. Предотвращение перебора с возвратами

Система Prolog автоматически выполняет перебор с возвратами, если это необходимо для достижения цели. Автоматический перебор с возвратами представляет собой полезный принцип программирования, поскольку он освобождает программиста от необходимости явно обеспечивать в программе перебор с возвратами. С другой стороны, неуправляемый перебор с возвратами может вызвать снижение эффективности программы. Поэтому иногда возникает необходимость управлять перебором с возвратами или даже предотвращать его. Такую задачу в языке Prolog можно выполнить с помощью *оператора отсечения*.

Вначале рассмотрим в качестве примера поведение простой программы, выполнение которой связано с некоторым ненужным перебором с возвратами, и определим те моменты, в которых перебор с возвратами бесполезен и приводит к снижению эффективности.

Рассмотрим двухступенчатую функцию, показанную на рис. 5.1. Соответствующую зависимость между X и Y можно определить с помощью приведенных ниже правил.

Правило 1. Если $X < 3$, то $Y = 0$.

Правило 2. Если $3 \leq X$ и $X < 6$, то $Y = 2$.

Правило 3. Если $6 \leq X$, то $Y = 4$.

Эти правила можно записать на языке Prolog в виде бинарного отношения

$f(X, Y)$

следующим образом:

```
f( X, 0) :- X < 3.           % Правило 1
f( X, 2) :- 3 <= X, X < 6.   % Правило 2
f( X, 4) :- 6 <= X.         % Правило 3
```

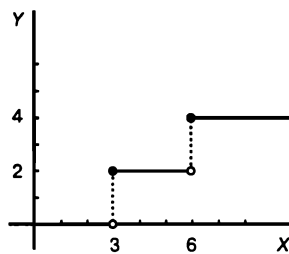


Рис. 5.1. Двухступенчатая функция

В данной программе, безусловно, предполагается, что перед вызовом на выполнение отношения $f(X, Y)$ переменная X уже конкретизирована значением какого-либо числа, поскольку это требуется для операций сравнения.

Проведем два эксперимента с этой программой. Каждый эксперимент выявит некоторые источники неэффективности в программе, и мы устраним каждый из этих источников последовательно с помощью механизма отсечения.

5.1.1. Эксперимент 1

Проанализируем, что произойдет при выполнении следующего вопроса:

?- $f(1, Y), 2 < Y$.

При выполнении первой цели, $f(1, Y)$, переменная Y конкретизируется значением 0. Поэтому вторая цель принимает следующий вид:

$2 < 0$

Данная цель не достигается, поэтому не достигается и весь список целей. Причина этого очевидна, но, прежде чем принять, что данный список целей не достигим, система Prolog проверяет с помощью перебора с возвратами еще два бесперспективных варианта. Подробная трассировка этого процесса показана на рис. 5.2.

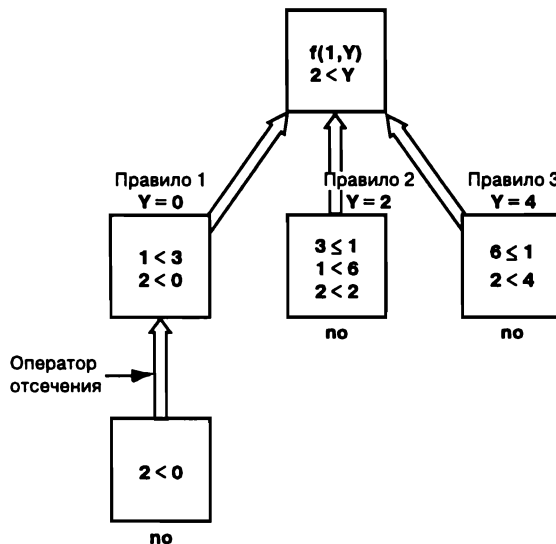


Рис. 5.2. Трассировка выполнения, при которой в точке, обозначенной как "Оператор отсечения", уже известно, что цели, определенные правилами 2 и 3, не достижимы

Приведенные выше три правила, касающиеся функциональной зависимости f , являются взаимоисключающими, поэтому может быть достигнуто не больше одной заданной в них цели. Но система Prolog, в отличие от программиста, не имеет информации о том, что после достижения цели, заданной одним правилом, нет смысла пытаться использовать другие правила, поскольку попытка их достижения неизбежно окончится неудачей. В примере, приведенном на рис. 5.2, известно, что цель правила 1 будет достигнута в точке, обозначенной как "Оператор отсечения". В этот момент необходимо явно сообщить системе Prolog, что не нужно выполнять перебор с возвратами, чтобы предотвратить осуществление бесполезных действий. Эту задачу можно решить с использованием механизма отсечения. Оператор отсечения записывается как восклицательный знак (!) и вставляется между целями как своего рода *псевдоцель*. Рассматриваемая программа, переформулированная с использованием операторов отсечения, принимает вид

```
f( X, 0) :- X < 3, !.  
f( X, 2) :- 3 =< X, X < 6, !.  
f( X, 4) :- 6 =< X.
```

Теперь символ ! предотвращает перебор с возвратами в тех точках, в которых он появляется в программе. Если после этого будет задан следующий вопрос:

```
?- f( 1, Y), 2 < Y.
```

система Prolog сформирует такую же левую ветвь трассировки выполнения, как показано на рис. 5.2. Выполнение данной ветви окончится неудачей при обработке цели $2 < 0$. Затем система Prolog предпринимает попытку осуществить перебор с возвратами, но не сможет пройти точку в программе, обозначенную символом !, поэтому альтернативные ветви, которые соответствуют правилам 2 и 3, так и не будут сформированы.

Новая программа, составленная с использованием операторов отсечения, в целом является более эффективной, чем первоначальная версия, без этих операторов. Если выполнение программы должно закончиться неудачей, новая программа в целом распознает это быстрее, чем первоначальная.

На основании этого можно сделать вывод, что введение в рассматриваемую программу операторов отсечения позволило повысить ее эффективность. Но если в данном примере будут удалены операторы отсечения, программа все равно выработает такие же результаты; просто она, возможно, будет затрачивать больше времени. В данном случае введение операторов отсечения привело лишь к изменению процедурного значения программы и не повлияло на результаты ее работы. Тем не менее ниже будет показано, что использование операторов отсечения способно также повлиять и на результаты.

5.1.2. Эксперимент 2

Теперь проведем второй эксперимент, со второй версией рассматриваемой программы. Предположим, что задан следующий вопрос:

```
?- f( 7, Y).  
Y = 4
```

Проанализируем, что при этом произошло. Была предпринята попытка проверить все три правила, и только после этого был получен ответ. Таким образом, выработана приведенная ниже последовательность целей.

- **Попытка применить правило 1.** Цель $7 < 3$ не достигается; выполнить возврат и попытаться применить правило 2 (оператор отсечения не был применен).
- **Попытка применить правило 2.** Цель $3 \leq 7$ достигается, но затем не достигается цель $7 < 6$; выполнить возврат и попытаться применить правило 3 (оператор отсечения не был применен).
- **Попытка применить правило 3.** Цель $6 \leq 7$ достигается.

Эта трассировка выполнения позволяет обнаружить еще один источник неэффективности. Вначале было установлено, что выражение $X < 3$ не является истинным (цель $7 < 3$ не достижима). Следующей целью является $3 \leq X$ (цель $3 \leq 7$ достижима). Но нам известно, что после неудачного завершения первой проверки вторая проверка обязательно будет успешной, поскольку она является отрицанием первой. Следовательно, вторая проверка является избыточной и соответствующая цель может быть опущена. Справедливым является также аналогичное утверждение в отношении цели $6 \leq X$ в правиле 3. Эти рассуждения приводят к созданию приведенной ниже более лаконичной формулировки трех правил.

Если $X < 3$, то $Y = 0$,
иначе, если $X < 6$, то $Y = 2$,
иначе $Y = 4$.

Теперь мы можем удалить из программы условия, которые гарантированно будут истинными при каждом их выполнении. Это позволяет получить третью версию данной программы:

```
f( X, 0) :- X < 3, !.  
f( X, 2) :- X < 6, !.  
f( X, 4).
```

Эта программа вырабатывает такие же результаты, что и первоначальная версия, но является более эффективной по сравнению с обеими предыдущими версиями. Но что произойдет, если операторы отсеечения будут удалены именно в этой версии? Программа примет следующий вид:

```
f( X, 0) :- X < 3.  
f( X, 2) :- X < 6.  
f( X, 4).
```

Такая программа, в отличие от предыдущих, способна вырабатывать несколько решений, причем некоторые из них являются неправильными, как, например, при получении ответов на следующий вопрос:

```
?- f( 1, Y).  
Y = 0;  
Y = 2;  
Y = 4;  
по
```

Важно отметить, что, в отличие от второй версии данной программы, на этот раз операторы отсеечения не только влияют на процедурное поведение, но и изменяют результаты программы.

Более точное значение механизма отсеечения описано ниже.

Назовем *родительской целью* ту цель, которая согласована с головой предложения, содержащего оператор отсеечения. Если в качестве цели обнаружен оператор отсеечения, эта цель немедленно достигается, но приводит к фиксации в системе всех результатов выбора, полученных с того момента, как была вызвана родительская цель, и до того момента, как встретился оператор отсеечения. Все оставшиеся альтернативы между родительской целью и оператором отсеечения отбрасываются.

Чтобы пояснить это определение, рассмотрим некоторое предложение, заданное в следующей форме:

$H :- B_1, B_2, \dots, B_m, !, \dots, B_n.$

Предположим, что это предложение вызывается целью G , которая согласована с H . В этом случае G представляет собой родительскую цель. Ко времени обнаружения оператора отсеечения система уже нашла некоторое решение, соответствующее целям $B_1, \dots, B_m$. После выполнения оператора отсеечения это (текущее) решение $B_1, \dots, B_m$ замораживается и все возможные оставшиеся альтернативы отбрасываются. Кроме

того, цель G теперь становится зафиксированной на данном предложении: все попытки согласовать цель G с головой какого-то другого предложения исключаются.

Применим эти правила к следующему примеру:

C :- P, Q, R, !, S, T, U.

C :- V.

A :- B, C, D.

?- A.

Здесь A, B, C, D, P и т.д. имеют синтаксис термов. Оператор отсечения повлияет на выполнение цели C, как показано на рис. 5.3. Перебор с возвратами будет возможен в пределах списка целей P, Q, R, но как только будет достигнут оператор отсечения, все альтернативные решения в списке целей P, Q, R подавляются. Альтернативное предложение, касающееся термина C,

C :- V.

также отбрасывается. Но перебор с возвратами в пределах списка целей S, T, U все еще возможен. Родительской целью в предложении, содержащем оператор отсечения, является цель C в следующем предложении:

A :- B, C, D.

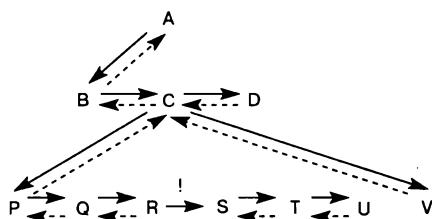


Рис. 5.3. Влияние оператора отсечения на выполнение программы. Начиная с терма A, сплошные стрелки обозначают последовательность вызовов, а пунктирные стрелки — перебор с возвратами. Между термами R и S возможно только “одностороннее движение”

Поэтому оператор отсечения повлияет лишь на выполнение цели C. С другой стороны, он останется “невидимым” с позиций цели A. Таким образом, автоматический перебор с возвратами в пределах списка целей B, C, D остается активным, несмотря на наличие оператора отсечения в предложении, используемом для достижения цели C.

5.2. Примеры использования оператора отсечения

5.2.1. Вычисление максимума

Процедуру поиска наибольшего из двух чисел можно запрограммировать как следующее отношение:

max(X, Y, Max)

где Max = X, если X больше или равен Y, и Max равен Y, если X меньше Y. Это соответствует следующим двум предложениям:

max(X, Y, X) :- X >= Y.

max(X, Y, Y) :- X < Y.

Такие два правила являются взаимоисключающими. Если цель первого правила достигается, то попытка достичь цели второго правила оканчивается неудачей. Если же оканчивается неудачей попытка достичь первой цели, то вторая должна быть достигнута успешно. Поэтому возможна приведенная ниже более лаконичная формулировка, с использованием выражения "иначе".

```
Если X >= Y, то Max = X,  
иначе Max = Y.
```

Эту формулировку можно записать на языке Prolog с использованием оператора отсечения следующим образом:

```
max( X, Y, X) :- X >= Y, !.  
max( X, Y, Y).
```

Следует отметить, что при использовании этой процедуры необходимо соблюдать осторожность. Она является безопасной, если в цели `max( X, Y, Max)` параметр `Max` не конкретизирован. Возможная проблема проиллюстрирована в приведенном ниже примере неправильного применения этой процедуры.

```
?- max( 3, 1, 1).  
yes
```

Указанное ограничение можно преодолеть с использованием следующей формулировки отношения `max`:

```
max( X, Y, Max) :-  
  X >= Y, !, Max = X  
  ;  
  Max = Y.
```

5.2.2. Проверка принадлежности к списку, при которой возможно только одно решение

Выше в этой книге для определения того, входит ли элемент `X` в состав списка `L`, использовалось следующее отношение:

```
member( X, L)
```

Для определения этого отношения применялась такая программа:

```
member( X, [X | L] ).  
member( X, [_ | L] ) :- member( X, L).
```

Эта программа является недетерминированной: если в списке имеется несколько элементов `X`, программа не только определяет наличие элемента `X`, но и последовательно находит все эти элементы. Теперь преобразуем `member` в детерминированную процедуру, которая находит только первое вхождение элемента. Для этого не требуется вносить существенные изменения; достаточно лишь предотвратить перебор с возвратами сразу после нахождения элемента `X`, который происходит после успешного выполнения первого предложения. Модифицированная программа приведена ниже.

```
member( X, [X | L] ) :- !.  
member( X, [_ | L] ) :- member( X, L).
```

Эта программа вырабатывает только одно решение, например:

```
?- member( X, [a,b,c]).  
X = a;  
no
```

5.2.3. Добавление элемента к списку без дублирования

Часто возникает необходимость добавить элемент `X` к списку `L` таким образом, чтобы добавление `X` происходило лишь в том случае, если этот элемент `X` пока еще отсутствует в списке `L`. А если `X` уже имеется в `L`, то `L` должен оставаться неизмен-

ным, поскольку в этом списке нежелательно иметь лишние дубликаты. Такое отношение `add` имеет следующие три параметра:

```
add( X, L, L1)
```

где `X` — добавляемый элемент, `L` — список, в который должен быть добавлен `X`, и `L1` — новый, результирующий список. Соответствующее правило добавления можно сформулировать следующим образом:

Если `X` входит в состав списка `L`, то `L1 = L`,

в противном случае `L1` равен `L` со вставленным элементом `X`.

Проще всего можно вставить элемент `X` перед списком `L` таким образом, чтобы `X` стал головой списка `L1`. Поэтому рассматриваемое отношение можно запрограммировать следующим образом:

```
add( X, L, L) :- member( X, L), !.  
add( X, L, [X | L]).
```

Поведение этой процедуры иллюстрируется на следующем примере:

```
?- add( a, [b,c], L).  
L = [a,b,c]  
?- add( X, [b,c], L).  
L = [b,c]  
X = b  
?- add( a, [b,c,X], L).  
L = [b,c,a]  
X = a
```

Аналогично приведенному выше примеру для отношения `max`, отношение `add( X, L1, L2)` предназначено для вызова с неконкретизированной переменной `L2`. В противном случае результат может оказаться непредсказуемым, например, будет успешно выполнен вопрос `add( a, [a], [a,a])`.

Этот пример является весьма поучительным, поскольку невозможно без труда составить программу “добавления без дублирования”, в которой не использовался бы оператор отсеечения или какая-либо иная конструкция, происходящая от оператора отсеечения. В частности, если в приведенной выше программе будет опущен оператор отсеечения, то отношение `add` будет также добавлять дублирующиеся элементы, например, как показано ниже.

```
?- add( a, [a,b,c], L).  
L = [a,b,c];  
L = [a,a,b,c]
```

Поэтому в данном случае оператор отсеечения необходим для формулировки наменченного отношения, а не просто для повышения эффективности. Эта особенность рассматриваемого оператора иллюстрируется также в следующем примере.

5.2.4. Классификация с разбивкой по категориям

Предположим, что рассматривается база данных результатов встреч по теннису, в которых участвовали члены некоторого клуба. Пары соревнующихся игроков не были упорядочены с помощью какого-либо систематического способа (как при проведении турнира), поэтому база данных содержит информацию лишь о том, чем закончились встречи каждого игрока с некоторыми другими игроками. Эти результаты введены в программу после их представления в виде фактов примерно таким образом:

```
beat( tom, jim).  
beat( ann, tom).  
beat( pat, jim).
```

Необходимо определить отношение

```
class( Player, Category)
```

которое позволяет разбить игроков на категории. Предусмотрены только три категории, описанные ниже.

- winner. Победителем является каждый игрок, который победил во всех поединках.
- fighter. Боец — это любой игрок, который выиграл в некоторых встречах, а в некоторых проиграл.
- sportsman. Звание спортсмена-любителя присваивается тому игроку, который уступил во всех поединках.

Например, если доступны лишь приведенные выше результаты, то Энн и Пэт — победители, Том — боец, а Джим — спортсмен. Правило определения бойца можно легко сформулировать следующим образом:

X является бойцом, если
 есть некоторый Y, такой, что X побеждает Y, и
 есть некоторый Z, такой, что Z побеждает X.

Теперь сформулируем правило определения победителя:

X является победителем, если
 X победил некоторого Y и
 X не был побежден никем.

Последняя формулировка содержит отрицание “не”, которое нельзя непосредственно представить с помощью средств Prolog, рассматриваемых до сих пор. Поэтому формулировка отношения winner на первый взгляд кажется немного сложнее. Такая же проблема возникает при анализе отношения sportsman. Эту проблему можно обойти, объединив определение отношения winner с определением отношения fighter и воспользовавшись выражением “в противном случае”. Такая формулировка приведена ниже.

Если X кого-то победил и X был кем-то побежден,
 то X — боец,
 в противном случае, если X кого-то победил,
 то X — победитель,
 в противном случае, если X был кем-то побежден,
 то X — спортсмен.

Эту формулировку можно сразу же перевести на язык Prolog. Взаимное исключение трех альтернативных категорий обозначается операторами отсечения следующим образом:

```
class( X, fighter) :-
    beat( X, _),
    beat( _, X), !.

class( X, winner) :-
    beat( X, _), !.

class( X, sportsman) :-
    beat( _, X).
```

Обратите внимание на то, что оператор отсечения в предложении для категории sportsman не требуется. При использовании подобных процедур с операторами отсечения необходимо соблюдать осторожность. Ниже показано, что при этом может произойти в ином случае.

```
?- class( tom, C).
C = fighter; % Как и следовало ожидать
no
?- class( tom, sportsman).
yes % Вопреки ожиданиям
```

Вызов отношения class является безопасным, если второй параметр не конкретизирован. В противном случае могут быть получены непредсказуемые результаты.

Упражнения

5.1. Рассмотрим следующую программу:

```
P(1).  
P(2) :- !.  
P(3).
```

Напишите все ответы системы Prolog на приведенные ниже вопросы.

а) ?- p(X) .

б) ?- p(X), p(Y) .

в) ?- p(X), !, p(Y) .

5.2. Приведенное ниже отношение классифицирует числа на три категории: положительные, нуль и отрицательные.

```
class( Number, positive) :- Number > 0.  
class( 0, zero).  
class( Number, negative) :- Number < 0.
```

Определите эту процедуру более эффективным способом с использованием оператора отсечения.

5.3. Определите процедуру

```
split( Numbers, Positives, Negatives)
```

которая разбивает список чисел на два списка: положительные (включая нуль) и отрицательные, например, как показано ниже.

```
split( [3,-1,0,5,-2], [3,0,5], [-1,-2] )
```

Предложите две версии: одну с оператором отсечения, а другую без него.

5.3. Отрицание как недостижение цели

Рассмотрим, каким образом может быть представлено на языке Prolog утверждение “Мэри нравятся все животные, кроме змей”. Одну часть этого утверждения можно представить легко: Мэри нравится любой X, если X — животное. На языке Prolog это можно выразить следующим образом:

```
likes( mary, X) :- animal( X).
```

Но нам необходимо исключить змей. Такую задачу можно выполнить, используя другую формулировку, как показано ниже.

Если X — змея, то утверждение ‘Мэри нравится X’ не является истинным.

В противном случае, если X — животное, то Мэри нравится X.

Утверждение “нечто не является истинным” можно выразить на языке Prolog с использованием особой цели, fail, попытка достичь которой всегда оканчивается неудачей, вынуждая тем самым окончиться неудачей попытку достичь родительской цели. Приведенную выше формулировку можно перевести на язык Prolog с применением цели fail следующим образом:

```
likes( mary, X) :-  
    snake( X), !, fail.
```

```
likes( mary, X) :-  
    animal( X).
```

В этой программе первое правило посвящено змеям: если X — змея, то оператор отсечения предотвращает перебор с возвратами (исключая тем самым второе правило), а цель fail вызывает неудачное завершение. Эти два предложения могут быть записаны более компактно в виде одного предложения:

```
likes( mary, X) :-  
    snake( X), !, fail  
    ;  
    animal( X).
```

Такую же идею можно использовать для определения отношения `different( X, Y)`

которое принимает истинное значение, если X и Y являются различными. Но необходимо уточнить рассматриваемую задачу, поскольку слово “различный” можно трактовать несколькими способами, следующим образом:

- X и Y не являются буквально одинаковыми;
- X и Y не согласуются;
- значения арифметических выражений X и Y не равны.

В данном случае выберем вариант, при котором X и Y являются различными, если они не согласуются. Ключом к решению задачи по оформлению этого утверждения на языке Prolog является приведенное ниже рассуждение.

Если X и Y согласуются, вызов отношения `different( X, Y)` оканчивается неудачей, в противном случае вызов отношения `different( X, Y)` завершается успешно.

Снова воспользуемся сочетанием оператора отсечения и цели `fail` следующим образом:

```
different( X, X) :- !, fail.  
different( X, Y).
```

Кроме того, эту программу можно также записать в виде одного предложения:

```
different( X, Y) :-  
    X = Y, !, fail  
    ;  
    true.
```

Применяемая здесь цель `true` всегда достигается.

Эти примеры показывают, что было бы удобно иметь унарный предикат `not`, такой, что вызов

```
not( Goal)
```

возвращал бы истинное значение, если цель `Goal` не является истинной. Определим отношение `not`, как показано ниже.

Если `Goal` достигается, то `not( Goal)` не достигается, в противном случае `not( Goal)` достигается.

Это определение может быть записано на языке Prolog следующим образом:

```
not( P) :-  
    P, !, fail  
    ;  
    true.
```

В дальнейшем предполагается, что `not` — встроенная процедура Prolog, которая действует в соответствии с приведенным здесь определением. Предполагается также, что определен префиксный оператор `not`, поэтому цель

```
not( snake(X))
```

может быть записана также как

```
not snake( X)
```

Некоторые реализации Prolog действительно поддерживают такую систему обозначений. В ином случае мы можем всегда определить отношение `not` самостоятельно. Возможен также вариант, при котором `not Goal` записывается как `\+Goal`. Такое менее наглядное обозначение рекомендовано и в стандарте Prolog по следующей причине: отношение `not`, определенное как недостижение цели (как в данном случае), не полностью соответствует понятию отрицания в математической логике. А если это различие не учитывается в программе, то может стать причиной непредсказуемого поведения программы из-за того, что при использовании отношения `not` не соблюдаются меры предосторожности. Эта тема рассматривается ниже в данной главе.

Тем не менее отношение `not` — полезное средство и может часто с успехом применяться вместо оператора отсечения. Рассматриваемые два примера могут быть переоформлены с применением оператора `not` следующим образом:

```
likes( mary, X) :-  
    animal( X),  
    not snake( X).
```

```
different( X, Y) :-  
    not( X = Y).
```

Безусловно, такие программы выглядят лучше по сравнению с первоначальными формулировками. Они являются более естественными и удобными для чтения.

Кроме того, с использованием `not` может быть переоформлена программа классификации игроков в теннис из предыдущего раздела, в результате чего эта программа будет в большей степени соответствовать первоначальному словесному определению трех категорий:

```
class( X, fighter) :-  
    beat( X, _),  
    beat( _, X).
```

```
class( X, winner) :-  
    beat( X, _),  
    not beat( _, X).
```

```
class( X, sportsman) :-  
    beat( _, X),  
    not beat( X, _).
```

В качестве еще одного примера использования оператора `not` снова рассмотрим программу 1 для решения задачи с восемью ферзями из предыдущей главы (см. листинг 4.2). В ней было определено отношение `no_attack` между одним ферзем и другими ферзями. Это отношение можно также сформулировать как отрицание отношения `attack`. Программа, откорректированная соответствующим образом, приведена в листинге 5.1.

Листинг 5.1. Еще одна программа решения задачи с восемью ферзями

```
.....  
solution( []).  
solution( [X/Y | Others]) :-  
    solution( Others),  
    member( Y, [1,2,3,4,5,6,7,8]), % Обычный предикат member  
    not attacks( X/Y, Others).  
  
attacks( X/Y, Others) :-  
    member( X1/Y1, Others),  
    ( Y1 = Y;  
      Y1 is Y + X1 - X;  
      Y1 is Y - X1 + X ).  
.....
```

Упражнения

5.4. Предположим, что даны два списка, `Candidates` и `RuledOut`, в которых указаны кандидаты в депутаты и лица, выбывшие из предвыборной борьбы. Напишите последовательность целей (с использованием отношений `member` и `not`), которая позволяет с помощью перебора с возвратами найти все элементы в списке `Candidates`, не находящиеся в списке `RuledOut`.

5.5. Определите отношение для вычитания множеств

```
set_difference( Set1, Set2, SetDifference)
```

в котором все три множества представлены в виде списков, например:

```
set_difference( [a,b,c,d], [b,d,e,f], [a,c])
```

5.6. Определите предикат

```
unifiable( List1, Term, List2)
```

Здесь List2 представляет собой список всех элементов списка List1, которые согласуются с термом Term, но не конкретизируются в результате этого согласования, например:

```
?- unifiable( [X, b, t(Y)], t(a), List)
List = [X, t(Y)]
```

Обратите внимание на то, что X и Y должны оставаться неконкретизированными, несмотря на то, что согласование с термом t(a) может вызвать такую конкретизацию. Подсказка: используйте правило `not( Term1 = Term2)`; если цель `Term1 = Term2` достигается, то попытка достичь цели `not( Term1 = Term2)` оканчивается неудачей и в результате этого конкретизация отменяется!

5.4. Проблемы, связанные с использованием отрицания и оператора отсечения

Применение средства отсечения позволяет достичь определенных успехов, но за это приходится платить. Преимущества и недостатки использования оператора отсечения были проиллюстрированы на примерах в предыдущих разделах. Подведем общий итог; вначале для этого рассмотрим преимущества, как описано ниже.

1. Оператор отсечения часто позволяет повысить эффективность программы. Общий замысел здесь состоит в том, что этот оператор дает возможность явно сообщить системе Prolog, что не следует проверять другие альтернативы, поскольку они неизбежно окончатся неудачей.
2. Использование оператора отсечения позволяет определять взаимоисключающие правила, поэтому с его помощью можно представлять правила, имеющие следующую форму:

Если условие P, то заключение Q,
в противном случае заключение R.

Таким образом, оператор отсечения повышает выразительную мощь языка Prolog.

Предпосылки, препятствующие применению оператора отсечения, вытекают из того факта, что при этом возникает опасность утратить ценное соответствие между декларативным и процедурным значениями программы. Если в программе не применяется оператор отсечения, имеется возможность изменять порядок предложений и целей и такая перестановка влияет лишь на эффективность или успешность завершения программы, но не на ее декларативное значение. С другой стороны, изменение порядка следования предложений в программах с операторами отсечения может повлиять на их декларативное значение. Таким образом, после перестановки предложений в такой программе могут быть получены иные результаты, как показано в следующем примере:

```
p :- a, b.
p :- c.
```

Эта программа имеет следующее декларативное значение: p является истинным, если и только если a и b одновременно являются истинными или c является истинным. Такое утверждение может быть записано в виде следующей логической формулы:

```
p <==> (a & b) v c
```

Последовательность этих двух предложений может быть изменена, но при этом декларативное значение останется неизменным. Теперь вставим оператор отсечения следующим образом:

```
p :- a, !, b.
p :- c.
```


После этого программа приобретает такое декларативное значение:

```
p <==> (a & b) v (~a & c)
```

Если теперь порядок следования предложений будет изменен следующим образом на противоположный:

```
p :- c.
```

```
p :- a, !, b.
```

то программа примет такое значение:

```
p <==> c v (a & b)
```

Из этого следует важный вывод, что если применяется средство отсечения, то необходимо уделять больше внимания процедурным аспектам. К сожалению, эта дополнительная сложность приводит к повышению вероятности ошибок программирования.

В примерах, приведенных в предыдущих разделах, было показано, что иногда удаление оператора отсечения из программы приводит к изменению декларативного значения программы. Но есть и такие случаи, в которых оператор отсечения не оказывает никакого влияния на декларативное значение. Использование операторов отсечения последнего типа в меньшей степени способствует возникновению ошибок в программе, и поэтому операторы отсечения такого типа иногда называют *зелеными операторами отсечения*. С точки зрения удобства программ для чтения зеленые операторы отсечения являются “безвредными”, и их применение вполне допустимо. При чтении программы зеленые операторы отсечения могут игнорироваться.

В отличие от этого, операторы отсечения, влияющие на декларативное значение, принято называть *красными операторами отсечения*. Красные операторы отсечения представляют собой такие операторы, в результате введения которых программы становятся более сложными для понимания, и они должны использоваться с особой осторожностью.

Оператор отсечения часто используется в сочетании со специальной целью, *fail*. В частности, мы определили отрицание цели (*not*) как неудачу в достижении цели. Отрицание, определенное таким образом, представляет собой особый, более ограниченный способ использования оператора отсечения. По соображениям наглядности мы предпочитаем использовать оператор *not* вместо сочетания “оператор отсечения — цель *fail*” (если это возможно), поскольку интуитивно отрицание воспринимается как более понятная операция, чем сочетание “оператор отсечения — цель *fail*”.

Следует отметить, что использование оператора *not* также может привести к возникновению проблем, поэтому и его следует использовать с осторожностью. Как уже было сказано, проблема, связанная с оператором *not*, состоит в том, что он не полностью соответствует понятию отрицания в математике. Например, если системе Prolog будет задан следующий вопрос:

```
?- not human( mary ).
```

то она, вероятно, ответит “yes”. Но этот ответ не следует понимать таким образом, будто Prolog отвечает, что Мэри не человек. В действительности этот ответ Prolog означает, что в программе нет достаточной информации, чтобы определить, является ли Мэри человеком. Такая ситуация возникает в связи с тем, что при обработке цели *not* система Prolog не пытается доказать эту цель непосредственно. Вместо этого она пытается доказать обратное, а если обратное не может быть доказано, Prolog предполагает, что достигается цель *not*.

Такой ход рассуждений основан на так называемом *предположении о замкнутости мира*. В соответствии с этим предположением мир замкнут, в том смысле, что все существующее в мире утверждается в программе или может быть выведено из программы. Таким образом, если какие-то факты не заданы в программе (или не могут быть выведены на основе имеющейся в ней информации), то они не являются истинными, а, следовательно, истинным является их отрицание. Такая особенность функционирования системы Prolog говорит о том, что ее эксплуатация требует осо-

бой осторожности, поскольку пользователи обычно не руководствуются предположением о замкнутости мира. Поэтому если в программу не введено явно предложение `human( mary )`.

то это не означает, что мы не считаем Мэри человеком, но ведь система об этом не знает!

Чтобы дополнительно изучить вопрос, связанный с тем, почему применение оператора `not` требует осторожности, рассмотрим следующий пример, касающийся ресторанов:

```
good_standard( jeanluis ).
```

```
expensive( jeanluis ).
```

```
good_standard( francesco ).
```

```
reasonable( Restaurant ) :- % Ресторан с приемлемыми ценами
```

```
not expensive( Restaurant ). % - это НЕ дорогой ресторан!
```

Если после ввода этой программы будет задан вопрос

```
?- good_standard( X ), reasonable( X ).
```

для определения того, какие рестораны характеризуются высоким уровнем обслуживания и приемлемыми ценами, то система Prolog ответит

```
X = francesco
```

Если же будет задан, на первый взгляд, такой же вопрос

```
?- reasonable( X ), good_standard( X ).
```

то система Prolog ответит

```
no
```

Рекомендуем читателю выполнить трассировку программы, чтобы понять причины получения разных ответов. Ключевая разница между обоими вопросами состоит в том, что в первом случае при выполнении термина `reasonable( X )` переменная `X` уже конкретизирована, а во втором случае — еще не конкретизирована. В качестве общей рекомендации можно указать следующее: оператор `not Goal` выполняется безопасно, если переменные в цели `Goal` уже конкретизированы ко времени вызова цели `not Goal`. В противном случае могут быть получены непредсказуемые результаты по причинам, описанным ниже.

Проблема, связанная с неконкретизированными отрицаемыми целями, является результатом неблагоприятного изменения квантификации переменных при использовании отрицания как недостижения цели. При обычной интерпретации в системе Prolog вопрос

```
?- expensive( X ).
```

означает: “Существует ли такой `X`, что цель `expensive( X )` является истинной? Если да, то каков этот `X`?” Поэтому к `X` применяется квантор существования. В соответствии с этим система Prolog отвечает, что `X = jeanluis`. Но вопрос

```
?- not expensive( X ).
```

не интерпретируется так: “Существует ли такой `X`, что достигается цель `not expensive(X)`?” Если бы это было действительно так, то можно было рассчитывать на получение ответа `X = francesco`. Но Prolog отвечает “no”, поскольку при использовании отрицания как недостижения цели квантор существования заменяется квантором всеобщности. Вопрос `not expensive( X )` интерпретируется следующим образом:

```
not( существует X, такой, что достигается цель expensive(X) )
```

Это равносильно следующему утверждению:

Для всех `X` утверждение `expensive(X)` является ложным

В этой главе подробно рассматриваются проблемы, связанные с использованием оператора отсечения, которые также косвенно касаются и оператора `not`. Автор поставил перед собой задачу предупредить читателей о необходимости соблюдать осторожность при работе с оператором отсечения, а не полностью отказаться от его применения. Оператор отсечения является полезным и часто необходимым. Кроме того, сложности, аналогичные тем, которые связаны с использованием оператора отсечения в языке Prolog, часто наблюдаются и при разработке программ на других языках.

Резюме

- *Оператор отсечения* позволяет предотвратить перебор с возвратами. Он используется, во-первых, для повышения эффективности программ, а во-вторых, для усиления выразительной мощи языка.
- *Повышение эффективности* достигается благодаря тому, что системе Prolog можно явно передать указание (с помощью оператора отсечения), что не нужно проверять альтернативные цели, о которых заранее известно, что попытка их достижения окончится неудачей.
- Оператор отсечения позволяет сформулировать взаимоисключающие заключения с помощью правил, имеющих следующую форму:
если Условие, то Заключение1, в противном случае Заключение2
- Оператор отсечения позволяет ввести в программу конструкцию *отрицания как недостижения цели*; оператор `not Goal` определяется как неудача при достижении цели `Goal`.
- В программе иногда возникает необходимость в использовании двух *специальных целей*, `true` и `fail`; первая всегда достигается, а последняя никогда не достигается.
- Имеются также некоторые предпосылки, препятствующие использованию оператора отсечения, связанные с тем, что вставка оператора отсечения может нарушить соответствие между декларативным и процедурным значениями программы. Поэтому признаком хорошего стиля программирования является соблюдение осторожности при использовании оператора отсечения и отказ от его применения без достаточных оснований.
- Определение оператора `not` как недостижения цели не полностью соответствует понятию отрицания в математической логике. Поэтому использование оператора `not` также требует особой осторожности.

Дополнительные источники информации

Проведение различия между *зелеными операторами отсечения* и *красными операторами отсечения* было предложено в [160]. В [87] предложен иной оператор отрицания для языка Prolog, более приемлемый с точки зрения математики, но требующий больших вычислительных издержек.

Ввод и вывод

В этой главе...

6.1. Обмен данными с файлами	136
6.2. Обработка файлов, состоящих из термов	139
6.3. Манипулирование символами	143
6.4. Формирование и анализ атомов	144
6.5. Чтение программ	146

В этой главе рассматриваются некоторые встроенные средства, предназначенные для чтения данных из компьютерных файлов и вывода данных в файлы. Соответствующие процедуры могут также использоваться для форматирования объектов данных в программе в целях создания желаемого внешнего представления этих объектов. Кроме того, рассматриваются средства чтения программ, а также формирования и анализа атомов.

6.1. Обмен данными с файлами

Метод взаимодействия пользователя и программы, который применялся до сих пор, предусматривал ввод пользователем вопросов к программе и получение ответов программы в виде результатов конкретизации переменных. Такой метод обмена данными является простым и достаточным для обеспечения ввода и вывода информации. Тем не менее он часто является не совсем приемлемым, поскольку не обладает достаточной гибкостью. Поэтому этот основной метод обмена данными необходимо дополнить в следующих областях:

- ввод данных в форматах, отличных от вопросов, например в форме английских предложений;
- вывод информации в любом требуемом формате;
- ввод и вывод в любой компьютерный файл или на любое устройство, а не только на пользовательский терминал.

Состав встроенных предикатов, предназначенных для использования в качестве подобных расширений, зависит от реализации Prolog. В данной главе рассматривается простой и удобный набор соответствующих предикатов, который входит в состав многих реализаций Prolog. Тем не менее необходимо всегда обращаться к руководству по соответствующей реализации Prolog для ознакомления с подробными сведениями и характерными особенностями тех или иных средств. Во многих реализациях Prolog предусмотрены различные дополнительные средства, не описанные в данной главе. К ним относятся, в частности, средства работы с окнами, графические примитивы для рисования на экране, средства ввода информации с помощью мыши и т.д.

Вначале рассматривается тема, связанная с перенаправлением ввода и вывода информации в файлы, а затем показано, как можно обеспечить ввод и вывод данных в различных формах.

На рис. 6.1 показана общая ситуация, в которой программа Prolog осуществляет обмен данными с несколькими файлами. В принципе, программа может считывать данные из нескольких входных файлов, называемых также *входными потоками*, и выводить данные в несколько выходных файлов, которые называются *выходными потоками*. Данные, поступающие с пользовательского терминала, рассматриваются как еще один входной поток. Аналогичным образом, данные, выводимые на терминал, представляют собой один из выходных потоков. Эти потоки не фиксируются на жестком диске, но связаны с информационными структурами, аналогичными файлам (которые принято называть *псевдофайлами*) и известными под именем *user*. Имена других файлов могут быть выбраны программистом с учетом правил именования файлов в используемой компьютерной системе.

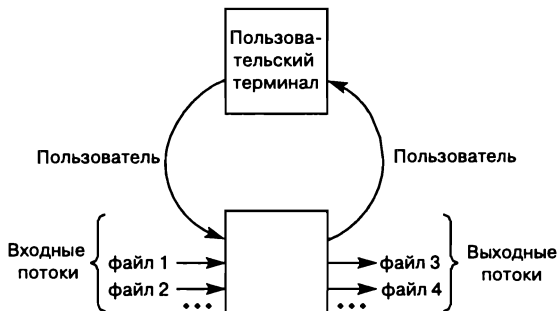


Рис. 6.1. Взаимодействие программы Prolog и нескольких файлов

В любой момент времени в ходе выполнения программы Prolog “активны” только два файла: один для ввода, а другой для вывода. Эти два файла, соответственно, называются *текущим входным потоком* и *текущим выходным потоком*. В начале выполнения программы эти два потока соответствуют пользовательскому терминалу. Текущий входной поток можно перевести в другой файл, *Filename*, с помощью следующей цели:

```
see( Filename)
```

Эта цель достигается (если только не обнаружится какая-либо ошибка, связанная с переменной *Filename*) и вызывает в качестве побочного эффекта то, что ввод переключается с предыдущего входного потока на файл *Filename*. Поэтому типичным примером использования предиката *see* является приведенная ниже последовательность целей, которая обеспечивает чтение определенных данных из файла *file1*, а затем повторное переключение на терминал.

```
...
see( file1),
read_from_file( Information),
see( user),
...
```

Текущий выходной поток можно изменить с помощью цели, заданной в следующей форме:

```
tell( Filename)
```

Ниже приведена последовательность целей для вывода некоторой информации в файл *file3*, а затем перенаправления последующего вывода снова на терминал.

```
...
tell( file3),
write_on_file( Information),
tell( user),
...
```

Цель

seen

закрывает текущий входной файл, а цель

told

закрывает текущий выходной файл.

В данной главе предполагается, что файлы могут обрабатываться только последовательно, хотя во многих реализациях Prolog предусмотрена также обработка файлов с произвольным доступом. Последовательные файлы обрабатываются таким же образом, как и потоки на терминале. Каждый запрос на чтение определенной информации из входного файла вызывает чтение данных с текущей позиции в текущем входном потоке. После чтения текущая позиция, безусловно, перемещается на следующий непрочитанный элемент. Поэтому выполнение следующего запроса на чтение начинается с чтения данных в этой новой текущей позиции. Если запрос на чтение выполняется в конце файла, то информация, возвращаемая таким запросом, представляет собой атом `end_of_file`.

Операции записи выполняются таким же образом; каждый запрос на вывод информации приводит к добавлению этой информации к концу текущего выходного потока. Возможность возвращаться назад и перезаписывать часть файла не предусмотрена.

В этой главе рассматриваются только так называемые *текстовые файлы*, т.е. файлы, состоящие из *символов*. Символы представляют собой буквы, цифры и специальные символы. Некоторые из специальных символов называются *непечатаемыми*, поскольку после вывода их на терминал они не появляются на экране. Но эти символы могут оказывать иной эффект, например изменять интервалы между столбцами и строками на экране.

Файлы могут обрабатываться в языке Prolog двумя основными способами, в зависимости от формы представления в них информации. Один из способов состоит в том, что основным элементом файла является символ. В соответствии с этим один запрос на ввод или вывод вызывает чтение или запись единственного символа. В этой главе предполагается, что для этого служат встроенные предикаты `get`, `get0` и `put`.

Еще один способ обработки файлов состоит в том, что в качестве основных структурных блоков файла рассматриваются более крупные информационные единицы. Вполне естественно, что в качестве такой более крупной единицы принят терм Prolog. Поэтому при каждом запросе ввода-вывода такого типа из текущего входного потока или в текущий выходной поток передается, соответственно, целый терм. Предикатами для передачи термов являются `read` и `write`. Разумеется, в этом случае информация в файле должна находиться в форме, совместимой с синтаксисом термов.

Безусловно, выбор той или иной организации файла зависит от рассматриваемой задачи. Если спецификация задачи позволяет представить информацию естественным образом в синтаксической структуре термов, то следует использовать файл, состоящий из термов. В таком случае появляется возможность передавать целый осмысленный фрагмент информации с помощью одного запроса. С другой стороны, есть такие задачи, характер которых определяет необходимость использования некоторой другой организации файлов. В качестве примера можно назвать обработку предложений на естественном языке, например, для организации диалога на английском языке между системой и пользователем. В подобных случаях файлы должны рассматриваться как последовательности символов, которые нельзя непосредственно интерпретировать как термы.

6.2. Обработка файлов, состоящих из термов

6.2.1. Предикаты read и write

Встроенный предикат `read` используется для чтения термов из текущего входного потока, а цель

```
read( X)
```

вызывает чтение следующего терма, `T`, и согласование этого терма с `X`. Если `X` представляет собой переменную, то в результате `X` становится конкретизированной значением `T`, а если согласование оканчивается неудачей, цель `read( X)` не достигается. Предикат `read` является детерминированным, поэтому в случае неудачи не выполняется перебор с возвратами для ввода другого терма. За каждым термом во входном файле должны следовать точка и пробел (или символ с обозначением конца строки).

Если предикат `read( X)` вызывается на выполнение после достижения конца текущего входного файла, то переменная `X` становится конкретизированной значением атома `end_of_file`.

Встроенный предикат `write` выводит терм, поэтому цель

```
write( X)
```

выводит терм `X` в текущий выходной файл. Терм `X` выводится в такой же стандартной синтаксической форме, в которой Prolog обычно отображает значения переменных. Полезным средством языка Prolog является то, что процедура `write` "умеет" отображать любые термы, независимо от того, насколько они могут быть сложными.

Как правило, в любой реализации Prolog предусмотрены дополнительные встроенные предикаты форматирования вывода, которые вставляют в выходной поток пробелы и символы с обозначением новой строки. Например, цель

```
tab( N)
```

обеспечивает вывод `N` пробелов, а предикат `nl` (который не имеет параметров) обеспечивает вывод символа с обозначением конца строки.

Применение этих процедур иллюстрируется в приведенных ниже примерах.

Предположим, что используется следующая процедура, которая вычисляет куб любого числа:

```
cube( N, C) :-  
    C is N * N * N.
```

Допустим, что необходимо использовать ее для вычисления кубов ряда чисел. Такую задачу можно выполнить с помощью приведенной ниже последовательности вопросов.

```
?- cube(2,X).  
X = 8  
?- cube( 5, Y).  
Y = 125  
?- cube(12,Z).  
Z = 1728
```

Для получения каждого числа в данном случае приходилось вводить соответствующую цель, поэтому модифицируем программу таким образом, чтобы процедура `cube` могла сама считывать данные. Теперь программа будет считывать данные и выводить значения соответствующих им кубов до тех пор, пока не будет считан атом `stop`:

```
cube :-  
    read( X),  
    process( X).  
  
process( stop) :- !.  
process( N) :-
```

```
C is N * N * N,
write( C),
cube.
```

Это — наглядный пример программы, декларативное значение которой сформулировать почти невозможно. Но ее процедурное значение является несложным: чтобы выполнить процедуру `cube`, вначале следует прочитать переменную `X`, а затем ее обработать; если `X = stop`, то работа закончена, в противном случае необходимо вывести куб `X` и рекурсивно вызвать процедуру `cube` для дальнейшей обработки данных. Таблицу кубов чисел можно подготовить с использованием этой новой процедуры следующим образом:

```
?- cube.
2.
8
5.
125
12.
1728
stop.
yes
```

Пользователем на терминале были введены числа 2, 5 и 12; остальные числа выведены программой. Обратите внимание на то, что за каждым числом, введенным пользователем, должна следовать точка, которая обозначает конец термина.

На первый взгляд может показаться, что есть возможность упростить приведенную выше процедуру `cube`. Но следующая попытка упрощения является неправильной:

```
cube :-
  read( stop), !.
cube :-
  read( N),
  C is N * N * N,
  write( C),
  cube.
```

Причину, по которой эта программа не работает должным образом, можно легко обнаружить, выполнив трассировку программы с входными данными, допустим 5. После чтения этого числа цель `read( stop)` не будет достигнута и введенное число безвозвратно потеряется. Следующая цель `read` обеспечит ввод очередного термина. С другой стороны, может оказаться, что сигнал `stop` считан целью `read( N)`, а это в дальнейшем вызовет попытку перемножить нечисловые данные.

Процедура `cube` обеспечивает взаимодействие пользователя и программы. В подобных случаях обычно желательно, чтобы программа перед чтением новых данных с терминала сообщала пользователю, что она готова принять информацию и, возможно, даже показывала, какого рода информацию она ожидает. Такая задача обычно решается путем отправки пользователю перед чтением данных определенного сигнала в виде так называемого *приглашения*. Рассматриваемая процедура `cube` может быть откорректирована соответствующим образом, например, как показано ниже.

```
cube :-
  write( 'Next item, please: '),
  read( X),
  process( X).

process( stop) :- !.
process( N) :-
  C is N * N * N,
  write( 'Cube of '), write( N), write( ' is '),
  write( C), nl,
  cube.
```

В таком случае диалог с новой версией процедуры `cube` может происходить, например, следующим образом:


```
?- cube.
Next item, please: 5.
Cube of 5 is 125
Next item, please: 12.
Cube of 12 is 1728
Next item, please: stop.
yes
```

В зависимости от реализации Prolog, после вывода приглашения может потребоваться дополнительный запрос (скажем, такой как `ttyflush`), который вынуждал бы приглашение фактически появляться на экране перед чтением.

В следующих разделах рассматриваются некоторые типичные примеры операций, связанных с чтением и записью.

6.2.2. Вывод списков на внешнее устройство

Кроме стандартного формата Prolog для списков, предусмотрено несколько других удобных форм для отображения списков, которые являются более предпочтительными в некоторых ситуациях. Например, процедура

```
writelist( L )
```

выводит список `L` таким образом, что каждый элемент `L` записывается на отдельной строке:

```
writelist([]).
writelist( [X | L] ) :-
    write( X ), nl,
    writelist( L ).
```

Если имеется список списков, одной из удобных форм вывода является запись элементов каждого списка на отдельной строке. Для этого можно определить процедуру `writelist2`. Пример ее применения приведен ниже.

```
?- writelist2( [ [a,b,c], [d,e,f], [g,h,i] ] ).
a b c
d e f
g h i
```

Для выполнения этой задачи используется следующая процедура:

```
writelist2([]).
writelist2( [L | LL] ) :-
    doline( L ), nl,
    writelist2( LL ).
```

```
doline([]).
doline( [X | L] ) :-
    write( X ), tab( 1 ),
    doline( L ).
```

Список, состоящий из целых чисел, иногда удобно представить в виде гистограммы. Следующая процедура, `bars`, отображает список в этой форме, при условии, что числа в списке не слишком велики. Пример использования процедуры `bars` приведен ниже.

```
?- bars( [3,4,6,5] ).
***
****
*****
*****
```

Процедура `bars` может быть определена следующим образом:

```
bars([]).
bars([N| L] ) :-
    stars( N ), nl,
    bars( L ).
```

```
stars( N) :-
    N > 0,
    write( *),
    N1 is N - 1,
    stars( N1).
stars( N) :-
    N =< 0.
```

6.2.3. Обработка файла, состоящего из термов

Типичная последовательность целей для обработки целого файла *F* может выглядеть примерно так:

```
..., see( F), processfile, see( user), ...
```

В данном случае `processfile` — это процедура, которая считывает и обрабатывает каждый терм в файле *F* один за другим до тех пор, пока не будет обнаружен конец файла. Типичная схема процедуры `processfile` приведена ниже.

```
processfile :-
    read( Term),                % Чтение, при условии, что Term - не переменная
    process( Term).

process( end_of_file) :- !. % Вся работа выполнена
process( Term) :-
    treat( Term),              % Обработать текущий элемент
    processfile.              % Обработать остальную часть файла
```

В данном случае цель `treat( Term)` представляет любые действия, которые могут быть выполнены с каждым термом. Примером может служить процедура для отображения на терминале каждого терма вместе с его порядковым номером. Назовем эту процедуру `showfile`. Она должна иметь дополнительный параметр для подсчета количества прочитанных термов.

```
showfile( N) :-
    read( Term),
    show( Term, N).

show( end_of_file, _) :- !.
show( Term, N) :-
    write( N), tab( 2), write( Term), nl,
    N1 is N + 1,
    showfile( N1).
```

Упражнения

6.1. Допустим, что *f* — файл, состоящий из термов. Определите процедуру

```
findterm( Term)
```

отображающую на терминале первый терм в файле *f*, который согласуется с термом *Term*.

6.2. Допустим, что *f* — файл, состоящий из термов. Напишите процедуру

```
findallterms( Term)
```

отображающую на терминале все термы в файле *f*, которые согласуются с термом *Term*. Обеспечьте, чтобы *Term* в этом процессе не конкретизировался (поскольку при этом будет исключена возможность его согласования с термами, которые могут в дальнейшем встретиться в файле).

6.3. Манипулирование символами

Символ записывается в текущий выходной поток с помощью следующей цели:

```
put( C)
```

где C — код ASCII (число от 0 до 127) выводимого символа. Например, вопрос

```
?- put( 65), put( 66), put( 67)
```

вызывает вывод следующих данных:

```
ABC
```

где 65 — код ASCII символа “A”, 66 — символа “B” и 67 — символа “C”.

Отдельный символ может быть считан из текущего входного потока с помощью цели

```
get0( C)
```

Эта цель вызывает чтение текущего символа из входного потока и конкретизацию переменной C значением кода ASCII этого символа. Вариантом предиката `get0` является `get`, который используется для чтения *непробельных символов* (символов, отличных от пробела, знака табуляции и т.п.). Поэтому цель

```
get( C)
```

вызывает пропуск всех непечатаемых символов (в частности, пробелов) от текущей позиции ввода во входном потоке вплоть до первого печатаемого символа. После этого, как обычно, считывается данный символ и переменная C конкретизируется значением его кода ASCII.

В качестве примера использования предикатов, передающих отдельные символы, определим процедуру `squeeze`, предназначенную для выполнения следующей задачи: вводить текст из текущего входного потока и выводить тот же текст, переформатированный таким образом, что несколько пробелов между словами заменяются одинарными пробелами. Для простоты предположим, что любой текст, обрабатываемый процедурой `squeeze`, оканчивается точкой и что слова разделены одним или несколькими пробелами, а не другими символами. В таком случае допустимые входные данные имеют вид

```
The      robot tried      to pour wine out      of the      bottle.
```

Цель `squeeze` выведет этот текст в следующей форме:

```
The robot tried to pour wine out of the bottle.
```

Процедура `squeeze` имеет такую же структуру, как и процедуры обработки файлов, приведенные в предыдущем разделе. Вначале она считывает первый символ и выводит его, после чего выполняет остальную часть алгоритма обработки, в зависимости от этого символа. При этом могут рассматриваться три варианта, которые соответствуют следующим случаям: символ представляет собой точку, пробел или букву. Взаимное исключение этих трех вариантов достигается в программе с помощью операторов отсеечения, как показано ниже.

```
squeeze :-  
    get0( C),  
    put( C),  
    dorest( C).  
  
dorest( 46) :- !,    % 46 - это код точки в кодировке ASCII; вся работа выполнена  
dorest( 32) :- !,    % 32 - это код пробела в кодировке ASCII  
    get( C),          % Пропустить остальные пробелы  
    put( C),  
    dorest( C).  
dorest( Letter) :-  
    squeeze.
```

Упражнение

- 6.3. Обобщите процедуру `squeeze`, чтобы она позволяла также обрабатывать запятые. Необходимо удалять все пробелы, которые непосредственно предшествуют запятым, а после каждой запятой должен находиться один пробел.

6.4. Формирование и анализ атомов

Часто желательно иметь возможность представлять в программе в виде атомов информацию, считанную как последовательность символов. Для этой цели может применяться встроенный предикат `name`, который позволяет связать между собой атомы и входящие в состав их имени символы в коде ASCII. Поэтому предикат

```
name( A, L)
```

принимает истинное значение, если `L` — список кодов символов ASCII в имени атома `A`. Например, предикат

```
name( zx232, [122,120,50,51,50])
```

принимает истинное значение. Ниже описаны две типичные области применения предиката `name`.

1. Если дано имя атома, разбить его на отдельные символы.

2. Если дан список символов, объединить его в имя атома.

Примером приложения первого вида может служить программа, которая касается заказов, такси и водителей. Предположим, что эти понятия представлены в программе с помощью примерно таких атомов:

```
order1, order2, driver1, driver2, taxial, taxilux
```

Предикат

```
taxi( X)
```

проверяет, представляет ли атом `X` информацию о такси:

```
taxi( X) :-  
    name( X, Xlist),  
    name( taxi, Tlist),  
    conc( Tlist, _, Xlist).      % Является ли слово 'taxi' префиксом X?
```

Предикаты `order` и `driver` можно определить аналогичным образом.

Следующий пример демонстрирует использование способа объединения символов в атомы. Определим предикат

```
getsentence( Wordlist)
```

который считывает текст на естественном языке в произвольной форме и конкретизирует список `Wordlist` значениями некоторого внутреннего представления этого текста. Наиболее приемлемым вариантом внутреннего представления, позволяющим обеспечить дальнейшую обработку текста, является следующий: каждое слово во входном тексте представляется как атом Prolog, а весь текст преобразуется в список атомов. Например, если текущий входной поток имеет вид

```
Mary was pleased to see the robot fail.
```

то цель `getsentence(Sentence)` вызовет такую конкретизацию:

```
Sentence = [ 'Mary', was, pleased, to, see, the, robot, fail]
```

Для простоты предположим, что текст всегда завершается точкой и в нем отсутствуют знаки препинания.

Соответствующая программа приведена в листинге 6.1. Процедура `getsentence` вначале считывает текущий входной символ `Char`, а затем передает этот символ в процедуру `getrest` для завершения работы. Процедура `getrest` должна выполнять свои действия с учетом описанных ниже трех случаев.

1. Символ Char представляет собой точку. В этом случае чтение текста закончено.
2. Символ Char является пробелом. Игнорировать его и перейти к выполнению процедуры getsentence на остальной части ввода.
3. Символ Char представляет собой букву. Вначале прочитать слово Word, которое начинается с символа Char, затем использовать процедуру getsentence для чтения остальной части текста и формирования списка Wordlist. Суммарным результатом является список [Word | Wordlist].

Листинг 6.1. Процедура преобразования текста в список атомов

```

/*
Процедура getsentence считывает предложение и преобразовывает слова
в список атомов. Например,

getsentence( Wordlist)

создает список

Wordlist = [ 'Mary', was, pleased, to, see, the, robot, fail]

при получении на входе предложения

Mary was pleased to see the robot fail.
*/

getsentence( Wordlist) :-
    get0( Char),
    getrest( Char, Wordlist).

getrest( 46, [] ) :- !.           % Конец предложения; 46 - код ASCII для '.'

getrest( 32, Wordlist) :- !,      % 32 - код ASCII для пробела
    getsentence( Wordlist).       % Пропустить пробел

getrest( Letter, [Word | Wordlist] ) :-
    getletters( Letter, Letters, Nextchar), % Прочитать буквы текущего слова
    name( Word, Letters),
    getrest( Nextchar, Wordlist).

getletters( 46, [], 46) :- !.      % Конец слова; 46 - код точки

getletters( 32, [], 32) :- !.      % Конец слова; 32 - пробел

getletters( Let, [Let | Letters], Nextchar) :-
    get0( Char),
    getletters( Char, Letters, Nextchar).

```

Процедурой, которая считывает символы одного слова, является следующая:

```
getletters( Letter, Letters, Nextchar)
```

Три параметра этой процедуры описаны ниже.

1. Letter — это текущая буква (уже считанная) считываемого слова.
2. Letters — это список букв (начиная с Letter) вплоть до конца слова.
3. Nextchar — это входной символ, который непосредственно следует за считанным словом. Nextchar должен быть небуквенным символом.

Дополним этот пример комментарием, касающимся возможного применения процедуры getsentence. Она может использоваться в программе для обработки текста на естественном языке. Тексты, представленные в виде списков слов, находятся в форме, подходящей для дальнейшей обработки на языке Prolog. Одним из простых

примеров является поиск определенных ключевых слов во входном тексте. Гораздо более сложной задачей может явиться понимание этого текста, иными словами, извлечение из него смысла, представленного в виде некоторой заранее выбранной формальной структуры. Это важное направление исследований в области искусственного интеллекта представлено в главе 21.

Упражнения

6.4. Определите отношение

```
starts( Atom, Character)
```

чтобы проверить, начинается ли имя атома Atom с символа Character.

6.5. Определите процедуру plural, которая преобразует форму существительных в единственном числе в форму во множественном числе, например, следующим образом:

```
?- plural( table, X)
X = tables
```

6.6. Напишите процедуру

```
search( KeyWord, Sentence)
```

которая при каждом ее вызове находит в текущем входном файле предложение, содержащее указанное входное слово KeyWord. Предложение Sentence должно находиться в его первоначальной форме, т.е. должно быть представлено как последовательность символов или как один атом (для этого можно использовать процедуру getsentence из этого раздела, откорректированную соответствующим образом).

6.5. Чтение программ

Для передачи программ в систему Prolog могут использоваться встроенные предикаты, которые позволяют, как принято называть эту операцию, *получить консультацию* из файлов или *откомпилировать* файлы с программами. Подробные сведения о том, как реализованы операции получения консультации и компиляции файлов, зависят от реализации Prolog. Ниже рассматриваются некоторые основные средства, доступные во многих версиях Prolog.

Системе Prolog можно сообщить, что она должна прочитать и обработать программу из файла F, с помощью цели в форме `consult(F)`, например, следующим образом:

```
?- consult( program3)
```

В зависимости от реализации может потребоваться присвоить файлу с указанным именем, в данном случае `program3`, расширение, которое обозначает его как программный файл Prolog. Результатом достижения этой цели становится то, что все предложения в файле `program3` считываются и загружаются в память. Затем эти предложения используются системой Prolog при формировании ответов на последующие вопросы пользователя. В дальнейшем на протяжении того же сеанса для получения консультации может быть загружен другой файл. По сути, результат такой операции состоит в том, что предложения из этого нового файла загружаются в память. Но нюансы выполнения данной операции зависят от реализации Prolog и от других обстоятельств. Например, если новый файл содержит предложения, касающиеся процедуры, которая определена в файле, ранее использовавшемся для получения консультации, то в одном варианте новые предложения могут быть добавлены к концу текущего набора предложений, а другой вариант может предусматривать полную замену предыдущего определения этой процедуры новым определением.

Для получения консультации с помощью одной и той же цели `consult` можно использовать несколько файлов, например, следующим образом:

```
?- consult( [ program3, program4, queens] ).
```

Подобный вопрос может быть также записан более просто:

```
?- [ program3, program4, queens].
```

Программы, использовавшиеся для получения консультации, служат в качестве дополнительной информации для интерпретатора Prolog (именно поэтому операция получения консультации имеет такое название). Если данная реализация Prolog предусматривает также использование компилятора, то программы могут быть загружены в откомпилированной форме. Это позволяет повысить быстродействие программы, причем обычно быстродействие откомпилированного кода по сравнению с интерпретируемым возрастает в 5–10 раз. Программы загружаются в память в откомпилированной форме с помощью встроенного предиката `compile`, например, следующим образом:

```
?- compile( program3 ).
```

или

```
?- compile( [ program4, queens, program6] ).
```

Откомпилированные программы характеризуются более высоким быстродействием, а интерпретируемые программы проще для отладки, поскольку их можно проверять и проводить трассировку с помощью средств отладки Prolog. Поэтому интерпретатор обычно используется на этапе разработки программы, а компилятор — при создании окончательной версии программы.

Еще раз следует отметить, что подробные сведения о получении консультаций и компилировании файлов зависят от реализации Prolog. Обычно любая реализация Prolog позволяет также пользователю вводить и редактировать программу в интерактивном режиме.

Резюме

- *Ввод и вывод* (отличный от того, который связан с выполнением запросов к программе) осуществляются с использованием *встроенных процедур*. В этой главе представлен простой и практичный набор подобных процедур, которые предусмотрены во многих реализациях Prolog.
- Представленный здесь набор предназначен для работы с файлами, имеющими последовательную организацию. В программе имеются *текущий входной поток* и *текущий выходной поток*. Пользовательский терминал рассматривается как файл с именем `user`.
- Переключение между потоками осуществляется с помощью перечисленных ниже предикатов.
 - `see( File)`. Файл `File` становится текущим входным потоком.
 - `tell( File)`. Файл `File` становится текущим выходным потоком.
 - `seen`. Закрывает текущий входной поток.
 - `told`. Закрывает текущий выходной поток.
- Файлы считываются и записываются следующими способами:
 - как последовательности символов;
 - как последовательности термов.
- Ниже перечислены встроенные процедуры для *чтения и записи символов и термов*.
 - `read( Term)`. Вводит следующий терм.

- `write( Term)`. Выводит терм `Term`.
- `put( CharCode)`. Выводит символ с указанным кодом ASCII.
- `get0( CharCode)`. Вводит следующий символ.
- `get( CharCode)`. Вводит следующий печатаемый символ.
- Для форматирования применяются две процедуры, перечисленные ниже.
 - `nl`. Выводит символ с обозначением новой строки.
 - `tab( N)`. Выводит `N` пробелов.
- Процедура `name( Atom, CodeList)` применяется для анализа и формирования атомов. `CodeList` — это список кодов символов ASCII в имени атома `Atom`.
- Во многих реализациях Prolog предусмотрены дополнительные средства для обработки файлов, отличных от последовательных, для вывода на экран окон, поддержки графических примитивов, ввода информации с помощью мыши и т.д.

Дополнительные сведения, касающиеся стандарта Prolog

Для некоторых предикатов, рассматриваемых в данной главе, в стандарте ISO языка Prolog [43] рекомендуются имена, отличные от используемых в большинстве реализаций Prolog. Но эти предикаты концептуально являются теми же самыми, поэтому для обеспечения совместимости достаточно определить соответствие между именами стандартных и фактически реализованных предикатов. Данное замечание относится к следующим предикатам, описанным в настоящей главе: `see(Filename)`, `tell(Filename)`, `get(Code)`, `put(Code)`, `name(Atom, CodeList)`. Соответствующим предикатам в стандарте присвоены такие имена: `set_input(Filename)`, `set_output(Filename)`, `get_code(Code)`, `put_code(Code)`, `atom_codes(Atom, CodeList)`.

Дополнительные встроенные предикаты

В этой главе...

7.1. Проверка типа термов	149
7.2. Создание и декомпозиция термов: предикаты <code>=..</code> , <code>functor</code> , <code>arg</code> и <code>name</code>	156
7.3. Операторы сравнения и проверки на равенство различных типов	159
7.4. Операции с базой данных	161
7.5. Средства управления	164
7.6. Предикаты <code>bagof</code> , <code>setof</code> и <code>findall</code>	165

В этой главе рассматриваются некоторые дополнительные встроенные предикаты, предназначенные для разработки более сложных программ Prolog. Эти средства позволяют реализовывать в программах такие операции, которые не были возможны при использовании только ранее описанных средств. Один из наборов подобных предикатов предназначен для манипулирования термами: проверки того, была ли некоторая переменная конкретизирована значением целого числа, декомпозиции термов, конструирования новых термов и т.д. Еще один полезный набор процедур предназначен для манипулирования базой данных системы Prolog; эти процедуры позволяют добавлять к программе новые предложения или удалять существующие.

Состав встроенных предикатов в значительной степени зависит от реализации Prolog. Но предикаты, рассматриваемые в данной главе, предусмотрены во многих реализациях Prolog, а в некоторых реализациях могут применяться дополнительные средства.

7.1. Проверка типа термов

7.1.1. Предикаты `var`, `nonvar`, `atom`, `integer`, `float`, `number`, `atomic`, `compound`

Термы могут принадлежать к разным типам: переменная, целое число, атом и т.д. Если терм представляет собой переменную, то он может быть в некоторый момент во время выполнения программы конкретизирован или не конкретизирован. Кроме того, если он конкретизирован, его значением может быть атом, структура и т.д. Иногда необходимо знать, к какому типу относится это значение. Например, в программе может потребоваться сложить значения двух переменных, `X` и `Y`, следующим образом:

```
Z is X + Y
```

Прежде чем выполнить эту цель, необходимо конкретизировать X и Y числовыми значениями. А если нет полной уверенности в том, что к этому моменту X и Y действительно конкретизированы числовыми значениями, то следует проверить это в программе перед выполнением арифметической операции.

Для этого может использоваться встроенный предикат `number`. Предикат `number(X)` принимает истинное значение, если X — число или переменная, значением которой является число. В таких случаях принято говорить, что X “должен в настоящее время обозначать” число. Таким образом, цель, предусматривающую сложение X и Y , можно *защитить* (обеспечить ее безошибочное выполнение) с помощью следующей проверки X и Y :

```
..., number( X ), number( Y ), Z is X + Y, ...
```

Если и X , и Y не являются целыми числами, то не будет предпринята попытка выполнить арифметическую операцию. Поэтому предикат `number` предназначен для “защиты” цели `Z is X + Y` от бессмысленного выполнения.

К встроенным предикатам такого типа относятся `var`, `nonvar`, `atom`, `integer`, `float`, `number`, `atomic`, `compound`. Назначение этих предикатов описано ниже.

- `var( X )`. Выполняется успешно, если X в настоящее время — неконкретизированная переменная.
- `nonvar( X )`. Выполняется успешно, если X — не переменная или X — уже конкретизированная переменная.
- `atom( X )`. Принимает истинное значение, если X в настоящее время обозначает атом.
- `integer( X )`. Принимает истинное значение, если X в настоящее время обозначает целое число.
- `float( X )`. Принимает истинное значение, если X в настоящее время обозначает число с плавающей точкой.
- `number( X )`. Принимает истинное значение, если X в настоящее время обозначает число.
- `atomic( X )`. Принимает истинное значение, если X в настоящее время обозначает число или атом.
- `compound( X )`. Принимает истинное значение, если X в настоящее время обозначает составной терм (структуру).

Приведенные ниже примеры вопросов к системе Prolog иллюстрируют использование этих встроенных предикатов.

```
?- var(Z), Z = 2.  
Z = 2  
?- Z = 2, var( Z ).  
no  
?- integer( Z ), Z = 2.  
no  
?- Z = 2, integer( Z ), nonvar( Z ).  
Z = 2  
?- atom( 3.14 ).  
no  
?- atomic( 3.14 ).  
yes  
?- atom( ==> ).  
yes  
?- atom( p(1) ).  
no  
?- compound( 2 + X )  
yes
```

Рассмотрим необходимость в использовании предиката `atom` на примере. Предположим, что требуется подсчитать, сколько раз указанный атом встречается в заданном списке объектов. Для этой цели определим следующую процедуру:

```
count( A, L, N)
```

где `A` — атом, `L` — список, а `N` — количество вхождений. Первая попытка определить процедуру `count` может представлять собой следующее:

```
count( _, [], 0).
count( A, [A | L], N) :- !,
    count( A, L, N1), % N1 - количество вхождений атома в хвосте списка
    N is N1 + 1.
count( A, L | [], N) :-
    count( A, L, N).
```

Проверим функционирование этой процедуры на нескольких примерах следующим образом:

```
?- count( a, [a,b,a,a], N).
N = 3
?- count( a, [a,b,X,Y], Na).
Na = 3
...
?- count( b, [a,b,X,Y], Nb).
Nb = 3
...
?- L = [a, b, X, Y], count( a, L, Na), count( b, L, Nb).
Na = 3
Nb = 1
X = a
Y = a
...
```

В последнем примере и `X`, и `Y` были конкретизированы значением `a`, поэтому были получены результаты подсчета, учитывающие только одно вхождение `a`, `Nb = 1`, но это не входило в наши намерения. Данная процедура предназначена для подсчета количества реальных вхождений указанного атома, а не количества термов, которые согласуются с этим атомом. В соответствии с данным более точным определением отношения `count` необходимо проверить, является ли голова рассматриваемого списка атомом. Модифицированная программа приведена ниже.

```
count( _, [], 0).
count( A, [B | L], N) :-
    atom( B), A = B, !, % Представляет ли собой B атом A?
    count( A, L, N1), % Если да, то выполнить подсчет в хвосте списка
    N is N1 + 1
    ;
    count( A, L, N). % Иначе выполнять подсчет только в хвосте списка
```

В следующем, более сложном упражнении по программированию в области решения числовых ребусов применяется предикат `nonvar`.

7.1.2. Решение числового ребуса с использованием предиката `nonvar`

Широко известный пример числового ребуса приведен ниже.

```
DONALD
+ GERALD
-----
ROBERT
```

Для решения этой задачи требуется записать вместо букв `D`, `O`, `N` и т.д. десятичные цифры таким образом, чтобы приведенная выше операция суммирования оказалась правильной. Всем буквам должны быть присвоены разные числовые значения,

поскольку в противном случае возможны тривиальные решения, например, в одном из таких решений все буквы равны нулю.

Определим следующее отношение:

```
sum( N1, N2, N)
```

где N1, N2 и N представляют три числа в указанном числовом ребусе. Цель `sum( N1, N2, N)` является истинной, если существует такая замена букв цифрами, что $N1 + N2 = N$.

Первым шагом к решению поставленной задачи является определение того, как представить числа N1, N2 и N в программе. Один способ достижения этой цели может предусматривать представление каждого числа в виде списка десятичных цифр. Например, число 225 может быть представлено списком [2, 2, 5]. Если эти цифры заранее не известны, вместо каждой цифры может быть подставлена неконкретизированная переменная. С использованием такого представления формулировку рассматриваемой задачи можно выразить таким образом:

```
[D, O, N, A, L, D]
+ [G, E, R, A, L, D]
= [R, O, B, E, R, T]
Number1 = [D11, D12, ..., D1i, ... ]
Number2 = [D21, D22, ..., D2i, ... ]
Number3 = [D31, D32, ..., D3i, ... ]
```

Требуется найти такую конкретизацию переменных D, O, N и т.д., для которой эта сумма становится действительной. После того как отношение `sum` будет запрограммировано, системе Prolog можно представить для решения эту задачу, задав следующий вопрос:

```
?- sum( [D, O, N, A, L, D], [G, E, R, A, L, D], [R, O, B, E, R, T] ).
```

Чтобы определить отношение `sum` на списках, состоящих из десятичных цифр, необходимо реализовать в программе правила суммирования в десятичной системе счисления. Суммирование выполняется поразрядно, начиная с крайнего правого младшего разряда, и продолжается в направлении влево, в сторону старших разрядов; при этом всегда учитывается цифра переноса из предыдущего разряда. Кроме того, необходимо предусмотреть использование множества доступных цифр, т.е. цифр, которые еще не использовались для конкретизации ранее встретившихся переменных. Поэтому в целом, кроме трех чисел, N1, N2 и N, требуется еще некоторая дополнительная информация (рис. 7.1):

- цифра переноса, образовавшаяся до суммирования чисел;
- цифра переноса, образовавшаяся после суммирования чисел;
- множество цифр, доступных перед суммированием;
- оставшиеся цифры, которые не использовались при суммировании.

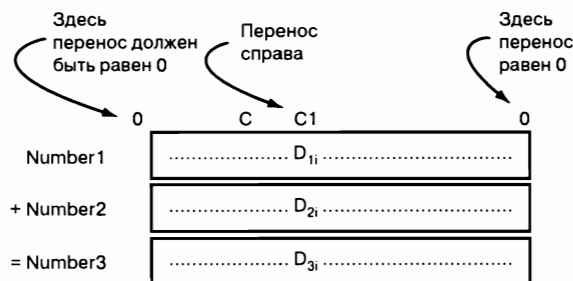


Рис. 7.1. Поразрядное суммирование; цифры в разряде i связаны между собой следующими соотношениями: $D_{3i} = (C1 + D_{1i} + D_{2i}) \bmod 10$, $C = (C1 + D_{1i} + D_{2i}) \div 10$

Для формулировки отношения `sum` снова воспользуемся принципом обобщения проблемы; для этого введем вспомогательное, более общее отношение, `sum1`. Отношение `sum1` имеет некоторые дополнительные параметры, которые соответствуют описанной выше дополнительной информации:

```
sum1( N1, N2, N, C1, C, Digits1, Digits)
```

где `N1`, `N2` и `N` — три числа, такие же, как в отношении `sum`, `C1` — цифра переноса из предыдущего разряда (до суммирования `N1` и `N2`) и `C` — цифра переноса в следующий разряд (после суммирования `N1` и `N2`). Пример применения этого отношения показан ниже.

```
?- sum1( [H,E], [6,E], [U,S], 1, 1, [1,3,4,7,8,9], Digits).
H = 8
E = 3
S = 7
U = 4
Digits = [1,9]
```

Этот пример соответствует следующей операции суммирования:

```
1 ← ← 1
  8  3
  6  3
  ---
  4  7
```

Как показано на рис. 7.1, переменные `C1` и `C` должны быть равны 0, если `N1`, `N2` и `N` соответствуют отношению `sum`. Кроме того, `Digits1` — список доступных цифр для конкретизации переменных в числах `N1`, `N2` и `N`, а `Digits` — список цифр, которые не использовались при конкретизации этих переменных. Поскольку при подборе чисел, соответствующих отношению `sum`, разрешается использовать любые десятичные цифры, отношение `sum` может быть определено в терминах отношения `sum1` следующим образом:

```
sum( N1, N2, N) :-
    sum1( N1, N2, N, 0, 0, [0,1,2,3,4,5,6,7,8,9], _).
```

Теперь вся сложность решения проблемы состоит в создании отношения `sum1`. Но это отношение является достаточно общим, чтобы его можно было определить рекурсивно. Предположим без потери общности, что три списка, представляющих три числа, имеют одинаковую длину. Безусловно, что рассматриваемый пример задачи удовлетворяет этому ограничению; в ином случае “более короткое” число можно дополнить слева нулями.

При определении отношения `sum1` могут рассматриваться два случая, как показано ниже.

1. Три числа представлены пустыми списками, таким образом:

```
sum1( [], [], [], C, C, Digs, Digs).
```

2. Все три числа имеют какой-то крайний левый (старший) разряд и оставшиеся цифры справа от этого разряда, поэтому они имеют следующую форму:

```
[D1 | N1], [D2 | N2], [D | N]
```

В этом случае должны быть выполнены два приведенных ниже условия.

- а) Три числа, `N1`, `N2` и `N`, должны соответствовать отношению `sum1`; при этом формируется некоторая цифра переноса (`C2`) в старший разряд и остается некоторое неиспользуемое подмножество десятичных цифр, `Digs2`.
- б) Самые старые разряды (`D1`, `D2` и `D`) и цифра переноса (`C2`) должны соответствовать соотношениям, показанным на рис. 7.1, согласно которым в результате сложения цифр `C2`, `D1` и `D2` формируются цифра `D` и цифра переноса в старший разряд. Это условие можно сформулировать в программе как отношение `digitsum`.

После перевода на язык Prolog условий, представленных в рассматриваемом случае, получаем следующее:

```
suml( [D1 | N1], [D2 | N2], [D | N], C1, C, Digs1, Digs) :-
    suml( N1, N2, N, C1, C2, Digs1, Digs2),
    digitsum( D1, D2, C2, D, C, Digs2, Digs).
```

Остается только определить на языке Prolog отношение digitsum. Следует отметить один тонкий нюанс, который касается использования металогического предиката nonvar. Переменные D1, D2 и D должны представлять собой десятичные цифры. Если одна из этих переменных еще не конкретизирована, она должна стать конкретизированной значением одной из цифр в списке Digs2. После этого данную цифру необходимо удалить из набора доступных цифр. А если переменные D1, D2 или D уже конкретизированы, то, безусловно, ни одна из доступных цифр не должна быть затронутой на их конкретизацию. Это условие реализуется в программе в виде недетерминированной операции удаления элемента от списка. Если элемент не является переменной, то ничего не удаляется (конкретизация не происходит). Ниже приведена программа, соответствующая этому условию.

```
del_var( Item, List, List) :-
    nonvar( Item), !.                                % Элемент Item уже конкретизирован
del_var( Item, [Item | List], List).                  % Удалить голову списка
del_var( Item, [A | List], [A | List]) :-
    del_var( Item, List, List).                        % Удалить элемент Item из хвоста списка
```

Окончательный вариант программы для решения числовых ребусов приведен в листинге 7.1. Эта программа включает также определения двух задач. С использованием данной программы можно задать системе Prolog вопрос, касающийся замены цифрами букв в словах DONALD, GERALD и ROBERT, следующим образом:

```
?- puzzle1( N1, N2, N), sum( N1, N2, N).
```

Листинг 7.1. Программа решения числовых ребусов

```
% Решение числовых ребусов

sum(N1, N2, N) :-                                % Числа, представленные как списки цифр
    suml( N1, N2, N,
        0, 0,                                % Обе цифры переноса, справа и влево, равны 0
        [0,1,2,3,4,5,6,7,8,9], _).           % Все доступные цифры

suml( [], [], [], C, C, Digits, Digits).

suml( [D1|N1], [D2|N2], [D|N], C1, C, Digs1, Digs) :-
    suml( N1, N2, N, C1, C2, Digs1, Digs2),
    digitsum( D1, D2, C2, D, C, Digs2, Digs).

digitsum( D1, D2, C1, D, C, Digs1, Digs) :-
    del_var( D1, Digs1, Digs2),                % Выбрать доступную цифру для D1
    del_var( D2, Digs2, Digs3),                % Выбрать доступную цифру для D2
    del_var( D, Digs3, Digs),                  % Выбрать доступную цифру для D
    S is D1 + D2 + C1,
    D is S mod 10,                             % Остаток
    C is S // 10.                              % Целочисленное деление

del_var( A, L, L) :-
    nonvar(A), !.                                % Переменная A уже конкретизирована

del_var( A, [A|L], L).                          % Удалить голову списка

del_var( A, [B|L], [B|L]) :-
    del_var( A, L, L).                          % Удалить элемент из хвоста списка

% Некоторые задачи
```

```
puzzle1( [D,O,N,A,L,D],
         [G,E,R,A,L,D],
         [R,O,B,E,R,T] ).
```

```
puzzle2( [O,S,E,N,D],
         [O,M,O,R,E],
         [M,O,N,E,Y] ).
```

Иногда решение подобных задач можно упростить, указав в качестве дополнительного ограничения цифровое значение одной из букв (например, сообщив, что D равно 5). Задачу, представленную в такой форме, можно сообщить системе Prolog с использованием отношения `sum1` следующим образом:

```
?- sum1( [5,O,N,A,L,5],
         [G,E,R,A,L,5],
         [R,O,B,E,R,T],
         0, 0, [0,1,2,3,4,6,7,8,9], _ ).
```

Любопытно отметить, что в обоих случаях имеется только одно решение. Иными словами, существует единственный способ замены букв цифрами.

Упражнения

- 7.1. Напишите процедуру `simplify` для упрощения символических выражений суммирования, состоящих из цифр и символов (прописных букв). Допустим, что эта процедура должна переупорядочивать слагаемые в выражениях таким образом, чтобы символы предшествовали цифрам. Ниже приведены примеры использования такой процедуры.

```
?- simplify( 1 + 1 + a, E ).
E = a + 2
?- simplify( 1 + a + 4 + 2 + b + c, E ).
E = a + b + c + 7
?- simplify( 3 + x + x, E ).
E = 2*x + 3
```

- 7.2. Определите процедуру

```
add_to_tail( Item, List)
```

для сохранения нового элемента в списке. Предположим, что все элементы, которые могут быть сохранены, не являются переменными. Список `List` содержит все сохраненные элементы. За ними следует хвост списка, который не конкретизирован и поэтому способен принимать новые элементы. Например, допустим, что в настоящее время в списке хранятся элементы `a`, `b` и `c`. В таком случае

```
List = [a, b, c | Tail]
```

где `Tail` — переменная. Цель

```
add_to_tail( d, List)
```

вызывает следующую конкретизацию:

```
Tail = [d | NewTail] и List = [a, b, c, d | NewTail]
```

Поэтому такая структура, по сути, может расти, принимая новые элементы. Определите также соответствующее отношение для проверки принадлежности к списку.

7.2. Создание и декомпозиция термов: предикаты `=..`, `functor`, `arg` и `name`

Для декомпозиции термов и создания новых термов предусмотрены три встроенных предиката: `functor`, `arg` и `"=.."`. Вначале рассмотрим предикат `=..`, который записывается как инфиксный оператор и читается как "юнив" (`univ`). Цель

```
Term =.. L
```

является истинной, если `L` — список, содержащий главный функтор `Term`, за которым следуют его параметры. Применение этого предиката демонстрируется на следующих примерах:

```
?- f( a, b) =.. L.
L = [f, a, b]
?- T =.. [rectangle, 3, 5].
T = rectangle( 3, 5)
?- Z =.. [p, X, f(X,Y)].
Z = p( X, f(X,Y))
```

Необходимость декомпозиции терма на его компоненты (функтор и параметры) и создания нового терма из указанного функтора и параметров можно продемонстрировать на следующем примере.

Рассмотрим программу, предназначенную для манипулирования геометрическими фигурами. К числу таких фигур относятся квадраты, прямоугольники, треугольники, окружности и т.д. Эти фигуры могут быть представлены в программе в виде термов, в которых функтор обозначает тип фигур, а параметры задают размер фигуры, например, как показано ниже.

```
square( Side)
triangle( Side1, Side2, Side3)
circle( R)
```

Одной из операций над подобными фигурами может быть их увеличение. Такую операцию можно реализовать в виде следующего отношения с тремя параметрами:

```
enlarge( Fig, Factor, Fig1)
```

где `Fig` и `Fig1` — геометрические фигуры одного и того же типа (с одинаковым функтором), а параметрами фигуры `Fig1` являются параметры фигуры `Fig`, увеличенные путем умножения на коэффициент `Factor`. Для простоты предположим, что все параметры фигуры `Fig` уже известны (иными словами, они конкретизированы числовыми значениями), известен также и коэффициент `Factor`. Ниже показан один из способов программирования отношения `enlarge`.

```
enlarge( square(A), F, square(A1)) :-
    A1 is F*A.
enlarge( circle(R), F, circle(R1)) :-
    R1 is F*R.
enlarge( rectangle(A,B), F, rectangle(A1,B1)) :-
    A1 is F*A, B1 is F*B.
...
```

Такой подход позволяет добиться успеха, но становится очень громоздким, если количество типов фигур достаточно велико. Кроме того, он не позволяет заранее учесть все возможные типы фигур, которые в дальнейшем могут быть предусмотрены в программе. Поэтому для каждого нового типа потребуется еще одно предложение, хотя во всех предложениях выполняется, по сути, одно и то же действие: берутся параметры первоначальной фигуры, все эти параметры умножаются на заданный коэффициент, после чего создается фигура того же типа с новыми параметрами.

Ниже показана одна (безуспешная) попытка обрабатывать, по меньшей мере, все фигуры с одним параметром с помощью одного предложения.


```
enlarge( Type(Par), F, Type(Par1) ) :-
    Par1 is F*Par.
```

Но обычно использование подобных конструкций в языке Prolog не допускается, поскольку функтор должен быть атомом, поэтому переменная Type не может применяться в позиции функтора по требованиям синтаксиса. Возможный способ выхода из этой ситуации состоит в использовании предиката “=..”. В таком случае процедура enlarge может быть представлена полностью обобщенно (для геометрического объекта любого типа) следующим образом:

```
enlarge( Fig, F, Fig1) :-
    Fig =.. [Type | Parameters],
    multiplylist( Parameters, F, Parameters1),
    Fig1 =.. [Type | Parameters1].
```

```
multiplylist( [], _, []).
multiplylist( [X | L], F, [X1 | L1]) :-
    X1 is F*X, multiplylist( L, F, L1).
```

Следующий пример использования предиката “=..” относится к области манипулирования символьными обозначениями, применяемыми в формулах. При этом часто используется операция, в которой определенное подвыражение заменяется другим выражением. Определим отношение

```
substitute( Subterm, Term, Subterm1, Term1)
```

следующим образом: если все вхождения подвыражения Subterm в выражении Term замещаются подвыражением Subterm1, то будет получено выражение Term1, например, как показано ниже.

```
?- substitute( sin(x), 2*sin(x)*f(sm(x)), t, F).
F = 2*t*f(t)
```

Под *вхождением* Subterm в Term подразумевается некоторая часть выражения Term, которая сопоставляется с подвыражением Subterm. Поиск вхождений осуществляется сверху вниз, поэтому цель

```
?- substitute( a+b, f( a, A+B), v, F).
```

приводит к получению следующих результатов:

```
F = f( a, v)           F = f( a, v + v)
A = a,                A = a + b
B = b                  B = a + b
```

При определении отношения substitute необходимо предусмотреть принятие следующих решений в зависимости от конкретной ситуации:

```
Если Subterm = Term, то Term1 = Subterm1;
в противном случае, если Term относится к типу 'atomic'
(не является структурой),
то Term1 = Term (ничего не требует замены),
в противном случае замена должна быть выполнена в параметрах Term.
```

Эти правила могут быть преобразованы в программу Prolog, показанную в листинге 7.2.

Листинг 7.2. Процедура замены субтерма в терме другим субтермом

```
% substitute( Subterm, Term, Subterm1, Term1):
% если все вхождения субтерма Subterm в терме Term будут заменены
% субтермом Subterm1, то будет получен терм Term1

% Случай 1. Замена всего терма

substitute( Term, Term, Term1, Term1) :- !.

% Случай 2. Ничего не требует замены, если Term относится к типу 'atomic'

substitute( _, Term, _, Term) :-
```

```

atomic(Term), !.

% Случай 3. Выполнение замены в параметрах

substitute( Sub, Term, Sub1, Term1) :-
    Term =.. [F|Args],
    substlist( Sub, Args, Sub1, Args1),
    Term1 =.. [F|Args1].
                                % Получить параметры
                                % Выполнить в них замену

substlist( _, [], _, []).

substlist( Sub, [Term|Terms], Sub1, [Term1|Terms1]) :-
    substitute( Sub, Term, Sub1, Term1),
    substlist( Sub, Terms, Sub1, Terms1).
.....

```

Безусловно, в качестве целей могут также использоваться термы, созданные с помощью предиката “=..”. Преимущество этого подхода состоит в том, что программа приобретает способность в процессе своей работы самостоятельно вырабатывать и выполнять цели, представленные в формах, которые не всегда можно было предвидеть ко времени написания этой программы. Последовательность целей, иллюстрирующих подобный эффект, может выглядеть таким образом:

```

obtain( Functor),
compute( Arglist),
Goal =.. [Functor | Arglist],
Goal

```

Здесь `obtain` и `compute` представляют собой некоторые определяемые пользователем процедуры для получения компонентов цели, которая должна быть создана в программе. После этого цель создается с помощью предиката “=..” и вызывается на выполнение путем указания ее имени, `Goal`.

Некоторые реализации Prolog могут требовать, чтобы все цели по мере их появления в программе синтаксически представляли собой либо атомы, либо структуры с атомом в качестве главного функтора. Поэтому в подобном случае переменная, независимо от ее окончательной конкретизации, может оказаться синтаксически неприемлемой для использования в качестве цели. Эту проблему можно обойти с помощью еще одного встроенного предиката, `call`, параметром которого является выполняемая цель. В соответствии с этим приведенный выше пример может быть переформулирован таким образом:

```

...
Goal =.. [Functor | Arglist],
call( Goal)

```

Иногда может потребоваться извлечь из терма только его главный функтор или один из параметров. Для этого допустимо, безусловно, воспользоваться отношением “=..”. Но может оказаться, что проще и эффективнее использовать одну из двух других встроенных процедур для манипулирования термами: `functor` и `arg`. Их назначение может быть описано следующим образом: цель

```
functor( Term, F, N)

```

является истинной, если `F` — главный функтор терма `Term` и `N` — арность функтора `F`. С другой стороны, цель

```
arg( N, Term, A)

```

является истинной, если `A` — `N`-й параметр в терме `Term`, при условии, что параметры пронумерованы слева направо, начиная с 1. Применение этих предикатов демонстрируется на следующих примерах:

```

?- functor( t( f(X), X, t), Fun, Arity)
Fun = t
Arity = 3
?- arg( 2, f( X, t(a), t(b)), Y).

```

```
Y = t(a)
?- functor( D, date, 3),
arg( 1, D, 29),
arg( 2, D, June),
arg( 3, D, 1982).
D = date( 29, june, 1982)
```

Последний пример показывает особую область применения предиката `functor`. Цель `functor( D, date, 3)` создает *общий терм* с тремя параметрами, главным функтором которого является `date`. Этот терм является общим в том смысле, что три его параметра представляют собой неконкретизированные переменные, имена которых генерируются системой Prolog, например:

```
D = date( _5, _6, _7)
```

Затем эти три переменные конкретизируются в приведенном выше примере с помощью трех целей `arg`.

С этим набором встроенных предикатов связан предикат `name`, применяемый для создания/декомпозиции атомов (см. главу 6). В данном разделе для полноты снова рассмотрим его назначение. Предикат

```
name( A, L)
```

принимает истинное значение, если `L` — список кодов символов ASCII в имени атома `A`.

Упражнения

7.3. Определите предикат `ground( Term)` таким образом, чтобы он принимал истинное значение, если терм `Term` не содержит неконкретизированных переменных.

7.4. Процедура `substitute`, описанная в данном разделе, вырабатывает только “самые внешние” подстановки, если даже есть другие варианты. Измените эту процедуру таким образом, чтобы с помощью перебора с возвратами вырабатывались все возможные варианты, например, как показано ниже.

```
?- substitute( a+b, f(A+B), new, NewTerm).
A = a
B = b
NewTerm = f( new);
A = a+b
B = a+b
NewTerm = f( new + new)
```

Первоначальная версия позволяет найти только первый ответ.

7.5. Определите отношение

```
subsumes( Term1, Term2)
```

таким образом, чтобы оно позволяло определить, является ли выражение `Term1` более общим, чем выражение `Term2`, например, как показано ниже.

```
?- subsumes( X, c).
yes
?- subsumes( g(X), g(t(Y)) ).
yes
?- subsumes( f(X,X), f(a,b) ).
no
```

7.3. Операторы сравнения и проверки на равенство различных типов

Рассмотрим, при каких условиях два терма рассматриваются как равные. До сих пор были представлены три типа операторов проверки на равенство, применяемых в

языке Prolog. Первый из них основан на согласовании и записывается следующим образом:

$X = Y$

Он принимает истинное значение, если X и Y согласуются. Еще один тип оператора проверки на равенство записывается таким образом:

$X \text{ is } E$

Он принимает истинное значение, если X соответствует значению арифметического выражения E . Кроме того, рассматривался следующий оператор:

$E1 \text{ } =:= \text{ } E2$

Он принимает истинное значение, если равны значения арифметических выражений $E1$ и $E2$. В отличие от этого, если значения двух арифметических выражений проверяются на неравенство, можно применить следующий оператор:

$E1 \text{ } \backslash= \text{ } E2$

Иногда в программе необходимо применить более строгий тип проверки на равенство: буквальное равенство, или идентичность двух термов. Такая проверка на равенство осуществляется с использованием еще одного встроенного предиката, который записывается как инфиксный оператор $==$:

$T1 == T2$

Он принимает истинное значение, если термы $T1$ и $T2$ *идентичны*; иными словами, они имеют полностью одинаковую структуру и все их соответствующие компоненты являются одинаковыми. В частности, должны быть также одинаковыми имена переменных. Обратным по смыслу отношением является отношение проверки на *неидентичность*, которое записывается следующим образом:

$T1 \backslash== T2$

Ниже приведены некоторые примеры применения операторов проверки на идентичность и неидентичность.

```
?- f( a, b) == f( a, b).
yes
?- f( a, b) == f( a, X).
no
?- f( a, X) == f( a, Y).
no
?- X \== Y.
yes
?- t( X, f(a,Y)) == t( X, f(a,Y)).
yes
```

В качестве примера переопределим отношение

$\text{count}(\text{Term}, \text{List}, N)$

рассматриваемое в разделе 7.1. Предположим, что в данном случае N представляет собой количество буквальных вхождений терма Term в список List :

```
count(_, [], 0).
count(Term, [Head | L ], N) :-
    Term == Head, !,
    count(Term, L, N1),
    N is N1 + 1
;
count(Term, L, N).
```

В этой книге уже рассматривались предикаты, предназначенные для сравнения арифметических значений термов, например $X + 2 < 5$. Еще один набор встроенных предикатов позволяет сравнивать лексикографические значения термов и поэтому определять на термах отношения упорядочения. Например, цель

$X @< Y$

расшифровывается таким образом: терм X предшествует терму Y . Отношение предшествования между простыми термами определяется путем алфавитного или цифрового упорядочения. Отношение предшествования между структурами определяется как предшествование их главных функторов. А если главные функторы равны, то решение принимается на основе определения отношений предшествования между самыми верхними, самыми левыми функторами в субтермах переменных X и Y . Соответствующие примеры приведены ниже.

```
?- paul @< peter.
yes
?- f(2)@<f(3).
yes
?- g(2)@<f(3).
no
?- g(2) @>= f(3).
yes
?- f( a, g(b), c) @< f( a, h(a), a).
yes
```

К этому же типу относятся все встроенные предикаты $@<$, $@=<$, $@>$, $@>=$, назначение которых очевидно.

7.4. Операции с базой данных

База данных, в соответствии с реляционной моделью баз данных, представляет собой спецификацию набора отношений. Любая программа Prolog может рассматриваться как подобная база данных; в ней спецификация отношений частично является явной (факты), а частично неявной (правила). Некоторые встроенные предикаты позволяют модифицировать эту базу данных во время выполнения программы. Такое обновление осуществляется путем добавления новых предложений в программу или удаления существующих предложений (причем эти операции осуществляются во время выполнения программы). Для этого предназначены предикаты `assert`, `asserta`, `assertz` и `retract`. Цель

```
assert( C)
```

всегда достигается и в качестве побочного эффекта вызывает *подтверждение* (assertion) предложения C , т.е. внесение его в базу данных.

Цель

```
retract( C)
```

выполняет противоположное действие: удаляет предложение, которое согласуется с предложением C . Выполнение этих операций можно продемонстрировать на примере следующего диалога с системой Prolog:

```
?- crisis.
no
?- assert( crisis).
yes
?- crisis.
yes
?- retract( crisis).
yes
?- crisis.
no
```

Таким образом, внесенные в базу данных предложения действуют точно так же, как часть “первоначальной” программы. Приведенный ниже пример показывает один из способов использования предикатов `assert` и `retract` для учета изменений ситуации. Предположим, что имеется следующая программа, в которой регистрируется состояние погоды:

```

nice :-
    sunshine, not raining.

funny :-
    sunshine, raining.

disgusting :-
    raining, fog.

raining.
fog.

```

Ниже приведен диалог с этой программой, который показывает, как постепенно изменяются содержимое базы данных и реакция программы.

```

?- nice.
no
?- disgusting.
yes
?- retract( fog).
yes
?- disgusting.
no
?- assert( sunshine).
yes
?- funny.
yes
?- retract( raining).
yes
?- nice.
yes

```

Вносить или извлекать из базы данных можно предложения любой формы. Но в зависимости от реализации Prolog может потребоваться, чтобы предикаты, добавляемые и извлекаемые с помощью предикатов `assert/retract`, были объявлены как динамические. Для этого используется директива `dynamic( PredicateIndicator)`. Предикаты, внесенные в базу данных с помощью предиката `assert`, а не `consult`, автоматически рассматриваются как динамические.

Следующий пример показывает, что предикат `retract` также является недетерминированным: с помощью единственной цели `retract` может быть удален целый набор предложений путем перебора с возвратами. Предположим, что в программе, применяемой для “консультации”, имеется следующий набор фактов:

```

fast( ann).
slow( tom).
slow( pat).

```

К этой программе можно добавить некоторое правило следующим образом:

```

?- assert(
(faster(X,Y) :-
fast(X), slow(Y) ) ).
yes
?- faster( A, B).
A = ann
B = tom
?- retract( slow(X) ).
X = tom;
X = pat;
no
?- faster( ann, _).
no

```

Обратите внимание на то, что при внесении в базу данных это правило (как параметр предиката `assert`) в соответствии с синтаксисом было заключено в круглые скобки.

При внесении предложения в базу данных может потребоваться указать позицию, в которой это новое предложение должно быть вставлено в базу данных. Предикаты `asserta` и `assertz` предоставляют возможность управлять позицией вставки. Цель `asserta(C)`

добавляет предложение `C` в начале базы данных, а цель

`assertz(C)`

вставляет предложение `C` в конце базы данных. Предполагается, что предикат `assert` эквивалентен `assertz`, как обычно предусмотрено в реализациях Prolog. В следующем примере демонстрируется, какое влияние на базу данных оказывают эти предикаты:

```
?- assert( p(b)), assertz( p(c)), assert( p(d)), asserta( p(a)).
yes
?- P(X)
X = a;
X = b;
X = c;
X = d
```

Предикаты `consult` и `assertz` связаны между собой определенной зависимостью. Получение консультации из файла с помощью предиката `consult` можно определить в терминах предиката `assertz` следующим образом: для получения консультации из файла прочитать каждый терм (предложение) файла и внести его в конец базы данных.

Одна из типичных областей применения предиката `asserta` состоит в сохранении уже вычисленных ответов на вопросы. Например, предположим, что в программе определен следующий предикат:

```
solve( Problem, Solution)
```

Теперь системе можно задать некоторый вопрос и потребовать, чтобы она запомнила ответ, для того, чтобы он мог использоваться в будущих вопросах:

```
?- solve( problem1, Solution),
asserta( solve( problem1, Solution)).
```

Если первая из приведенных целей достигается, то ответ (`Solution`) запоминается и используется наряду с любыми другими предложениями при формировании ответов на дальнейшие вопросы. Преимуществом такого “запоминания” ответов является то, что на дальнейшие вопросы, соответствующие подтвержденным фактам, система обычно находит ответ намного быстрее, чем на первый вопрос. После этого результат может быть просто выбран из базы данных как факт, не требуя вычисления в процессе, который, возможно, занимает много времени. Такой метод сохранения ранее полученных решений называют также *кэшированием*.

В одном из вариантов реализации этой идеи операция внесения в базу данных используется для выработки всех решений в форме таблицы фактов. Например, имеется возможность сформировать таблицу произведений всех пар целых чисел от 0 до 9 следующим образом: сформировать пару целых чисел X и Y , вычислить выражение Z is $X*Y$, внести в базу данных три числа в виде одной строки таблицы произведений, а затем вызвать ситуацию недостижения цели. Эта ситуация приведет к тому, что в результате перебора с возвратами будет сформирована еще одна пара целых чисел и в таблицу будет введена еще одна строка и т.д. Эта идея реализована в приведенной ниже процедуре `maketable`.

```
maketable :-
    L = [0,1,2,3,4,5,6,7,8,9],
    member( X, L),           % Выбрать первый коэффициент
    member( Y, L),           % Выбрать второй коэффициент
    Z is X*Y,
    assert( product(X,Y,Z)),
    fail.
```

Безусловно, что вопрос

```
?- maketable.
```

окончится неудачей, но в качестве побочного эффекта позволит ввести в базу данных всю таблицу умножения. После этого системе можно задать вопрос, например, какая пара чисел дает произведение 8:

```
?- product( A, B, 8).  
A = 1  
B = 8;  
A = 2  
B = 4;  
...
```

На данном этапе необходимо сделать одно замечание в отношении такого стиля программирования. Приведенные выше примеры показывали некоторые, безусловно, удобные способы применения предикатов `assert` и `retract`. Но при использовании этих предикатов необходимо соблюдать особую осторожность. Излишнее и непродуманное применение этих средств нельзя рекомендовать в качестве хорошего стиля программирования. Использование операций подтверждения и извлечения фактов и правил по сути приводит к модификации программы. Поэтому отношения, которые в какой-то момент были действительными, в другое время могут оказаться недействительными. В разное время на одни и те же вопросы система будет давать разные ответы. Поэтому применение значительного количества операций подтверждения и извлечения может затемнить смысл программы. В конечном итоге может оказаться, что поведение программы трудно понять, нелегко объяснить, и поэтому ей нельзя доверять.

Упражнения

7.6. Выполните указанные ниже действия после проведения упражнений с таблицей `product`.

- Составьте вопрос к системе Prolog для удаления всей таблицы `product` из базы данных.
- Измените вопрос таким образом, чтобы он удалял только те записи, в которых произведение равно нулю.

7.7. Определите отношение

```
copy_term( Term, Copy)
```

создающее такую копию `Copy` терма `Term`, в которой переименованы все переменные. Программу решения данной задачи легко сформировать с использованием предикатов `asserta` и `retract`. В некоторых реализациях Prolog уже предусмотрен такой встроенный предикат, `copy_term`.

7.5. Средства управления

До сих пор в этой книге фактически описана основная часть дополнительных средств управления, кроме `repeat`. Для полноты изложения в настоящем разделе представлен полный набор этих средств, которые описаны ниже.

- Оператор отсеечения, записываемый в программе как `!`, предотвращает перебор с возвратами. Он был представлен в главе 5. Еще одним полезным предикатом является `once( P )`, определяемый с помощью оператора отсеечения следующим образом:

```
once( P ) :- P, !.
```


Предикат `once( P )` вырабатывает только одно решение. Оператор отсечения, вложенный в предикат `once`, не предотвращает использование перебора с возвратами в других целях программы.

- `fail` — цель, которая никогда не достигается.
- `true` — цель, которая всегда достигается.
- `not( P )` — это отрицание как недостижение цели, которое действует в полном соответствии со следующим определением:
`not( P ) :- P, !, fail; true.`

Некоторые проблемы, связанные с использованием оператора отсечения и оператора `not`, подробно рассматривались в главе 5.

- Предикат `call( P )` вызывает цель `P`. Он выполняется успешно, если цель `P` достигается.
- `repeat` — это цель, которая всегда достигается. Ее особым свойством является то, что она недетерминирована, поэтому после каждого ее достижения по методу перебора с возвратом она генерирует еще одну, альтернативную ветвь выполнения. Цель `repeat` действует в соответствии со следующим определением:
`repeat.`
`repeat :- repeat.`

Типичный способ использования `repeat` иллюстрируется на приведенном ниже примере процедуры `dosquares`, которая считает последовательность чисел и выводит результаты возведения их в квадрат. Последовательность завершается атомом `stop`, который служит в качестве сигнала для останова этой процедуры.

```
dosquares :-
    repeat,
    read( X ),
    ( X = stop, !
    ;
    Y is X*X, write(Y),
    fail
    ).
```

7.6. Предикаты `bagof`, `setof` и `findall`

Выше было показано, что в программе можно сформировать по одному все объекты, соответствующие некоторой цели, с помощью перебора с возвратами. Но после выработки каждого следующего решения предыдущее удаляется и становится недоступным. Тем не менее иногда возникает необходимость иметь доступ сразу ко всем выработанным решениям, например, собранным в виде списка. Для этого применяются встроенные предикаты `bagof`, `setof` и `findall`.

Цель

```
bagof( X, P, L )
```

вырабатывает список `L`, состоящий из всех объектов `X`, таких, что цель `P` достигается. Как правило, применение такой конструкции имеет смысл, если `X` и `P` имеют некоторые общие переменные. Например, предположим, что в программе имеются следующие факты:

```
age( peter, 7 ).
age( ann, 5 ).
age( pat, 8 ).
age( tom, 5 ).
```

В таком случае может быть получен список всех детей в возрасте 5 лет с помощью следующей цели:

```
?- bagof( Child, age( Child, 5), List).
List = [ ann, tom]
```

Если в приведенной выше цели значение возраста не указано, то с помощью перебора с возвратами будут сформированы три списка детей, соответствующие трем значениям возраста:

```
?- bagof( Child, age( Child, Age), List).
Age = 7
List = [ peter];
Age = 5
List = [ ann, tom];
Age = 8
List = [ pat];
no
```

Может также потребоваться включить всех детей в один список, независимо от их возраста. Такую задачу можно решить, явно указав в вызове предиката `bagof` с помощью оператора “^”, что нас не интересует значение параметра `Age`; важно лишь то, что такое значение существует. Соответствующий вопрос может быть сформулирован таким образом:

```
?- bagof( Child, Age ^ age( Child, Age), List).
List = [ peter, ann, pat, tom]
```

По своей синтаксической форме знак операции “^” представляет собой заранее определенный инфиксный оператор типа `xfy`.

Если отсутствует решение для `P` в цели `bagof( X, P, L)`, то цель `bagof` не достигается. А если один и тот же объект `X` обнаруживается повторно, то в списке `L` появляются все его вхождения, что может вызвать присутствие в этом списке дублирующихся элементов.

Предикат `setof` аналогичен `bagof`. Цель

```
setof( X, P, L)
```

также вырабатывает список `L` объектов `X`, которые соответствуют предикату `P`. Но в данном случае список `L` упорядочивается, а дубликаты элементов, если они имеются, удаляются. Упорядочение объектов соответствует встроенному предикату `@<`, который определяет правила предшествования термов, например, следующим образом:

```
?- setof( Child, Age ^ age( Child, Age), ChildList),
   setof( Age, Child ^ age( Child, Age), AgeList).
ChildList = [ ann, pat, peter, tom]
AgeList = [ 5, 7, 8]
```

На типы объектов, собираемых в списки, не налагаются ограничения. Поэтому можно, например, сформировать список детей, отсортированный по их возрасту, составив пары в форме `Age/Child` таким образом:

```
?- setof( Age/Child, age( Child, Age), List)
List = [ 8/ann, 5/tom, 7/peter, 8/pat]
```

Еще одним предикатом из этого семейства, аналогичным `bagof`, является `findall`. Предикат

```
findall( X, P, L)
```

также вырабатывает список объектов, которые соответствуют предикату `P`. Предикат `findall` отличается от `bagof` тем, что в список собираются все объекты `X`, независимо от наличия (возможно) различных решений для переменных в предикате `P`, которые не являются общими с объектом `X`. Это различие демонстрируется на следующем примере:

```
?- findall( Child, age( Child, Age), List).
List = [ peter, ann, pat, tom]
```

Если не существует ни один объект `X`, который соответствует предикату `P`, то цель, заданная предикатом `findall`, достигается, создавая пустой список `L = []`.

Если в используемой реализации отсутствует `findall` как встроенный предикат, то его можно легко запрограммировать, как показано ниже. Все решения, соответствующие предикату `P`, вырабатываются с помощью принудительного перебора с возвратами. Каждое решение после его выработки немедленно вносится в базу данных, чтобы оно не было потеряно после обнаружения следующего решения. Вслед за тем, как будут выработаны и внесены все решения, их необходимо собрать в список и извлечь из базы данных. Весь этот процесс можно представить себе, как если бы все вырабатываемые решения формировали очередь. Каждое вновь выработанное решение с помощью предиката внесения `assert` добавляется к концу этой очереди, а после сбора всех решений очередь ликвидируется. Следует отметить, что конец этой очереди должен быть дополнительно обозначен атомом, позволяющим узнать, где находится конец очереди, например `"bottom"` (этот атом, безусловно, должен отличаться от любых решений, которые могут быть выработаны). Пример реализации `findall` в соответствии с этими рекомендациями приведен в листинге 7.3.

Листинг 7.3. Реализация отношения `findall`

```

findall( X, Goal, Xlist) :-
    call( Goal),                                % Найти решение
    assertz( queue(X) ),                        % Внести его в базу данных
    fail;                                       % Попытаться найти другие решения
    assertz( queue(bottom) ),                  % Отметить конец очереди решений
    collect( Xlist).                            % Собрать решения

collect( L) :-
    retract( queue(X) ), !,                    % Извлечь следующее решение
    ( X == bottom, !, L = []                  % Решения закончились?
    ;
    L = [X | Rest], collect( Rest) ).          % В противном случае собрать остальное

```

Упражнения

7.8. Примените процедуру `bagof` для определения отношения `powerset( Set, Subsets)`, которое вычисляет множество всех подмножеств заданного множества (все множества представлены в виде списков).

7.9. Примените процедуру `bagof` для определения отношения

`copy_term( Term, Copy)`

такого, что `Copy` представляет собой терм `Term` со всеми переименованными переменными.

Резюме

- В любой реализации Prolog обычно предусмотрен набор *встроенных процедур*, позволяющих осуществить несколько полезных операций, которые невозможны в базовой версии языка Prolog. В данной главе представлен подобный набор предикатов, доступных во многих реализациях Prolog.
- *Тип термина* можно проверить с помощью предикатов, перечисленных ниже.
 - `var( X). X` — (неконкретизированная) переменная.
 - `nonvar( X). X` — не переменная.
 - `atom( X). X` — атом.
 - `integer( X). X` — целое число.
 - `float( X). X` — действительное число.

- `atomic( X)`. *X* является либо атомом, либо числом.
 - `compound( X)`. *X* — структура.
- Для создания или декомпозиции термов могут применяться следующие предикаты:
- ```
Term =.. [Functor | ArgumentList]
functor(Term, Functor, Arity)
arg(N, Term, Argument)
name(Atom, CharacterCodes)
```
- Над термами могут выполняться описанные ниже операции сравнения.
- `X = Y`. *X* и *Y* согласуются.
  - `X == Y`. *X* и *Y* идентичны.
  - `X \== Y`. *X* и *Y* не идентичны.
  - `X := Y`. Арифметические значения *X* и *Y* равны.
  - `X \= Y`. Арифметические значения *X* и *Y* не равны.
  - `X < Y`. Арифметическое значение *X* меньше *Y* (к этому же типу относятся операторы `=<`, `>`, `>=`).
  - `X @< Y`. Терм *X* предшествует терму *Y* (к этому же типу относятся операторы `@=<`, `@>`, `@>=`).
- Программу Prolog можно рассматривать как реляционную базу данных, для обновления которой предусмотрены описанные ниже процедуры.
- `assert( Clause)`. Добавление предложения *Clause* к программе.
  - `asserta( Clause)`. Добавление в начале.
  - `assertz( Clause)`. Добавление в конце.
  - `retract( Clause)`. Удаление предложения, которое согласуется с предложением *Clause*.
- Все объекты, соответствующие заданному условию, могут быть собраны в список с помощью описанных ниже предикатов.
- `bagof( X, P, L)`. *L* — список из всех *X*, которые соответствуют условию *P*.
  - `setof( X, P, L)`. *L* — отсортированный список из всех *X*, которые соответствуют условию *P*.
  - `findall( X, P, L)`. Аналогичен `bagof`.
- `repeat` — средство управления, которое вырабатывает неограниченное количество вариантов для перебора с возвратами.

# Стиль и методы программирования

*В этой главе...*

|                                                    |     |
|----------------------------------------------------|-----|
| 8.1. Общие принципы качественного программирования | 169 |
| 8.2. Общий подход к разработке программ Prolog     | 171 |
| 8.3. Стиль программирования                        | 173 |
| 8.4. Отладка                                       | 176 |
| 8.5. Повышение эффективности                       | 177 |

В этой главе рассматриваются некоторые общие принципы качественного программирования. В ней особое внимание уделено следующим вопросам: “Какой подход должен применяться к разработке программ Prolog? Каковы элементы хорошего стиля программирования на языке Prolog? Как отлаживать программы Prolog? Как обеспечить повышение эффективности программ Prolog?”

## 8.1. Общие принципы качественного программирования

Что такое хорошая программа? Ответ на этот вопрос не так прост, поскольку есть целый ряд критериев, на основании которых можно судить, насколько качественной является программа. Ниже перечислены наиболее широко применяемые критерии.

- **Правильность.** Прежде всего, хорошая программа должна быть правильной. Иными словами, она должна выполнять те функции, для которых предназначена. Такое требование может показаться тривиальным и не заслуживающим внимания. Но в случае сложных программ правильность часто не обеспечивается. Одна из распространенных ошибок при написании программ состоит в том, что этот очевидный критерий игнорируется и основное внимание уделяется другим критериям, таким как эффективность или внешняя привлекательность программы.
- **Дружественность.** Хорошая программа должна быть удобной для использования и взаимодействия с ней.
- **Эффективность.** Хорошая программа не должна бесполезно расходовать такие ресурсы компьютера, как процессорное время и объем памяти.
- **Удобство для чтения.** Хорошая программа должна быть удобной для чтения и простой для понимания. Она не должна быть более сложной, чем необходимо. Следует избегать использования остроумных программистских приемов, которые затемняют смысл программы. Удобство программы для чтения во многом зависит от ее общей организации и компоновки.

- **Модифицируемость.** Хорошая программа должна быть удобной для модификации и дополнения. Программа становится более модифицируемой при использовании наглядной и модульной организации.
- **Надежность.** Хорошая программа должна быть надежной. Ее выполнение не должно завершаться аварийно сразу же после ввода пользователем каких-либо неправильных или непредвиденных данных. В случае подобных ошибок программа должна оставаться действующей и реагировать соответствующим образом (например, сообщать об ошибках).
- **Документированность.** Хорошая программа должна быть соответствующим образом документирована. Минимальным объемом документации является листинг программы, включающий достаточное количество комментариев к программе.

Важность того или иного критерия зависит от решаемой задачи и от того, в каких обстоятельствах разработана программа, а также от того, в какой среде она используется. Нет ни малейшего сомнения в том, что требование к правильности имеет наивысший приоритет. Таким задачам, как обеспечение удобства для чтения, дружелюбности, модифицируемости, надежности и наличия документации, обычно придается, по крайней мере, не меньший приоритет, чем задаче обеспечения эффективности.

В данной главе приведены некоторые общие рекомендации по достижению на практике описанных выше критериев. Одно из важных правил состоит в том, что вначале нужно обдумать решаемую задачу и только после этого приступить к фактическому написанию кода на используемом языке программирования. После достижения достаточного уровня понимания задачи и формулировки тщательно продуманного решения фактическое написание кода становится быстрым и легким, поэтому повышается вероятность того, что вскоре будет составлена правильная программа.

Широко распространенная ошибка состоит в том, что написание кода начинается еще до полного понимания постановки задачи. Основной причиной того, почему преждевременное начало процесса написания кода не рекомендуется, состоит в том, что размышление о поставленной задаче и поиск идей для ее решения должны осуществляться в терминах, в наибольшей степени соответствующих данной задаче. Но эти термины обычно не имеют ничего общего с синтаксисом используемого языка программирования и могут включать утверждения на естественном языке и графические иллюстрации к рассматриваемым понятиям.

Подобная формулировка решения должна быть переведена на язык программирования, но сам процесс перевода может оказаться нелегким. Один из удобных подходов состоит в использовании принципа поэтапного усовершенствования. В этом случае первоначальная формулировка решения рассматривается как “решение верхнего уровня”, а окончательная программа — как “решение нижнего уровня”.

В соответствии с принципом поэтапного усовершенствования окончательная программа разрабатывается в результате применения последовательности преобразований, или “усовершенствований” первоначального решения. Работа начинается с первого варианта решения, относящегося к верхнему уровню, а затем продолжается как ряд решений; все они являются эквивалентными, но каждое решение в этой последовательности представлено с большей степенью детализации. На каждом этапе усовершенствования концепции, использовавшиеся в предыдущих формулировках, прорабатываются более подробно, и их представление становится ближе к применяемому языку программирования. Но следует учитывать, что усовершенствования относятся и к определениям процедур, и к структурам данных. На начальных этапах обычно приходится иметь дело с более абстрактными, крупными единицами информации, структура которых уточняется позже.

Такая стратегия нисходящего поэтапного усовершенствования программы имеет следующие преимущества:

- она допускает использование приблизительных формулировок решений в терминах, наиболее соответствующих рассматриваемой задаче;

- в терминах подобных мощных понятий решение обычно выражается кратко и просто, поэтому чаще всего оказывается правильным;
- каждый этап усовершенствования может быть небольшим для того, чтобы было достаточно интеллектуальных ресурсов для его осуществления; в таком случае преобразование текущего решения в новое, более подробное представление, по всей вероятности, должно быть правильным, и таким же будет результирующее решение на следующем уровне детализации.

В случае языка Prolog речь может идти о поэтапном *усовершенствовании отношений*. Если задача требует решения алгоритмическими способами, можно также рассуждать об *усовершенствовании алгоритмов* исходя из процедурной трактовки языка Prolog.

Для того чтобы правильно уточнить решение на некотором уровне детализации и ввести полезные концепции на следующем, более низком уровне, нужны идеи. Поэтому процесс программирования является творческим, особенно для начинающих. С накоплением опыта программирование постепенно превращается из искусства в мастерство. Но всегда остается актуальным основной вопрос о том, где найти нужные идеи. Большинство идей приходит с опытом, приобретенным в результате успешного решения аналогичных задач. А если непосредственное программное решение неизвестно, может помочь изучение подобной задачи. Еще одним источником идей является повседневная жизнь. Например, если задача состоит в написании программы сортировки списка элементов, идея может возникнуть в результате поиска ответа на такой вопрос: “Как я сам сортирую комплект экзаменационных бумаг в соответствии с алфавитным порядком фамилий студентов?”

Общие принципы качественного программирования, кратко описанные в этом разделе, в основном распространяются и на язык Prolog. Некоторые дополнительные сведения, имеющие непосредственное отношение к языку Prolog, представлены в следующих разделах.

## 8.2. Общий подход к разработке программ Prolog

Одна из характерных особенностей языка Prolog состоит в том, что он допускает использование и процедурного, и декларативного подхода к разработке программ. Эти два подхода были подробно описаны в главе 2 и иллюстрировались на примерах, представленных в данной книге. Какой из этих подходов является наиболее эффективным и практичным, зависит от рассматриваемой проблемы. Декларативные решения обычно являются более простыми в реализации, но могут привести к созданию неэффективных программ.

В процессе разработки решения необходимо найти способы, позволяющие свести задачу к одной или нескольким более простым подзадачам. При этом один из важных вопросов состоит в том, как найти приемлемые подзадачи. В программировании на языке Prolog часто может успешно применяться несколько общих принципов, которые рассматриваются в следующих разделах.

### 8.2.1. Использование рекурсии

Этот принцип состоит в том, что задача сводится к нескольким случаям, принадлежащим к двум группам.

1. Тривиальные, или *граничные*, случаи.
2. *Общие* случаи, в которых решение составляется из решений отдельных (более простых) вариантов первоначальной задачи.

Этот метод применяется в языке Prolog постоянно. Рассмотрим еще один пример: обработка списка элементов таким образом, чтобы преобразование каждого элемента

осуществлялось с помощью одного и того же правила преобразования. Предположим, что рассматривается следующая процедура:

```
maplist(List, F, NewList)
```

где List — первоначальный список, F — правило преобразования (бинарное отношение) и NewList — список всех преобразованных элементов. Задачу преобразования списка List можно свести к двум случаям, описанным ниже.

**1. Граничный случай:** List = [].

Если List = [], то NewList = [], независимо от F

**2. Общий случай:** List = [X | Tail].

Чтобы преобразовать список, представленный в форме [X | Tail], необходимо выполнить следующие действия:

преобразовать элемент X в соответствии с правилом F, получив NewX, и преобразовать список Tail, получив NewTail;  
весь преобразованный список представляет собой [NewX | NewTail].

На языке Prolog эта формулировка может быть представлена следующим образом:

```
maplist([], _, []).
maplist([X | Tail], F, [NewX | NewTail]) :-
 G =.. [F, X, NewX],
 call(G),
 maplist(Tail, F, NewTail).
```

Предположим, что имеется список чисел и требуется получить список квадратов этих чисел. Процедура maplist может использоваться для решения этой задачи следующим образом:

```
square(X, Y) :-
 Y is X*X.
?- maplist([2, 6, 5], square, Squares).
Squares = [4, 36, 25]
```

Одна из причин, по которым рекурсия так естественно подходит для определения отношений в языке Prolog, состоит в том, что объекты данных часто сами имеют рекурсивную структуру. В частности, к категории подобных объектов относятся списки и деревья. Список является либо пустым (граничный случай), либо имеет голову и хвост, который сам является списком (общий случай). Бинарное дерево является либо пустым (граничный случай), либо имеет корень и два поддерева, которые сами являются бинарными деревьями (общий случай). Поэтому для обработки всего непустого дерева необходимо выполнить определенные операции с корнем, а затем перейти к обработке поддеревьев.

## 8.2.2. Обобщение

Часто бывает целесообразно обобщить первоначальную задачу, чтобы получить возможность сформулировать рекурсивное решение обобщенной задачи. В таком случае первоначальная задача решается как частный случай своей более общей версии. Для обобщения отношения обычно требуется ввести один или несколько дополнительных параметров. Основная проблема, которая может потребовать более глубокого анализа задачи, состоит в том, что нужно найти правильный способ обобщения.

В качестве примера снова вернемся к задаче с восемью ферзями. Первоначальная задача состояла в том, чтобы расставить на шахматной доске восемь ферзей таким образом, чтобы они не нападали друг на друга. Определим соответствующее отношение:

```
eightqueens(Pos)
```

Оно принимает истинное значение, если Pos — позиция с восемью не нападающими друг на друга ферзями. Продуктивная идея в данном случае состоит в том, что нужно обобщить количество ферзей и вместо восьми рассматривать N. В таком случае количество ферзей становится дополнительным параметром:

```
nqueens(Pos, N)
```



Преимущество такого обобщения состоит в том, что может быть сразу же получена рекурсивная формулировка отношения `nqueens`, приведенная ниже.

1. Граничный случай:  $N = 0$ .

Задача безопасного размещения нуля ферзей является тривиальной.

2. Общий случай:  $N > 0$ .

Чтобы можно было безопасно расставить на шахматной доске  $N$  ферзей, необходимо выполнить следующие условия:

- добиться получения безопасной конфигурации, состоящей из  $(N - 1)$  ферзей;
- поставить на доску последнего ферзя таким образом, чтобы он не нападал на всех прочих ферзей.

После решения обобщенной задачи первоначальная задача решается просто:

```
eightqueens(Pos) :- nqueens(Pos, 8).
```

### 8.2.3. Использование графических схем

При поиске идей для решения задачи часто бывает полезно подготовить некоторые графические схемы, наглядно представляющие данную задачу. Рисунок может помочь понять некоторые важные зависимости, которые относятся к рассматриваемой задаче. После этого достаточно просто описать на языке программирования то, что мы видим на рисунке.

В целом использование графических иллюстраций часто бывает полезным при решении любых задач, но создается впечатление, что такой подход является особенно продуктивным при разработке программ на языке Prolog. Ниже описаны основные причины, позволяющие понять, с чем это связано.

1. Язык Prolog особенно хорошо подходит для решения задач, в которых рассматриваются объекты и отношения между ними. Чаще всего подобные задачи могут быть естественным образом проиллюстрированы с помощью графов, узлы которых соответствуют объектам, а дуги — отношениям.
2. Структурированные объекты данных, применяемые в языке Prolog, могут быть естественным образом представлены в виде деревьев.
3. Декларативное значение программ Prolog упрощает преобразование графических представлений в синтаксические структуры Prolog, поскольку порядок, в котором объекты представлены на рисунке, фактически не имеет значения. Поэтому достаточно просто ввести в программу в любом порядке все, что мы видим на рисунке. (Но существует возможность в дальнейшем уточнить этот порядок по практическим соображениям повышения эффективности программы.)

## 8.3. Стиль программирования

При разработке программы необходимо придерживаться определенных соглашений по стилю, которые позволяют достичь следующих целей:

- уменьшить вероятность ошибок программирования;
- разрабатывать программы, удобные для чтения и понимания, простые в отладке и модификации.

В данном разделе рассматриваются некоторые составляющие хорошего стиля программирования на языке Prolog; к ним относятся определенные общие правила хорошего стиля, табличная организация длинных процедур и комментирование.

### 8.3.1. Некоторые правила хорошего стиля

Основные правила хорошего стиля программ на языке Prolog приведены ниже.

- Предложения программы должны быть короткими. Тело предложения, как правило, должно содержать лишь несколько целей.
- Процедуры должны быть небольшими, поскольку длинные процедуры являются сложными для понимания. Но допускается применение длинных процедур, если они имеют некоторую единообразную структуру (эта тема рассматривается более подробно ниже в данном разделе).
- Для процедур и переменных должны использоваться мнемонические имена. Имена должны подчеркивать смысл отношений и роль объектов данных.
- Важное значение имеет компоновка программ. Для повышения удобства чтения следует неизменно применять определенные правила расстановки пробелов, ввода пустых строк и отступов. Предложения, касающиеся одной и той же процедуры, должны быть собраны вместе; между предложениями должны находиться пустые строки (возможно, за исключением того случая, когда они представляют собой многочисленные факты, касающиеся одного и того же отношения); каждая цель может быть помещена на отдельной строке. Программы Prolog иногда напоминают стихи благодаря эстетической привлекательности заложенных в них идей и форм.
- Применяемые стилистические соглашения такого рода могут зависеть от конкретной программы, поскольку их выбор диктуется рассматриваемой задачей и личным вкусом. Но важно то, чтобы одни и те же соглашения неизменно использовались во всей программе.
- Оператор отсечения должен использоваться с осторожностью. Эти операторы не следует применять в тех случаях, если без них можно легко обойтись. По возможности, лучше использовать *зеленые операторы отсечения*, а не *красные операторы отсечения*. Как было описано в главе 5, оператор отсечения называется *зеленым*, если после его удаления декларативное значение предложения не изменяется. Использование *красных операторов отсечения* должно быть ограничено такими четко определенными конструкциями, как `not` или выбор между несколькими взаимоисключающими вариантами. Ниже приведен пример конструкции последнего типа.

если `Condition`, то `Goal1`, иначе `Goal2`

Ее можно перевести на язык Prolog с помощью операторов отсечения следующим образом:

```
Condition, !, % Условие Condition является истинным?
Goal1 % Если да, то Goal1,
;
Goal2 % иначе Goal2
```

- Процедура `not` также может стать причиной выполнения программой действий, неожиданных для пользователя, поскольку она тесно связана с оператором отсечения, поэтому необходимо иметь полное представление о том, как процедура `not` определена в языке Prolog. Тем не менее, если нужно сделать выбор между процедурой `not` и оператором отсечения, то применение первой часто бывает более оправданным, чем создание каких-то непонятных конструкций с оператором отсечения.
- Модификация программы с помощью предикатов `assert` и `retract` способна значительно затруднить понимание поведения программы. В частности, при использовании этих предикатов может оказаться, что одна и та же программа в разное время отвечает на одни и те же вопросы по-разному. Если в подобных случаях необходимо воспроизвести такое же поведение, как и прежде, то следует обеспечить полное восстановление предыдущего состояния программы,

которая была модифицирована в результате внесения и извлечения предложений из базы данных с применением этих предикатов.

- В результате использования точки с запятой смысл предложения может стать менее очевидным; иногда удобство программ для чтения может быть повышено путем разделения предложения, содержащего точку с запятой, на несколько предложений. Но возможно также, что это улучшение будет достигнуто за счет увеличения длины программы и снижения ее эффективности.

В качестве иллюстрации к нескольким рекомендациям, изложенным в данном разделе, рассмотрим следующее отношение:

```
merge(List1, List2, List3)
```

где List1 и List2 — упорядоченные списки, которые сливаются в список List3, например:

```
merge([2,4,7], [1,3,4,8], [1,2,3,4,4,7,8])
```

Ниже приведен пример реализации процедуры merge, выполненный в плохом стиле.

```
merge(List1, List2, List3) :-
 List1 = [],!, List3 = List2; % Первый список пуст
 List2 = [],!, List3 = List1; % Второй список пуст
 List1 = [X | Rest1],
 List2 = [Y | Rest2],
 (X < Y, !,
 Z = X, % Z - голова списка List3
 merge(Rest1, List2, Rest3);
 Z = Y,
 merge(List1, Rest2, Rest3)),
 List3 = [Z | Rest3].
```

А ниже представлена улучшенная версия этой процедуры, в которой исключены точки с запятой.

```
merge([], List, List) :-
 !. % Исключает выработку лишних решений
merge(List, [], List).
merge([X | Rest1], [Y | Rest2], [X | Rest3]) :-
 X < Y, !, merge(Rest1, [Y | Rest2], Rest3).
merge(List1, [Y | Rest2], [Y | Rest3]) :-
 merge(List1, Rest2, Rest3).
```

## 8.3.2. Табличная организация длинных процедур

Длинные процедуры являются приемлемыми, если в них используется некоторая единообразная структура. Как правило, такая форма организации процедуры представляет собой набор фактов, с помощью которого отношение фактически определяется в табличной форме. Преимущества такой организации длинной процедуры перечислены ниже.

- Структуру этой процедуры легко понять.
- Существует возможность дополнить процедуру; для этого достаточно ввести новые факты.
- Процедуру можно легко проверить и исправить либо модифицировать (заменяя некоторый факт независимо от других фактов).

## 8.3.3. Комментирование

Комментарии в программе должны прежде всего пояснять, для чего предназначена программа и как ее использовать, и только после этого представлять подробные сведения об используемом методе решения и других нюансах программирования. Ос-

новное назначение комментариев состоит в том, чтобы предоставить пользователю возможность эксплуатировать программу, понять ее и, возможно, модифицировать. Комментарии должны описывать в наиболее краткой из возможных форм все, что для этого необходимо.

Широко распространенной ошибкой является недостаточное применение комментариев, но в программе может также оказаться слишком много комментариев. Изложение сведений, очевидных при изучении кода самой программы, создает лишь ненужную нагрузку для тех, кто изучает эту программу.

Продолжительные комментарии должны предшествовать коду, к которому они относятся, а короткие комментарии следует чередовать с самим кодом. Ниже показано, какая информация в целом должна содержаться в комментариях.

- Назначение программы, способы ее использования (например, какая цель должна быть вызвана и каковы ожидаемые результаты), а также примеры использования.
- Предикаты верхнего уровня.
- Способы представления основных понятий (объектов).
- Время выполнения и требования программы к памяти.
- Основные ограничения программы.
- Все специальные средства, зависящие от системы.
- Назначение предикатов в программе; параметры этих предикатов; обозначения входных и выходных параметров, если известно, к какому типу они относятся. (*Входными* называются такие параметры, которые при вызове предиката всегда имеют полностью заданные значения, без неконкретизированных переменных.)
- Сведения об используемых алгоритмах и способах их реализации.

При описании предикатов часто применяются такое соглашение: в качестве ссылки на предикат используются имя предиката и его *арность* (количество формальных параметров), как показано ниже.

PredicateName/Arity

Например, ссылка на предикат `merge(List1, List2, List3)` может быть представлена как `merge/3`. Типы входных/выходных параметров обозначаются путем указания перед именами параметров префикса “+” (входной) или “-” (выходной). Например, обозначение `merge(+List1, +List2, -List3)` указывает, что первые два параметра `merge` являются входными, а третий — выходным.

## 8.4. Отладка

Если программа не действует в соответствии с ожидаемым, основная проблема состоит в том, как найти в ней ошибку (ошибки). Проще найти ошибку в отдельной части программы (или в модуле), чем во всей программе в целом. Поэтому удобным принципом отладки является проверка вначале меньших модулей программы, а затем — проверка более крупных модулей или всей программы.

Отладка программ на языке Prolog упрощается благодаря следующим двум особенностям: во-первых, Prolog является интерактивным языком, поэтому любую часть программы можно вызвать непосредственно с помощью подходящего вопроса к системе Prolog; во-вторых, в реализациях Prolog обычно предусмотрены специальные средства отладки. Благодаря наличию этих двух особенностей отладка программ Prolog может в целом осуществляться более эффективно по сравнению с большинством других языков программирования.

Основой средств отладки является трассировка. Выражение *трассировка цели* означает, что во время выполнения отображается информация, относящаяся к процессу достижения цели. Эта информация включает данные, описанные ниже.

- **Вступительная информация.** Имя предиката и значения параметров при вызове цели.
- **Заключительная информация.** В случае успеха — значения параметров, которые обеспечили достижение цели; в противном случае — сообщение о неудаче.
- **Информация о повторных вызовах.** Вызовы той же цели, обусловленные перебором с возвратами.

Между вступительной и заключительной информацией может быть представлена информация трассировки для всех подцелей этой цели. Поэтому есть возможность проследить за выполнением всей процедуры поиска ответа на вопрос вплоть до целей самого низкого уровня, когда были обнаружены факты. Может оказаться, что такая подробная трассировка практически нецелесообразна из-за чрезмерного объема информации трассировки, поэтому пользователь может указать, что требуется избирательная трассировка. Для обеспечения избирательности предусмотрены два основных механизма: во-первых, подавление информации трассировки, выходящей за определенные рамки; во-вторых, трассировка не всех предикатов, а лишь некоторого заданного подмножества предикатов.

Указанные средства отладки активизируются с помощью встроенных предикатов, зависящих от системы. Типичное подмножество таких предикатов описано ниже. Предикат

`trace`

активизирует исчерпывающую трассировку целей, которые будут выполняться в дальнейшем. Предикат

`notrace`

прекращает дальнейшую трассировку. Предикат

`spy( P)`

указывает, что необходимо выполнить трассировку предиката *P*. Он используется, если указанный предикат представляет особый интерес и необходимо избежать появления информации трассировки, которая относится к другим целям (находящимся либо выше, либо ниже уровня вызова предиката *P*). С помощью предиката `spy` можно активизировать “отслеживание” одновременно нескольких предикатов. Предикат

`nospy( P)`

прекращает “отслеживание” предиката *P*.

Для подавления трассировки ниже заданного уровня могут применяться специальные команды во время выполнения программы. Кроме того, может быть предусмотрено несколько других команд отладки, таких как команды возврата к предыдущей точке выполнения. Такая операция возврата позволяет, например, повторить выполнение участка программы, применяя трассировку с большей степенью детализации.

## 8.5. Повышение эффективности

Для оценки эффективности могут применяться разные характеристики, но чаще всего рассматриваются продолжительность выполнения и требования программы к памяти. Еще одной важной характеристикой эффективности является время, которое потребовалось программисту на разработку программы.

Традиционная компьютерная архитектура не слишком хорошо приспособлена для выполнения программ в стиле Prolog, которое состоит в достижении списка целей. Поэтому в языке Prolog приходится чаще сталкиваться с такими ограничениями,

как время и пространство, чем во многих других языках программирования. Будет ли это вызывать сложности при практической реализации, зависит от самой задачи. Проблема, связанная с низкой эффективностью по времени, является практически несущественной, даже если программа Prolog требует 1 секунду процессорного времени, а соответствующая программа на некотором другом языке, скажем Fortran, выполняется за 0,1 секунды, при условии, что эта программа применяется лишь несколько раз в сутки. Но такое различие в эффективности становится существенным, если две программы, соответственно, выполняются в течение 50 и 5 минут.

С другой стороны, во многих областях применения язык Prolog намного сокращает время разработки программ. Программы Prolog, как правило, являются более легкими для написания, понимания и отладки по сравнению с программами, написанными на традиционных языках программирования. К проблемам, относящимся к области применения языка Prolog, относятся символическая, нечисловая обработка, структурированные объекты данных и отношения между ними. В частности, язык Prolog успешно применялся в таких областях, как алгебраическое решение уравнений, планирование, использование баз данных, решение общих проблем, разработка прототипов, реализация языков программирования, дискретное и качественное моделирование, строительное проектирование, машинное обучение, понимание естественного языка, экспертные системы и другие области искусственного интеллекта. Однако числовые математические расчеты — это область, для которой Prolog подходит не столь естественно.

Что касается вычислительной эффективности, то выполнение откомпилированной программы в целом происходит намного быстрее по сравнению с интерпретацией программы. Поэтому, если система Prolog включает и интерпретатор, и компилятор, то при наличии жестких требований к быстродействию необходимо использовать компилятор.

Если программа характеризуется крайне низким быстродействием, ее работу часто можно намного ускорить путем усовершенствования реализации самого алгоритма. Но для этого необходимо изучить процедурные аспекты данной программы. Простым способом повышения вычислительной эффективности является поиск лучшего упорядочения предложений процедур и целей в телах предложений. Еще одним относительно простым методом является предоставление системе Prolog указаний с помощью операторов отсечения.

Идеи по повышению эффективности программы обычно рождаются в результате более глубокого понимания задачи. Создание более эффективного алгоритма, как правило, может стать результатом усовершенствований двух типов, описанных ниже.

- Повышение эффективности поиска путем предотвращения ненужного перебора с возвратами и останова процесса обработки ненужных альтернатив на самых ранних этапах.
- Использование для представления объектов в программе более подходящих структур данных, для того, чтобы можно было реализовать операции над объектами более эффективно.

В данной главе усовершенствования обоих типов рассматриваются на примерах. Кроме того, на примере будет показан еще один метод повышения эффективности. Этот метод, называемый *кэшированием*, основан на вставке в базу данных промежуточных результатов, которые, по всей вероятности, снова потребуются при проведении дальнейших вычислений. Вместо повторения вычислений можно выполнить выборку подобных результатов как уже известных фактов.

## 8.5.1. Повышение эффективности программы решения задачи с восемью ферзями

В качестве простого примера повышения эффективности поиска еще раз рассмотрим задачу с восемью ферзями (см. листинг 4.2). В этой программе координаты Y

ферзей отыскиваются путем последовательной проверки для каждого ферзя целых чисел от 1 до 8. Такая процедура программируется в виде следующей цели:

```
member(Y, [1,2,3,4,5,6,7,8])
```

Принцип действия отношения `member` состоит в том, что вначале проверяется значение  $Y = 1$ , затем  $Y = 2$ ,  $Y = 3$  и т.д. Но изучение процедуры, согласно которой осуществляются попытки поместить ферзей одного за другим на смежных вертикальных рядах доски, показывает, что такой порядок выполнения попыток является не самым подходящим. Причина этого состоит в том, что ферзи, стоящие на смежных вертикальных рядах, нападают друг на друга, если они не размещаются с интервалом, по меньшей мере, в две клетки в вертикальном направлении. В соответствии с этим наблюдением простая попытка повышения эффективности состоит в переупорядочении значений координат, применяемых в качестве потенциальных кандидатов, например, следующим образом:

```
member(Y, [1,5,2,6,3,7,4,8])
```

Это небольшое изменение позволяет сократить время, необходимое для поиска первого решения, в 3–4 раза.

В следующем примере показано, что аналогичная простая идея переупорядочения позволяет преобразовать задачу с такой сложностью, при которой время ее решения является практически неприемлемым, в тривиальную задачу.

## 8.5.2. Повышение эффективности программы раскраски карты

Задача раскраски карты состоит в том, что каждому государству на заданной карте должен быть присвоен один из четырех указанных цветов таким образом, чтобы ни в одной из пар соседних государств не использовался одинаковый цвет. В математике доказана теорема, которая гарантирует, что такое решение всегда возможно.

Предположим, что карта задана с помощью следующего отношения с описанием соседних государств:

```
ngb(Country, Neighbours)
```

где `Neighbours` — список государств, граничащих с государством `Country`. Поэтому карта Европы, в которую входит 30 государств, может быть задана (в алфавитном порядке) следующим образом:

```
ngb(albania, [greece, macedonia, yugoslavia]).
ngb(andorra, [france, spain]).
ngb(austria, [czech_republic, germany, hungary, italy, liechtenstein,
slovakia, slovenia, switzerland]).
...
```

Предположим, что решение должно быть представлено в виде списка пар в форме `Country/Colour`

которая определяет цвет для каждого государства на указанной карте. Для заданной карты названия государств определены заранее, и задача состоит в поиске значений для цветов. Таким образом, для карты Европы задача состоит в поиске подходящей конкретизации переменных `C1`, `C2`, `C3` и т.д. в следующем списке:

```
[albania/C1, andorra/C2, austria/C3, ...]
```

Предположим, что определен предикат

```
colours(Country_colour_list)
```

который принимает истинное значение, если список `Country_colour_list` соответствует ограничению по раскраске карты применительно к указанному отношению `ngb`. Предположим, что в качестве четырех цветов выбраны желтый, синий, красный и зеленый. Условие, согласно которому ни одно из двух соседних государств не должно быть окрашено на карте в одинаковый цвет, можно сформулировать на языке Prolog следующим образом:

```
colours([]).
colours([Country/Colour | Rest]) :-
 colours(Rest),
 member(Colour, [yellow, blue, red, green]),
 not(member(Country1 /Colour, Rest), neighbour(Country, Country1)).

neighbour(Country, Country1) :-
 ngb(Country, Neighbours),
 member(Country1, Neighbours).
```

В этой программе отношение `member( X, L)`, как обычно, обеспечивает проверку принадлежности к списку. Приведенная выше программа хорошо работает при наличии простых карт с небольшим количеством государств. Но при обработке карты Европы могут возникнуть проблемы. При условии, что доступен встроенный предикат `setof`, одна попытка раскрасить карту Европы может состоять в следующем. Вначале определим вспомогательное отношение:

```
country(C) :- ngb(C, _).
```

После этого вопрос для определения раскраски карты Европы можно сформулировать следующим образом:

```
?- setof(Cntry/Colour, country(Cntry), CountryColourList),
 colours(CountryColourList).
```

Цель `setof` формирует образец списка *государство/цвет* (`CountryColourList`) для карты Европы, в котором неконкретизированные переменные будут обозначать цвета. Предполагается, что после этого цель `colours` позволит конкретизировать цвета. Но такая попытка, скорее всего, окончится неудачей из-за низкой эффективности программы.

Подробное изучение способа, с помощью которого система Prolog пытается достичь цели `colours`, обнаруживает причину неэффективности. Государства в списке государств/цветов расположены в алфавитном порядке, который не имеет ничего общего с их географическим местонахождением. Порядок назначения цветов государствам соответствует их последовательности в списке (начиная с конца), которая в данном случае не зависит от отношения `ngb`. Поэтому процесс назначения цветов начинается в какой-то одной части карты, переходит к совсем иной ее части и т.д.; при этом передвижение по карте происходит в основном случайным образом. Это может вполне приводить к таким ситуациям, в которых государство, стоящее в очереди на раскраску, окружено многими другими государствами, уже окрашенными во все четыре возможных цвета. Поэтому возникает необходимость использовать перебор с возвратами, что приводит к снижению эффективности.

Поэтому очевидно, что эффективность программы зависит от того, в какой последовательности раскрашиваются государства. Интуиция подсказывает следующую простую стратегию раскраски, которая должна быть лучше по сравнению со случайной: начать с некоторого государства, имеющего много соседей, после этого перейти к его соседям, затем к соседям соседей и т.д. Поэтому при раскраске карты Европы лучше всего начать с Германии (которая имеет больше всего соседей). Безусловно, при формировании списка государств/цветов Германию необходимо поместить в конец списка, а другие государства добавлять в начало списка. Благодаря этому алгоритм раскраски, запуск которого происходит с конца списка, начнет свою работу с Германии и будет проходить по списку от одного соседнего государства к другому.

Такой образец списка государств/цветов позволяет существенно повысить эффективность по сравнению с первоначальным, алфавитным порядком, и теперь возможные варианты раскраски для карты Европы вырабатываются без каких-либо сложностей.

Упорядоченный должным образом список государств может быть сформирован вручную, но этого делать не стоит. Такую задачу позволяет решить приведенная ниже процедура `makelist`. Эта процедура начинает формирование списка с некоторого указанного государства (в данном случае с Германии) и собирает государства в список, называемый закрытым (`Closed`). Вначале каждое государство помещается в другой список, называемый открытым (`Open`), и только после этого переносится в список



Closed. После того как каждое государство переносится из списка Open в список Closed, в список Open добавляются его соседи.

```
makelist(List) :-
 collect([germany], [], List).

collect([], Closed, Closed). % Больше нет кандидатов на включение
 % в список Closed

collect([X | Open], Closed, List) :-
 member(X, Closed), !, % Элемент X уже внесен в список Closed?
collect(Open, Closed, List). % Отбросить элемент X

collect([X | Open], Closed, List) :-
 nbg(X, Ngbs), % Найти соседей элемента X
 conc(Ngbs, Open, Open1), % Поместить их в список Open1
 collect(Open1, [X | Closed], List). % Внести в список остальные элементы
```

Отношение conc, как обычно, представляет собой отношение для конкатенации списков.

### 8.5.3. Повышение эффективности конкатенации списков с использованием разностных списков

В программах, применяемых до сих пор, алгоритм конкатенации списков был запрограммирован следующим образом:

```
conc([], L, L).
conc([X | L1], L2, [X | L3]) :-
 conc(L1, L2, L3).
```

Такая программа является неэффективной, если первый список имеет большую длину. Следующий пример показывает, с чем это связано:

```
?- conc([a,b,c], [d,e], L).
```

Этот вопрос вырабатывает следующую последовательность целей:

```
conc([a,b,c], [d,e], L)
conc([b,c], [d,e], L'), где L = [a | L']
conc([c], [d,e], L''), где L' = [b | L'']
conc([], [d,e], L'''), где L'' = [c | L''']
true, где L''' = [d,e]
```

Из этого становится ясно, что данная программа, по сути, постоянно просматривает первый список до тех пор, пока не встретится пустой список.

Но нельзя ли на первом этапе вместо постепенной проработки первого списка пропустить весь первый список и добавить к нему второй список? Для этого необходимо знать, где находится конец списка; иными словами, требуется другое представление для списков. Одно из решений состоит в использовании структуры данных, называемой *разностными списками*. При этом список представляется в виде пары списков. Например, список

```
[a,b,d]
```

может быть представлен с помощью следующих двух списков:

```
L1 = [a,b,c,d,e]
L2 = [d,e]
```

Такая пара списков, которая сокращенно обозначается как L1-L2, представляет “разницу” между L1 и L2. Безусловно, подобная конструкция может применяться лишь при условии, что список L2 — суффикс списка L1. Следует отметить, что один и тот же список может быть представлен с помощью нескольких разностных пар. Поэтому, например, список [a, b, c] может быть представлен следующим образом:

```
[a,b,c]-[]
```

или

```
[a,b,c,d,e] - [d,e]
```

или

$[a,b,c,d,e \mid T] - [d,e \mid T]$

или

$[a,b,c \mid T] - T$

где  $T$  — любой список. Пустой список представляется с помощью любой пары в форме  $L-L$ .

Поскольку второй член пары обозначает конец списка, этот конец становится непосредственно доступным. Поэтому разностные списки могут использоваться для эффективной реализации конкатенации. Такой метод иллюстрируется на рис. 8.1. Соответствующее отношение конкатенации можно перевести на язык Prolog как следующий факт:

`concat( A1 - Z1, Z1 - Z2, A1 - Z2 ).`

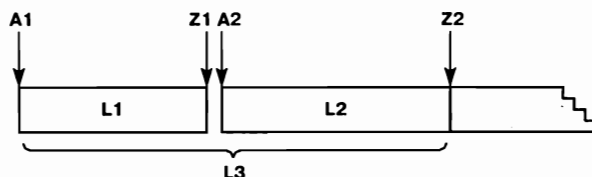


Рис. 8.1. Конкатенация списков, представленных разностными парами; список  $L1$  представлен как  $A1-Z1$ ,  $L2$  — как  $A2-Z2$ , а результат,  $L3$ , представлен как  $A1-Z2$ , где выражение  $Z1 = A2$  должно быть истинным

Ниже приведен пример использования отношения `concat` для конкатенации списка  $[a,b,c]$ , представленного парой  $[a,b,c \mid T1] - T1$ , и списка  $[d,e]$ , представленного как  $[d,e \mid T2] - T2$ .

`?- concat( [a,b,c | T1] - T1, [d,e | T2] - T2, L ).`

Конкатенация выполняется путем согласования этой цели с предложением, определяющим процедуру `concat`, что приводит к получению следующего результата:

$T1 = [d,e \mid T2]$

$L = [a,b,c,d,e \mid T2] - T2$

Благодаря его эффективности описанный метод конкатенации списков с помощью разностных списков находит очень широкое применение, хотя и не может использоваться в столь разнообразных вариантах, как обычная процедура `conc`.

## 8.5.4. Оптимизация последнего вызова и накапливающие параметры

Рекурсивные вызовы обычно занимают значительное пространство памяти, которое освобождается только после возврата из последнего вызова. Большое количество вложенных рекурсивных вызовов может привести к нехватке памяти. Но в некоторых случаях существует возможность выполнять вложенные рекурсивные вызовы без потребности в дополнительной памяти. В подобных случаях рекурсивная процедура имеет специальную форму, которая обеспечивает *хвостовую рекурсию*. В процедуре с хвостовой рекурсией имеется только один рекурсивный вызов, который оформляется как последняя цель последнего предложения в процедуре. Кроме того, цели, предшествующие рекурсивному вызову, должны быть детерминированными, для того чтобы после этого последнего вызова не происходил перебор с возвратами. Такой детерминированности можно добиться, например, поместив оператор отсече-

ния непосредственно перед рекурсивным вызовом. Обычно процедура с хвостовой рекурсией выглядит примерно так:

```
p(...) :- % В теле этого предложения отсутствуют рекурсивные вызовы
p(...) :- % В теле этого предложения отсутствуют рекурсивные вызовы
...
p(...) :-
 ..., !, % Оператор отсекающего гарантирует исключение перебора с возвратами
 p(...) % Вызов с хвостовой рекурсией
```

В тех случаях, когда используются подобные процедуры с хвостовой рекурсией, не требуется какая-либо информация после возврата из вызова. Поэтому такая рекурсия может выполняться как итерация, в которой для каждого очередного прохода по циклу не требуется дополнительная память. Система Prolog, как правило, обнаруживает такую возможность экономии памяти и реализует хвостовую рекурсию как итерацию. Подобная ситуация называется *оптимизацией хвостовой рекурсии* или *оптимизацией последнего вызова*.

Если требования по экономии памяти приобретают исключительно важное значение, одно из возможных решений может состоять в использовании формулировок процедур с хвостовой рекурсией. Часто действительно существует возможность легко преобразовать рекурсивную процедуру в процедуру с хвостовой рекурсией. В качестве примера рассмотрим следующий предикат вычисления суммы списка чисел:

```
sumlist(List, Sum)
```

Ниже приведен первый, бесхитростный вариант его определения.

```
sumlist([], 0).
sumlist([First | Rest], Sum) :-
 sumlist(Rest, Sum0),
 Sum is X + Sum0.
```

В этой процедуре не применяется хвостовая рекурсия, поэтому для суммирования очень большого списка требуется много рекурсивных вызовов, следовательно, много памяти. Но известно, что в любом типичном процедурном языке такое суммирование может осуществляться с помощью простого итерационного цикла. Как же в этом случае преобразовать `sumlist` в процедуру с хвостовой рекурсией и дать возможность системе Prolog осуществлять суммирование с помощью итерации? К сожалению, нельзя просто переставить местами цели в теле второго предложения, поскольку цель `is` может выполняться лишь после вычисления значения `Sum0`. Но ниже показан широко распространенный прием, который позволяет выполнить такую замену.

```
sumlist(List, Sum) :-
 sumlist(List, 0, Sum). % Вызов вспомогательного предиката
% sumlist(List, PartialSum, TotalSum):
% TotalSum = PartialSum + сумма чисел в списке List
sumlist([], Sum, Sum). % Общая сумма равна частной сумме
sumlist([First | Rest], PartialSum, TotalSum) :-
 NewPartialSum is PartialSum + First,
 sumlist(Rest, NewPartialSum, TotalSum).
```

Теперь эта процедура становится процедурой с хвостовой рекурсией, и система Prolog может осуществить оптимизацию последнего вызова.

Показанный выше на примере процедуры `sumlist` прием, обеспечивающий преобразование обычной рекурсивной процедуры в процедуру с хвостовой рекурсией, используется очень широко. Чтобы определить целевой предикат `sumlist/2`, мы ввели вспомогательный предикат `sumlist/3`. Дополнительный параметр, `PartialSum`, обеспечил возможность применить формулировку с хвостовой рекурсией. Такие дополнительные параметры используются часто и известны под названием *накапливающих параметров* (accumulator argument). Окончательный результат постепенно накапливается в таком накапливающем параметре во время последовательных рекурсивных вызовов.

Ниже приведен еще один известный пример преобразования процедуры в форму с хвостовой рекурсией с помощью введения накапливающего параметра.

```
reverse(List, ReversedList)
```

В списке `ReversedList` представлены такие же элементы, как и в списке `List`, но в обратном порядке. Следующая процедура является примером первой, прямолинейной попытки:

```
reverse([], []).
reverse([X | Rest], Reversed) :-
 reverse(Rest, RevRest),
 conc(RevRest, [X], Reversed). % Добавить элемент X к концу
```

Это — не процедура с хвостовой рекурсией. Кроме того, она является также очень неэффективной из-за наличия в ней цели `conc( RevRest, [X], Reversed)`, для которой требуется время, пропорциональное длине списка `RevRest`. Поэтому для изменения порядка следования элементов на противоположный в списке с длиной  $n$  приведенная выше процедура потребует время, пропорциональное  $n^2$ . Но, безусловно, обращение списка может быть выполнено за линейное время. Поэтому из-за ее неэффективности приведенную выше процедуру (которая уже стала классическим примером) часто называют также “наивной попыткой выполнить обращение списка”. В следующей, намного более эффективной версии процедуры введен накапливающий параметр:

```
reverse(List, Reversed) :-
 reverse(List, [], Reversed).
% reverse(List, PartReversed, Reversed):
% Для получения списка Reversed добавление элементов списка List
% к списку PartReversed осуществляется в обратном порядке
reverse([], Reversed, Reversed).
reverse([X | Rest], PartReversed, TotalReversed) :-
 reverse(Rest, [X | PartReversed], TotalReversed). % Добавить элемент X
 % к накапливающему параметру
```

Эта процедура является эффективной (она требует затрат времени, пропорциональных длине списка), и в ней применяется хвостовая рекурсия.

## 8.5.5. Моделирование массивов с помощью предиката `arg`

Структура списка является самым удобным средством представления множеств в языке Prolog, но доступ к любому элементу в списке осуществляется путем просмотра списка. Такая операция требует времени, пропорционального длине списка, поэтому, если списки имеют большую длину, она становится очень неэффективной. Древовидные структуры, представленные в главах 9 и 10, обеспечивают намного более эффективный доступ. Но часто имеется возможность обращаться к элементам структуры с помощью индексов элементов. В подобных случаях наиболее эффективными являются структуры массивов, предусмотренные в других языках программирования, поскольку они обеспечивают непосредственный доступ к нужному элементу.

В языке Prolog отсутствуют средства поддержки массивов, но массивы в определенной степени можно моделировать с помощью встроенных предикатов `arg` и `functor`. Примеры использования этих предикатов приведены ниже. Цель

```
functor(A, f, 100)
```

создает структуру со 100 элементами:

```
A = f(_, _, _, ...)
```

В других языках типичный пример оператора, в котором осуществляется непосредственный доступ к элементу массива, выглядит так:

```
A[60] = 1
```

В этом операторе значение 60-го элемента массива A инициализируется числом 1. Аналогичный эффект в языке Prolog может быть достигнут с помощью следующей цели:

```
arg(60, A, 1)
```

При этом происходит непосредственный доступ к 60-му компоненту составного термина A, который в результате конкретизируется следующим образом:

```
A = f(_, ..., _, 1, ..., _) % 60-й элемент равен 1
```

Особенность этой конструкции состоит в том, что время, необходимое для доступа к N-му компоненту структуры, не зависит от N. Еще один типичный пример оператора из другого языка программирования состоит в следующем:

```
X = A[60]
```

Этот оператор можно представить с помощью моделируемой конструкции массива на языке Prolog следующим образом:

```
arg(60, A, X)
```

Такая операция является гораздо более эффективной, чем развертывание списка из 100 элементов и обращение к 60-му элементу с помощью вложенной рекурсии по элементам списка. Но операция обновления значения элемента в моделируемом массиве является громоздкой. В других языках после инициализации значений в массиве их можно обновлять, например, следующим образом:

```
A[60] = A[60] + 1
```

Прямолинейный подход к моделированию такого обновления отдельного значения в массиве на языке Prolog может состоять в следующем: сформировать полностью новую структуру со 100 компонентами с помощью предиката `functor`, вставить новое значение в соответствующее место в этой структуре и заполнить все остальные компоненты значениями соответствующих компонентов из предыдущей структуры. Вся эта операция является громоздкой и очень неэффективной. Одна из идей по усовершенствованию данной операции состоит в том, что должны быть предусмотрены неконкретизированные *вакансии* (незаполненные места) в компонентах структуры, чтобы можно было в будущем размещать в этих вакансиях новые значения элементов массива. Поэтому можно, например, хранить значения последовательных обновлений в списке, хвост которого представляет собой неконкретизированную переменную — вакансию для будущих значений. В качестве примера рассмотрим следующие операции обновления значения переменной X в программе на процедурном языке:

```
X := 1; X := 2; X := 3
```

Эти обновления можно моделировать на языке Prolog с помощью метода вакансий следующим образом:

```
X = [1 | Rest1] % Соответствует X = 1, Rest1 - вакансия для будущих значений
Rest1 = [2 | Rest2] % Соответствует X = 2, Rest2 - вакансия для будущих значений
Rest2 = [3 | Rest3] % Соответствует X = 3
```

К этому моменту  $X = [1, 2, 3 | \text{Rest3}]$ . Очевидно, что хранятся все предыдущие значения X, а текущим является значение, непосредственно предшествующее вакансии. Но если количество последовательных обновлений велико, текущее значение становится глубоко вложенным и этот метод снова теряет эффективность. Еще одна идея, позволяющая исключить указанную причину снижения эффективности, состоит в том, что нужно отбрасывать предыдущие значения в тот момент, когда список становится слишком длинным, и снова начинать со списка, состоящего только из головы и неконкретизированного хвоста.

Несмотря на эти возможные осложнения, метод моделирования массивов с помощью предиката `arg` во многих случаях является достаточно простым и действует вполне успешно. В качестве одного из таких примеров можно привести решение 3 для задачи с восемью ферзями из главы 4 (см. листинг 4.4). Эта программа помещает очередного ферзя на вертикальный ряд (координата X), горизонтальный ряд (координата Y), восходящую диагональ (координата U) и нисходящую диагональ (координата

та V), которые в данный момент свободны. В программе сопровождаются множества координат, свободных в настоящее время, и после размещения нового ферзя на клетке с соответствующими координатами эти занятые координаты удаляются из указанных множеств. Для удаления координат U и V в листинге 4.4 применяется просмотр соответствующих списков, но эта операция является неэффективной. Эффективность можно легко повысить путем моделирования массивов. Поэтому множество, состоящее из всех 15 восходящих диагоналей, можно представить с помощью следующего термина с 15 компонентами:

```
Du = u(('_', '_', '_', '_', '_', '_', '_', '_', '_', '_', '_', '_'))
```

Предположим, что ферзь помещен на клетку  $(X, Y) = (1, 1)$ . Эта клетка находится на 8-й восходящей диагонали. Тот факт, что данная диагональ теперь занята, может быть отмечен путем конкретизации 8-го компонента структуры Du значением 1 (т.е. значением соответствующей координаты X) следующим образом:

```
arg(8, Du, 1)
```

Теперь Du приобретает вид

```
Du = u(('_', '_', '_', '_', '_', '1', '_', '_', '_', '_', '_', '_'))
```

Если в дальнейшем будет предпринята попытка поместить ферзя на клетку  $(X, Y) = (3, 3)$ , также лежащую на 8-й диагонали, то для этого потребуется выполнить следующую операцию:

```
arg(8, Du, 3) % Здесь X = 3
```

Такая попытка окончится неудачей, поскольку 8-й компонент структуры Du уже равен 1. Поэтому программа не позволит поместить на одну и ту же диагональ еще одного ферзя. Эта реализация множеств восходящих и нисходящих диагоналей приводит к созданию намного более эффективной программы по сравнению с приведенной в листинге 4.4.

## 8.5.6. Повышение эффективности путем внесения в базу данных производных фактов

Иногда в ходе вычислений приходится снова и снова достигать одной и той же цели. Поскольку в системе Prolog отсутствует специальный механизм обнаружения подобных ситуаций, то повторяются целые последовательности вычислений.

В качестве примера рассмотрим программу вычисления N-го числа Фибоначчи для заданного N. Ряд Фибоначчи имеет следующий вид:

```
1, 1, 2, 3, 5, 8, 13, ...
```

Каждое число в этом ряде, за исключением первых двух, представляет собой сумму предыдущих двух чисел. Определим предикат

```
fib(N, F)
```

для вычисления N-го числа Фибоначчи, F, соответствующего заданному значению N. Эти числа можно получить последовательно, начиная с  $N = 1$ . В приведенной ниже программе fib вначале рассматриваются первые два числа Фибоначчи как два частных случая, а затем задается общее правило, касающееся ряда Фибоначчи.

```
fib(1, 1). % 1-е число Фибоначчи
fib(2, 1). % 2-е число Фибоначчи
fib(N, F) :- % N-е число Фибоначчи, N > 2
 N > 2,
 N1 is N - 1, fib(N1, F1),
 N2 is N - 2, fib(N2, F2),
 F is F1 + F2. % N-е число - сумма двух предшествующих ему чисел
```

В данной программе часть вычислений почти всегда выполняется повторно. В этом можно легко убедиться, проведя трассировку выполнения следующей цели:

```
?- fib(6, F).
```

На рис. 8.2 иллюстрируются характерные особенности вычислительного процесса. Например, третье число Фибоначчи,  $f(3)$ , требуется в трех местах, и каждый раз повторяется одно и то же вычисление.

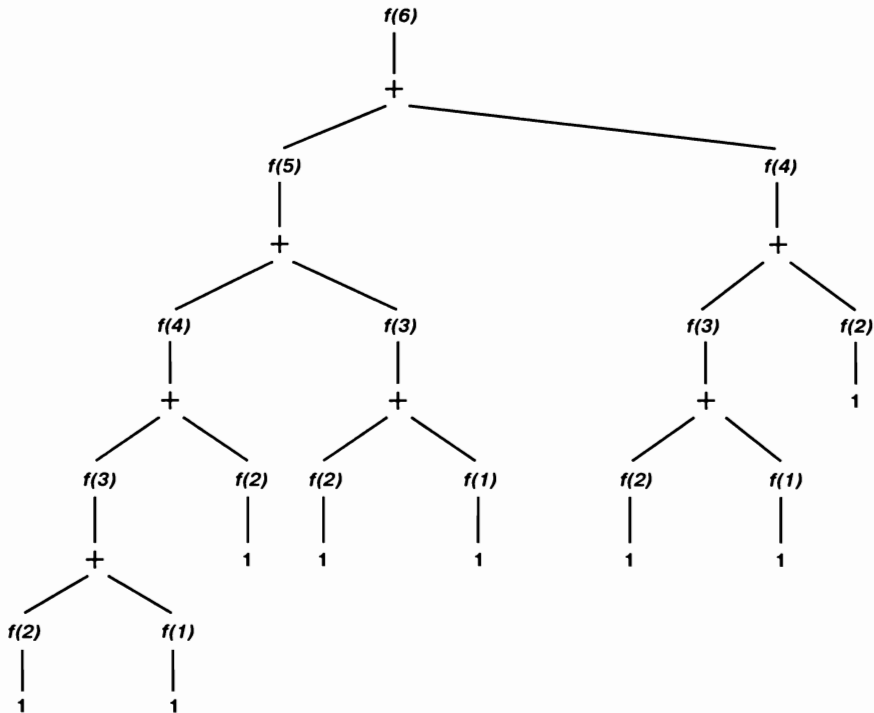


Рис. 8.2. Вычисление 6-го числа Фибоначчи с помощью процедуры *fib*

Этого можно легко избежать, запоминая каждое вновь полученное число Фибоначчи. Идея состоит в том, что следует использовать встроенную процедуру *asserta* и вводить в базу данных такие (промежуточные) результаты как факты. Эти факты должны предшествовать другим предложениям, касающимся *fib*, чтобы можно было исключить необходимость в использовании общего правила в тех случаях, если результат уже известен. Модифицированная процедура *fib2* отличается от процедуры *fib* только тем, что в ней предусмотрено внесение информации в базу данных следующим образом:

```

fib2(1, 1).
fib2(2, 1).
fib2(N, F) :-
 N > 2,
 N1 is N - 1,
 fib2(N1, F1),
 N2 is N - 2,
 fib2(N2, F2),
 F is F1 + F2,
 asserta(fib2(N, F)).

```

% 1-е число Фибоначчи  
 % 2-е число Фибоначчи  
 % N-е число Фибоначчи, N > 2  
 % Запомнить N-е число

Эта программа предпринимает попытки найти ответ на любую цель *fib2*, вначале рассматривая сохраненные в базе данных факты, касающиеся этого отношения, и только после этого обращается к общему правилу. В результате после выполнения цели *fib2( N, F)* все числа Фибоначчи вплоть до N-го числа становятся зафиксиро-

ванными. На рис. 8.3 показан процесс вычисления 6-го числа Фибоначчи процедурой `fib2`. Сравнение с рис. 8.2 показывает, что вычислительная сложность процедуры уменьшилась. Чем больше значение  $N$ , тем более существенным становится это уменьшение.

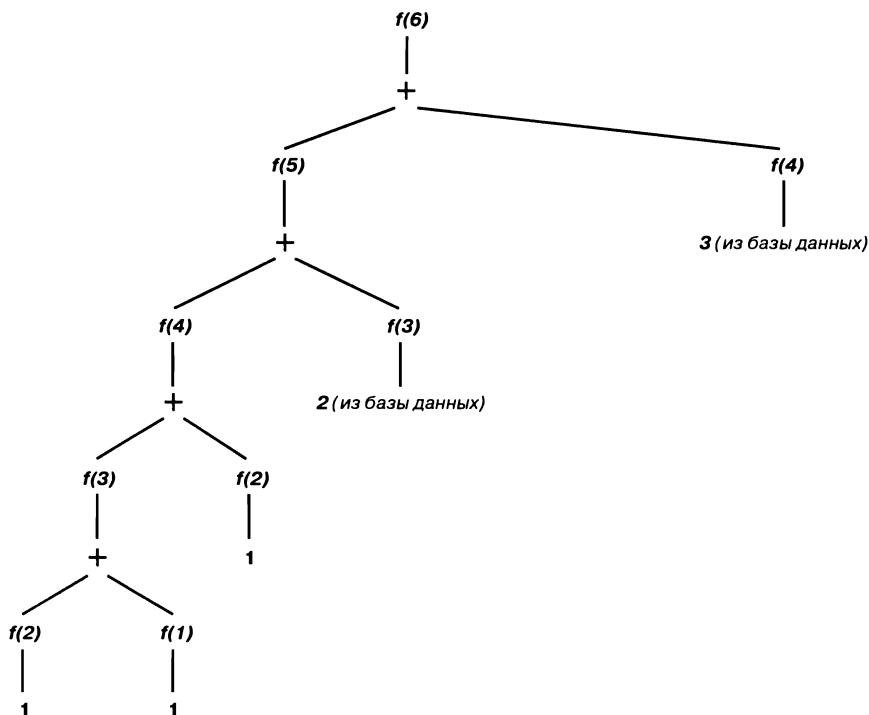


Рис. 8.3. Вычисление 6-го числа Фибоначчи с помощью процедуры `fib2`, которая запоминает предыдущие результаты; благодаря этому сокращается объем вычислений по сравнению с процедурой `fib` (см. рис. 8.2)

Внесение в базу данных промежуточных результатов, называемое также *кэшированием*, — это стандартный метод предотвращения повторных вычислений. Но следует отметить, что в случае чисел Фибоначчи можно применить более предпочтительный метод предотвращения повторных вычислений с использованием другого алгоритма, а не вносить в базу данных промежуточные результаты. Этот другой алгоритм приводит к созданию программы, которая является более трудной для понимания, но более эффективной при выполнении. В предыдущей версии  $N$ -е число Фибоначчи определялось как сумма своих двух предшественников, а для развертывания всего процесса вычислений “в направлении вниз”, к двум первоначальным числам Фибоначчи применялись рекурсивные вызовы. Вместо этого можно организовать работу “в направлении вверх”, начиная с двух исходных чисел и вычисляя значения ряда одно за другим, по возрастанию. При этом достаточно только во время остановиться после получения  $N$ -го значения. Основная часть работы в такой программе выполняется с помощью следующей процедуры:

```
forwardfib(M, N, F1, F2, F)
```

где  $F1$  и  $F2$  —  $(M-1)$ -е и  $M$ -е числа Фибоначчи, а  $F$  —  $N$ -е число Фибоначчи. Принцип действия отношения `forwardfib` показан на рис. 8.4. В соответствии с этим рисунком процедура `forwardfib` находит последовательность преобразований, позволяю-



щую достичь окончательной конфигурации (при которой  $M = N$ ) из заданной начальной конфигурации. При вызове `forwardfib` все параметры, кроме  $F$ , должны быть конкретизированными, а  $M$  должно быть меньше или равно  $N$ . Соответствующая программа показана ниже.

```
fib3(N, F) :-
 forwardfib(2, N, 1, 1, F). % Первые два числа Фибоначчи равны 1
forwardfib(M, N, F1, F2, F2) :-
 M >= N. % N-е число Фибоначчи достигнуто
forwardfib(M, N, F1, F2, F) :-
 M < N, % N-е число Фибоначчи еще не достигнуто
 NextM is M + 1,
 NextF2 is F1 + F2,
 forwardfib(NextM, N, F2, NextF2, F).
```

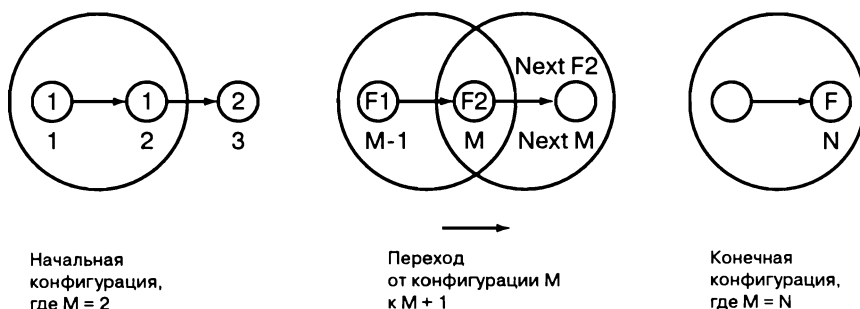


Рис. 8.4. Соотношения между числами ряда Фибоначчи; конфигурация, обозначенная большим кружком, определяется тремя компонентами: индексом  $M$  и двумя последовательными числами Фибоначчи,  $f(M-1)$  и  $f(M)$

Следует отметить, что в процедуре `forwardfib` применяется хвостовая рекурсия, а  $M$ ,  $F1$  и  $F2$  представляют собой накапливающие параметры.

## Упражнения

8.1. Во всех процедурах, показанных ниже (`sub1`, `sub2` и `sub3`), реализовано отношение, обеспечивающее выделение подсписка. Процедура `sub1` имеет “более процедурное” определение, а процедуры `sub2` и `sub3` написаны в “более декларативном” стиле. Изучите поведение этих трех процедур на примере нескольких списков, обращая особое внимание на их эффективность. Две из них по своему действию являются примерно одинаковыми и имеют аналогичную эффективность. Каковы эти две процедуры? Почему третья является менее эффективной?

```
sub1(List, Sublist) :-
 prefix(List, Sublist).
sub1([_ | Tail], Sublist) :-
 sub1(Tail, Sublist). % Sublist является подсписком списка Tail

prefix(_, []).
prefix([X | List1], [X | List2]) :-
 prefix(List1, List2).

sub2(List, Sublist) :-
 conc(List1, List2, List),
 conc(List3, Sublist, List1).

sub3(List, Sublist) :-
 conc(List1, List2, List),
 conc(Sublist, _, List2).
```

### 8.2. Определите отношение

```
add_at_end(List, Item, NewList)
```

для добавления элемента `Item` к концу списка `List` и получения списка `NewList`. Предусмотрите возможность представления обоих списков с помощью разностных пар.

### 8.3. Определите отношение

```
reverse(List, ReversedList)
```

в котором оба списка представлены с помощью разностных пар.

### 8.4. Переоформите процедуру `collect`, приведенную в разделе 8.5.2, с использованием представления для списков с помощью разностных пар, чтобы операция конкатенации могла быть выполнена более эффективно.

### 8.5. Приведенная ниже процедура вычисляет максимальное значение в списке чисел.

```
max([X], X).
max([X | Rest], Max) :-
 max(Rest, MaxRest),
 (MaxRest >= X, !, Max = MaxRest
 ;
 Max = X).
```

Преобразуйте ее в процедуру с хвостовой рекурсией. Подсказка: введите накапливающий параметр `MaxSoFar`.

### 8.6. Переоформите программу 3 для задачи с восемью ферзями (см. листинг 4.4) с использованием массивов, моделируемых с помощью предиката `arg`, для представления множеств свободных диагоналей, как описано в разделе 8.5.5. Проведите измерения быстродействия, чтобы узнать, насколько повысилась эффективность.

### 8.7. Реализуйте операцию обновления значения элемента в массиве, моделируемом с помощью предикатов `functor` и `arg`, используя вакансии для будущих значений, в соответствии с указаниями, приведенными в разделе 8.5.5.

## Резюме

- Для оценки качества программ применяются следующие критерии:
  - правильность;
  - дружелюбность;
  - эффективность;
  - удобство для чтения;
  - модифицируемость;
  - надежность;
  - документированность.
- Принцип *поэтапного совершенствования* может стать основой удобного подхода к организации процесса разработки программы. Поэтапное совершенствование распространяется на отношения, алгоритмы и структуры данных.
- В языке Prolog перечисленные ниже методы часто позволяют находить идеи для усовершенствований.
  - Использование рекурсии. Выявление граничных и общих случаев рекурсивного определения.
  - Обобщение. Формулировка общей задачи, которая может оказаться более простой для решения по сравнению с первоначальной.

- **Использование графических схем.** Графическое изображение условий задачи может помочь выявить важные отношения.
- Необходимо придерживаться определенных *стилистических соглашений* для уменьшения вероятности программистских ошибок, повышения удобства программ для чтения, отладки и модификации.
- В системах Prolog обычно предусмотрены *средства отладки* программ. К числу наиболее полезных из них относятся *средства трассировки*.
- Существует много методов *повышения эффективности* программы. Наиболее простыми из этих методов являются следующие:
  - *переупорядочение* целей и предложений;
  - управление перебором с возвратами путем вставки *операторов отсечения*;
  - *запоминание* (с помощью предиката `asserta`) решений, которые в ином случае пришлось бы вычислять повторно.
- В результате применения более развитых методов создаются лучшие алгоритмы (к числу особенно ярких примеров относятся алгоритмы, способствующие повышению эффективности поиска) и лучшие структуры данных. Часто используемыми методами программирования такого типа являются следующие:
  - разностные списки;
  - хвостовая рекурсия, оптимизация последнего вызова;
  - накапливающие параметры;
  - моделирование массивов с помощью предикатов `functor` и `arg`.

## Дополнительные источники информации

В [117] и [132] приведены результаты глубоких исследований проблемы эффективности проектирования программ и стиля программирования на языке Prolog. Стерлинг (Sterling) отредактировал сборник статей [153] с описанием проектов крупных программ Prolog для целого ряда практических приложений.

# Операции со структурами данных

*В этой главе...*

|                                                         |     |
|---------------------------------------------------------|-----|
| 9.1. Сортировка списков                                 | 192 |
| 9.2. Представление множеств с помощью бинарных деревьев | 197 |
| 9.3. Вставка и удаление в бинарном словаре              | 201 |
| 9.4. Отображение деревьев                               | 206 |
| 9.5. Графы                                              | 208 |

Одной из фундаментальных проблем программирования является поиск способов представления сложных объектов данных, таких как множества, и эффективной реализации операций с этими объектами. В данной главе рассматриваются часто используемые структуры данных, которые относятся к трем крупным семействам: списки, деревья и графы. В ней описаны способы представления этих структур на языке Prolog и приведены программы для выполнения некоторых операций с этими структурами, таких как сортировка списков, представление наборов данных с помощью древовидных структур, хранение данных в виде деревьев и выборка данных из древовидных структур, поиск путей в графах и т.д. В этой главе рассматривается целый ряд примеров, поскольку указанные операции являются исключительно характерными для программирования на языке Prolog.

## 9.1. Сортировка списков

Список может быть отсортирован, если определено отношение упорядочения между элементами в списке. Для целей этого описания предполагается, что существует следующее отношение упорядочения:

```
gt(X, Y)
```

которое означает, что  $X$  больше  $Y$ , независимо от того, какой смысл вкладывается в понятие “больше”. Если рассматриваемыми элементами являются числа, то отношение  $gt$  может быть, по всей вероятности, определено следующим образом:

```
gt(X, Y) :- X>Y.
```

Если же элементы представляют собой атомы, то отношение  $gt$  может соответствовать лексикографическому упорядочению, например, определенному как:

```
gt(X, Y) :- X@>Y.
```

Напомним, что это отношение упорядочивает также составные термы. Предположим, что терм

```
sort(List, Sorted)
```

обозначает отношение, где  $List$  является списком элементов, а  $Sorted$  — списком тех же элементов, отсортированным в порядке возрастания в соответствии с отноше-

нием `gt`. В данной главе рассматриваются три определения этого отношения на языке Prolog, которые основаны на разных принципах сортировки списков. Ниже описан первый принцип.

Для сортировки списка `List` необходимо выполнить следующее:

- найти в списке `List` два смежных элемента, `X` и `Y`, таких, что отношение `gt( X, Y)` принимает истинное значение, и поменять местами `X` и `Y` в списке `List`, получив при этом список `List1`; после этого отсортировать `List1`;
- если в списке `List` нет ни одной пары смежных элементов, `X` и `Y`, таких, что отношение `gt( X, Y)` принимает истинное значение, то список `List` уже отсортирован.

Результатом перестановки местами двух элементов, `X` и `Y`, которые занимают неправильное положение по отношению друг к другу, является то, что после этой перестановки новый список в большей степени напоминает отсортированный список. После достаточного количества перестановок в конечном итоге должно быть достигнуто такое состояние, в котором все элементы занимают правильные положения. Такой принцип сортировки известен под названием *пузырьковой сортировки* (bubble sort). Поэтому соответствующей процедуре Prolog, приведенной ниже, будет присвоено название `bubblesort`.

```
bubblesort(List, Sorted) :-
 swap(List, List1), !, % Есть ли допустимая перестановка в списке List?
 bubblesort(List1, Sorted).

bubblesort(Sorted, Sorted). % В противном случае список уже отсортирован

swap([X, Y | Rest], [Y, X | Rest]) :- % Поменять местами первые два элемента
 gt(X, Y).

swap([Z | Rest], [Z | Rest1]) :- % Поменять местами элементы в хвосте
 swap(Rest, Rest1).
```

Еще одним простым алгоритмом сортировки является сортировка вставкой, которая основана на описанном ниже принципе.

Чтобы отсортировать непустой список, `L = [X | T]`, необходимо выполнить следующее.

1. Отсортировать хвост `T` списка `L`.
2. Вставить голову `X` списка `L` в отсортированный хвост списка, в такую позицию, чтобы результирующий список оставался отсортированным. В результате будет получен весь отсортированный список.

Такую постановку задачи можно перевести на язык Prolog в виде следующей процедуры `insertsort`:

```
insertsort([], []).
insertsort([X | Tail], Sorted) :-
 insertsort(Tail, SortedTail), % Отсортировать хвост списка

insert(X, SortedTail, Sorted). % Вставить X в надлежащем месте

insert(X, [Y | Sorted], [Y | Sorted1]) :-
 gt(X, Y), !,
 insert(X, Sorted, Sorted1).

insert(X, Sorted, [X | Sorted]).
```

Процедуры сортировки `bubblesort` и `insertsort` являются простыми, но неэффективными. Из этих двух процедур сортировка вставкой (`insertsort`) является более эффективной. Но среднее время, необходимое для сортировки списка длиной  $n$  с помощью процедуры `insertsort`, возрастает пропорционально  $n^2$ . Поэтому для сортировки длинных списков предусмотрен гораздо более эффективный алгоритм сортировки — `quicksort`. Он основан на следующем принципе, который иллюстрируется на рис. 9.1.

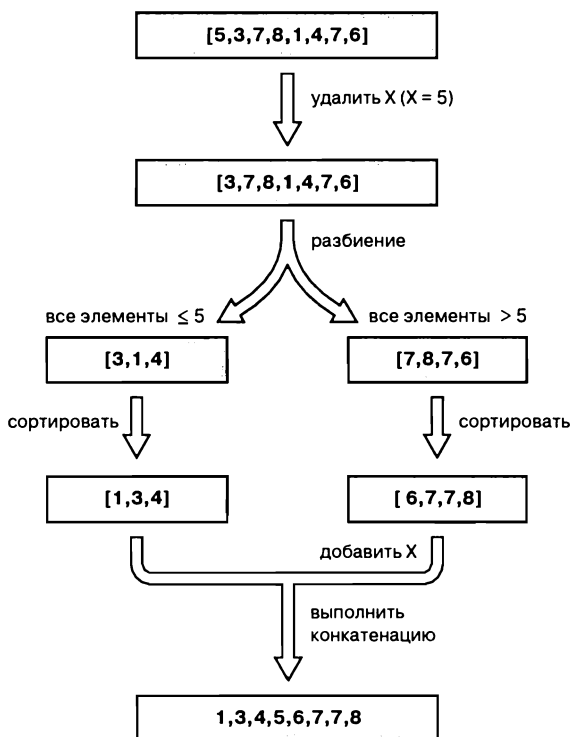


Рис. 9.1. Сортировка списка с помощью алгоритма `quicksort`

Описание алгоритма `quicksort` приведено ниже.

Чтобы отсортировать непустой список `L`, необходимо выполнить следующие действия.

1. Удалить некоторый элемент `X` из списка `L` и разбить остаток `L` на два списка, называемые `Small` и `Big`, следующим образом: перенести все элементы списка `L`, которые больше `X`, в список `Big`, а все остальные — в список `Small`.
2. Отсортировать список `Small`, получив список `SortedSmall`.
3. Отсортировать список `Big`, получив список `SortedBig`.
4. Весь отсортированный список представляет собой конкатенацию списков `SortedSmall` и `[X | SortedBig]`.

Если сортируемый список пуст, то результатом сортировки также является пустой список. Реализация процедуры `quicksort` на языке Prolog приведена в листин-

ге 9.1. Характерной особенностью этой реализации является то, что элемент X, удаляемый из списка L, всегда представляет собой голову списка L. Операция разбиения списка программируется в виде следующего отношения с четырьмя параметрами:

```
split(X, L, Small, Big)
```

Сложность этого алгоритма, определяемая затратами времени на выполнение, зависит от того, насколько удачно происходит разбиение сортируемого списка. Если список разбивается на два списка, имеющих приблизительно равную длину, то затраты времени на выполнение этой процедуры сортировки пропорциональны  $n \log n$ , где  $n$  — длина сортируемого списка. Если, вопреки ожиданиям, разбиение всегда приводит к тому, что один список становится больше другого, то затраты времени становятся пропорциональными  $n^2$ . Но, к счастью, исследования показали, что средняя производительность процедуры quicksort ближе к наилучшему варианту, чем к наихудшему.

Программу, приведенную в листинге 9.1, можно дополнительно усовершенствовать, применив лучшую реализацию операции конкатенации. Благодаря использованию представления списков в виде разностных пар, описанного в главе 8, операция конкатенации становится тривиальной. Для применения этой идеи в рассматриваемой процедуре сортировки списки, применяемые в листинге 9.1, можно представить в виде пары списков в форме A-Z следующим образом:

SortedSmall представлен как A1-Z1

SortedBig представлен как A2-Z2

#### Листинг 9.1. Процедура быстрой сортировки quicksort

```
% quicksort(List, SortedList):
% сортировка списка List с применением алгоритма quicksort

quicksort([], []).

quicksort([X|Tail], Sorted) :-
 split(X, Tail, Small, Big),
 quicksort(Small, SortedSmall),
 quicksort(Big, SortedBig),
 conc(SortedSmall, [X|SortedBig], Sorted).

split(X, [], [], []).

split(X, [Y|Tail], [Y|Small], Big) :-
 gt(X, Y), !,
 split(X, Tail, Small, Big).

split(X, [Y|Tail], Small, [Y|Big]) :-
 split(X, Tail, Small, Big).
```

Таким образом, конкатенация списков SortedSmall и [X | SortedBig] соответствует конкатенации следующих пар:

A1 - Z1 и [X | A2] - Z2

Список, полученный в результате конкатенации, представляется с помощью следующего выражения:

A1 - Z2, где Z2 = [X | A2]

Пустой список представлен любой парой Z-Z. В результате систематического внесения этих изменений в программу (см. листинг 9.1) получена более эффективная реализация процедуры quicksort, запрограммированная в виде quicksort2 в листинге 9.2. В процедуре quicksort все еще используется обычное представление списков, но сортировка фактически выполняется более эффективной процедурой

quicksort2, в которой применяется представление списков в виде разностных пар. Отношение между этими двумя процедурами выглядит следующим образом:

```
quicksort(L, S) :-
 quicksort2(L, S - []).
```

**Листинг 9.2. Более эффективная реализация процедуры quicksort с использованием представления списков в виде разностных пар; отношение split( x, List, Small, Big) приведено в листинге 9.1**

```
% quicksort(List, SortedList):
% сортировка списка List с применением алгоритма quicksort

quicksort(List, Sorted) :-
 quicksort2(List, Sorted - []).

% quicksort2(List, SortedDiffList): сортировка списка List; результат
% представлен как разностный список

quicksort2([], Z - Z).

quicksort2([X | Tail], A1 - Z2) :-
 split(X, Tail, Small, Big),
 quicksort2(Small, A1 - [X | A2]),
 quicksort2(Big, A2 - Z2).
```

## Упражнения

**9.1.** Напишите процедуру для слияния двух отсортированных списков и получения третьего списка, например, следующим образом:

```
?- merge([2,5,6,6,8], [1,3,5,9], L).
L = [1,2,3,5,5,6,6,8,9]
```

**9.2.** Различие между программами сортировки, приведенными в листингах 9.1 и 9.2, состоит в том, что используются иные представления списков, — в последней применяются разностные списки. Процедура преобразования обычных списков в разностные является простой и может выполняться механически. В качестве упражнения систематически внесите соответствующие изменения в программу, приведенную в листинге 9.1, чтобы преобразовать ее в программу, которая показана в листинге 9.2.

**9.3.** Рассматриваемая программа quicksort обнаруживает низкую эффективность, если сортируемый список уже полностью отсортирован или почти отсортирован. Объясните, с чем это связано.

**9.4.** Еще один перспективный принцип сортировки списков, позволяющий устранить указанный выше недостаток процедуры quicksort, основан на том, что список делится на части, после чего меньшие списки сортируются, а затем эти отсортированные меньшие списки сливаются. В соответствии с этим, чтобы отсортировать список L, необходимо выполнить следующее:

- разделить список L на два списка, L1 и L2, приблизительно равной длины;
- отсортировать L1 и L2, получив списки S1 и S2;
- выполнить слияние списков S1 и S2 и получить отсортированный список L.

Такой алгоритм известен под названием *алгоритма сортировки – слияния*. Реализуйте алгоритм сортировки-слияния и сравните эффективность полученной программы с программой quicksort.



## 9.2. Представление множеств с помощью бинарных деревьев

Одна из обычных областей применения списков состоит в представлении множеств объектов. Недостатком использования списка для представления множества является то, что при этом операция проверки принадлежности к множеству довольно неэффективна. Предикат `member( X, L)` для проверки того, входит ли элемент `X` в состав списка `L`, обычно программируется следующим образом:

```
member(X, [X | L]).
```

```
member(X, [Y | L]) :-
 member(X, L).
```

Эта процедура, чтобы найти элемент `X` в списке `L`, просматривает элементы списка один за другим до тех пор, пока не будет найден элемент `X` или обнаружен конец списка. В случае использования больших списков такая операция становится очень неэффективной.

Для представления множеств могут использоваться различные древовидные структуры, которые обеспечивают более эффективную реализацию отношения проверки принадлежности к множеству. В данном разделе рассматриваются такие структуры, как бинарные деревья.

Бинарное дерево либо является пустым, либо состоит из следующих трех компонентов:

- корень;
- левое поддереву;
- правое поддереву.

Корнем может служить любой объект, но поддеревья должны снова представлять собой бинарные деревья. Пример такой структуры показан на рис. 9.2. Это дерево соответствует множеству  $\{a, b, c, d\}$ . Элементы этого множества хранятся как узлы дерева. На рис. 9.2 пустые поддеревья не показаны; например, узел `b` имеет два поддерева, из которых оба пусты.

Предусмотрено много способов представления бинарных деревьев в виде термов Prolog. Один из простых вариантов состоит в том, чтобы корень бинарного дерева рассматривался как главный функтор термина, а поддеревья — как его параметры. В соответствии с этим пример дерева, приведенный на рис. 9.2, может быть представлен следующим образом:

```
a(b, c(d))
```

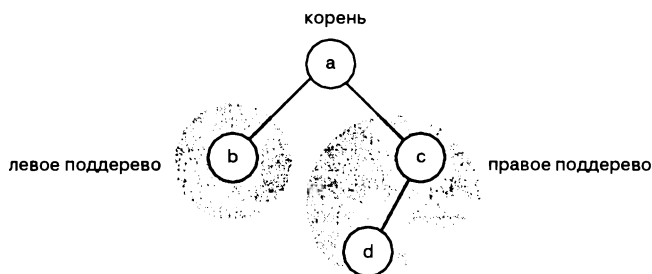


Рис. 9.2. Бинарное дерево

Но такое представление имеет множество недостатков; в частности, оно требует применения отдельного функтора для каждого узла дерева. Это может привести к затруднениям, если сами узлы представляют собой структурированные объекты.

Лучший и более широко применяемый способ представления бинарных деревьев состоит в следующем: используются специальный символ для представления пустого дерева и функтор для формирования непустого дерева из трех его компонентов (корня и двух поддеревьев). В данном разделе в качестве специального символа и функтора применяются следующие:

- атом `nil` используется для представления пустого дерева;
- применяется функтор `t`, такой, что дерево, которое имеет корень `X`, левое поддерево `L` и правое поддерево `R`, представляется термом `t( L, X, R )`, как показано на рис. 9.3.

При таком способе представления дерево, показанное в качестве примера на рис. 9.2, обозначается следующим термом:

```
t(t(nil, b, nil), a, t(t(nil, d, nil), c, nil))
```

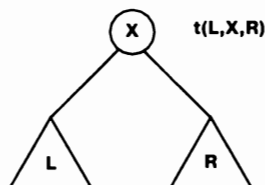


Рис. 9.3. Представление бинарных деревьев

Теперь рассмотрим отношение проверки принадлежности к множеству, называемое в данном разделе как `in`. Цель

```
in(X, T)
```

является истинной, если `X` — узел в дереве `T`. Отношение `in` может быть определено с помощью приведенных ниже правил.

`X` находится в дереве `T`, если:

- `X` является корнем `T`, или
- `X` находится в левом поддереве `T`, или
- `X` находится в правом поддереве `T`.

Эти правила можно непосредственно перевести на язык Prolog следующим образом:

```
in(X,t(_ ,X,_)).
in(X,t(L,_ ,_)) :-
 in(X, L).
in(X, t(_ ,_,R)) :-
 in(X, R).
```

Очевидно, что цель

```
in(X, nil)
```

не будет достигаться при любом значении `X`.

Рассмотрим поведение процедуры `in`. В приведенных ниже примерах `T` представляет собой дерево, приведенное на рис. 9.2. Цель

```
in(X, T)
```

с помощью перебора с возвратами находит все данные в множестве в таком порядке:

```
X = a; X = b; X = c; X = d
```

Теперь рассмотрим, насколько эффективной является процедура `in`. Цель

in( a, T)

немедленно достигается в результате выполнения первого предложения в процедуре in. С другой стороны, для достижения цели

in( d, T)

потребуется несколько рекурсивных вызовов процедуры in, прежде чем будет в конечном итоге найден элемент d. Аналогичным образом, цель

in( e, T)

не достигается только после завершения поиска с помощью рекурсивных вызовов процедуры in во всех поддеревьях дерева T.

Поэтому такая конструкция является столь же неэффективной, как и простой способ представления множества в виде списка. Но можно добиться ее значительного усовершенствования после введения отношения упорядочения между элементами данных во множестве. В таком случае появляется возможность упорядочивать данные в дереве слева направо согласно этому отношению. Непустое дерево  $t(Left, X, Right)$  называется *упорядоченным слева направо*, если соблюдаются следующие условия.

1. Все узлы в левом поддереве, Left, меньше X.
2. Все узлы в правом поддереве, Right, больше X.
3. Оба поддерева также являются упорядоченными.

В дальнейшем такое бинарное дерево будем называть *бинарным словарем*. Пример подобного дерева приведен на рис. 9.4.

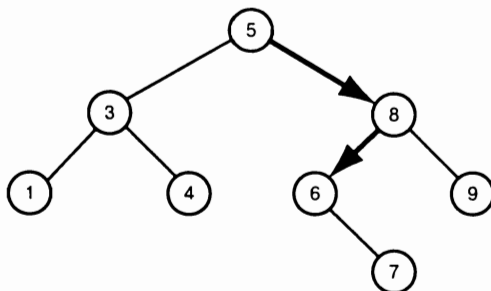


Рис. 9.4. Пример бинарного словаря; элемент 6 достигается в результате прохождения по указанному в дереве пути  $5 \rightarrow 8 \rightarrow 6$

Преимущество упорядочения состоит в том, что для поиска любого объекта в бинарном словаре всегда достаточно провести поиск не более чем в одном поддереве. Ключом к такой экономии усилий при поиске X является то, что в результате сравнения X и корня из рассмотрения немедленно исключается, по меньшей мере, одно из поддеревьев. Например, проведем поиск элемента 6 в дереве, показанном на рис. 9.4. Начнем с корня 5, сравним 6 и 5 и определим, что  $6 > 5$ . Поскольку все данные в левом поддереве должны быть меньше 5, единственная оставшаяся возможность состоит в проведении поиска элемента 6 в правом поддереве. Поэтому поиск продолжается в правом поддереве, происходит переход к узлу 8 и т.д.

Общий метод поиска в бинарном словаре описан ниже.

Чтобы найти элемент X в словаре D, необходимо выполнить следующие действия:

- если X — корень D, то элемент X найден, в противном случае,
- если X — меньше, чем корень D, то проводить поиск X в левом поддереве D, в противном случае

- проводить поиск  $X$  в правом поддереве  $D$ ;
- если дерево  $D$  пусто, поиск оканчивается неудачей.

Эти правила запрограммированы в виде процедуры `in`, приведенной в листинге 9.3. Отношение `gt( X, Y )` означает:  $X$  больше  $Y$ . Если элементы, хранящиеся в дереве, представляют собой числа, то это отношение принимает вид  $X > Y$ .

### Листинг 9.3. Процедура поиска элемента $X$ в бинарном словаре

```
% in(X, Tree): элемент X находится в бинарном словаре Tree

in(X, t(_, X, _)).

in(X, t(Left, Root, Right)) :-
 gt(Root, X), % Корень больше, чем X
 in(X, Left). % Проводить поиск в левом поддереве

in(X, t(Left, Root, Right)) :-
 gt(X, Root), % X больше, чем корень
 in(X, Right). % Проводить поиск в правом поддереве
```

В определенном смысле сама процедура `in` может также использоваться для формирования бинарного словаря. Например, следующий ряд целей вызовет формирование словаря  $D$ , который содержит элементы 5, 3, 8:

```
?- in(5, D), in(3, D), in(8, D).
D = t(t(D1, 3, D2), 5, t(D3, 8, D4)).
```

Переменные  $D1$ ,  $D2$ ,  $D3$  и  $D4$  являются четырьмя неопределенными поддеревьями. Они могут представлять собой что угодно, но дерево  $D$  все равно будет содержать заданные элементы 3, 5 и 8. Структура сформированного словаря зависит от порядка целей в вопросе (рис. 9.5).

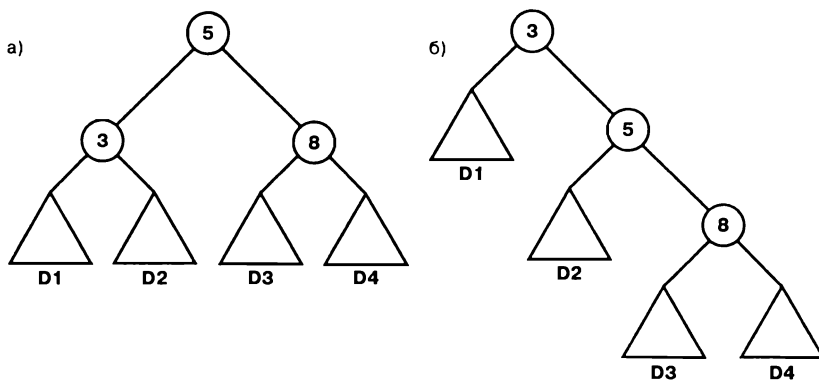


Рис. 9.5. Пример того, что структура бинарного словаря зависит от порядка целей в вопросе: а) дерево  $D$ , формируемое при выполнении последовательности целей `in( 5, D), in( 3, D), in( 8, D)`; б) дерево, формируемое при обработке вопроса `in( 3, D), in( 5, D), in( 8, D)`

Теперь уместно привести комментарии по поводу того, какова эффективность поиска в словарях. Вообще говоря, поиск элемента в словаре осуществляется более эффективно, чем поиск в списке. Как оценить степень повышения этой эффективности? Предположим, что  $n$  — количество элементов в наборе данных. Если множество представлено списком, то ожидаемое время поиска должно быть пропорционально

его длине  $n$ . В среднем приходится выполнять просмотр списка до некоторого места, находящегося примерно в середине этого списка. А если множество представлено в виде бинарного словаря, то время поиска примерно пропорционально высоте дерева. *Высотой дерева* называется длина самого длинного пути между корнем и листом дерева. Но высота зависит от формы дерева.

Дерево называется (примерно) сбалансированным, если для каждого узла в дереве два поддерева этого узла содержат приблизительно равное количество элементов. Если словарь с  $n$  узлами полностью сбалансирован, его высота пропорциональна  $\log n$ . Поэтому считается, что сбалансированное дерево имеет *логарифмическую сложность*. Разница между  $n$  и  $\log n$  является показателем повышения эффективности поиска в сбалансированном словаре по сравнению со списком. Но, к сожалению, такое правило соблюдается, только если дерево приблизительно сбалансировано. Если же дерево перестает быть таковым, эффективность поиска в нем снижается. В крайнем случае полностью несбалансированного дерева это дерево, по сути, превращается в список. В таком случае высота дерева становится равной  $n$  и эффективность поиска в дереве становится столь же низкой, как и в списке. Поэтому всегда необходимо стремиться к созданию сбалансированных словарей. Методы достижения этой цели будут описаны в главе 10.

## Упражнения

9.5. Определите следующие предикаты:

а) `binarytree( Object)`.

б) `dictionary( Object)`.

для определения того, является ли `Object` бинарным деревом или бинарным словарем; при написании этих процедур используйте обозначения данного раздела.

9.6. Определите процедуру

`height( BinaryTree, Height)`

для вычисления высоты бинарного дерева исходя из предположения, что высота пустого дерева равна 0, а одноэлементного дерева — 1.

9.7. Определите отношение

`linearize( Tree, List)`

позволяющее собрать все узлы дерева `Tree` в список.

9.8. Определите отношение

`maxelement( D, Item)`

такое, чтобы `Item` был наибольшим элементом, хранящимся в бинарном словаре `D`.

9.9. Модифицируйте процедуру

`in( Item, BinaryDictionary)`

добавив третий параметр, `Path`, такой, чтобы `Path` представлял собой путь между корнем словаря и элементом `Item`.

## 9.3. Вставка и удаление в бинарном словаре

При сопровождении динамического набора данных может потребоваться вставлять новые элементы в этот набор, а также удалять из него некоторые ненужные элементы. Поэтому для набора данных,  $S$ , может быть предусмотрен следующий обычный перечень операций.

- `in( X, S)`. Проверка принадлежности элемента  $X$  к набору данных  $S$ .

- `add( S, X, S1)`. Добавление элемента  $X$  к набору данных  $S$  и получение набора данных  $S1$ .
- `del( S, X, S1)`. Удаление элемента  $X$  из набора данных  $S$  и получение набора данных  $S1$ .

Определим отношение `add`. Проще всего вставлять новые данные на нижнем уровне дерева, чтобы новый элемент становился листом дерева в такой позиции, в которой сохраняется упорядочение этого дерева. На рис. 9.6 показано, какие изменения происходят в дереве при выполнении ряда операций вставки. Определим операцию вставки такого рода как `addleaf( D, X, D1)`.

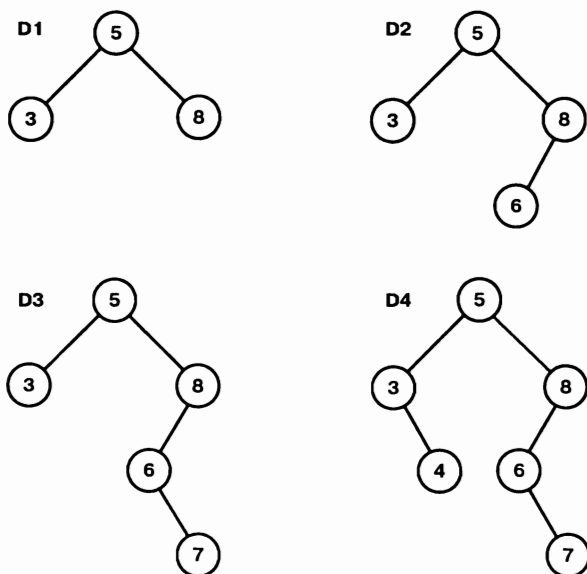


Рис. 9.6. Пример применения операций вставки в бинарный словарь на уровне листа; эти деревья соответствуют такой последовательности операций вставки: `add( D1, 6, D2)`, `add( D2, 7, D3)`, `add( D3, 4, D4)`

Правила вставки на уровне листа состоят в следующем.

- Результатом вставки элемента  $X$  в пустое дерево является дерево `t( nil, X, nil)`.
- Если  $X$  — корень  $D$ , то  $D1 = D$  (вставка дубликатов элементов не выполняется).
- Если корень  $D$  больше  $X$ , то вставить  $X$  в левое поддерево  $D$ ; если корень  $D$  меньше  $X$ , то вставить  $X$  в правое поддерево.

Соответствующая программа приведена в листинге 9.4.

#### Листинг 9.4. Вставка элемента в бинарный словарь в качестве листа

```

% addleaf(Tree, X, NewTree):
% вставка элемента X в качестве листа в бинарный словарь Tree
% и получение словаря NewTree

addleaf(nil, X, t(nil, X, nil)).

addleaf(t(Left, X, Right), X, t(Left, X, Right)).

```

```

addleaf(t(Left, Root, Right), X, t(Left1, Root, Right)) :-
 gt(Root, X),
 addleaf(Left, X, Left1).

addleaf(t(Left, Root, Right), X, t(Left, Root, Right1)) :-
 gt(X, Root),
 addleaf(Right, X, Right1).

```

---

Теперь рассмотрим операцию удаления. Задача удаления листа является простой, но обеспечить удаление внутреннего узла гораздо сложнее. Операцию удаления листа можно фактически определить как обратную по отношению к операции вставки на уровне листа, как показано ниже.

```

delleaf(D1, X, D2) :-
 addleaf(D2, X, D1).

```

К сожалению, если  $X$  — внутренний узел, то эта операция не может применяться в связи с возникновением проблемы, проиллюстрированной на рис. 9.7. Здесь  $X$  имеет два поддерева,  $Left$  и  $Right$ . После удаления  $X$  в дереве появится пустое место и поддерева  $Left$  и  $Right$  больше не будут соединены с остальной частью дерева. Эти поддерева нельзя также непосредственно присоединить к родительскому узлу  $X$  (к узлу  $A$ ), поскольку узел  $A$  способен присоединить к себе только одно из них.

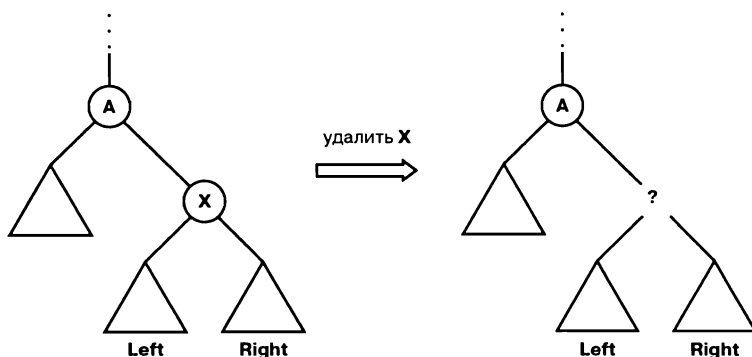


Рис. 9.7. Операция удаления элемента  $X$  из бинарного словаря, при которой возникает проблема, связанная с тем, что нужно восстановить структуру дерева после удаления  $X$

Если одно из поддеревьев ( $Left$  или  $Right$ ) является пустым, решение упрощается: непустое поддерево должно быть присоединено к узлу  $A$ . А если оба поддерева непусты, то можно воспользоваться одной идеей, которая проиллюстрирована на рис. 9.8. Крайний левый узел поддерева  $Right$  (узел  $Y$ ) переносится из его текущей позиции вверх для заполнения промежутка, образовавшегося в результате удаления узла  $X$ . После такого переноса дерево остается упорядоченным. Безусловно, та же идея допускает выполнение симметричной операции, предусматривающей перенос крайнего правого узла поддерева  $Left$ .

В соответствии с этими соображениями может быть составлена программа для операции удаления элемента из бинарного словаря, приведенная в листинге 9.5. Перенос крайнего левого узла правого поддерева осуществляется с помощью следующего отношения:

```
delmin(Tree, Y, Tree1)
```

где  $Y$  — наименьший (т.е. крайний левый) узел дерева  $Tree$ , а  $Tree1$  — дерево  $Tree$  после удаления элемента  $Y$ .

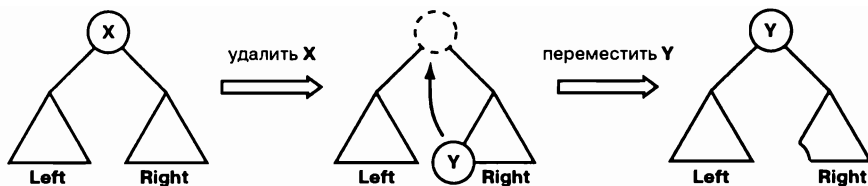


Рис. 9.8. Операция заполнения промежутка, образовавшегося в результате удаления узла X

### Листинг 9.5. Операция удаления из бинарного словаря

```
% del(Tree, X, NewTree):
% удаление элемента X из бинарного словаря Tree и получение словаря NewTree

del(t(nil, X, Right), X, Right).

del(t(Left, X, nil), X, Left).

del(t(Left, X, Right), X, t(Left, Y, Right1)) :-
 delmin(Right, Y, Right1).

del(t(Left, Root, Right), X, t(Left1, Root, Right)) :-
 gt(Root, X),
 del(Left, X, Left1).

del(t(Left, Root, Right), X, t(Left, Root, Right1)) :-
 gt(X, Root),
 del(Right, X, Right1).

% delmin(Tree, Y, NewTree):
% удаление минимального элемента X из бинарного словаря Tree
% и получение словаря NewTree

delmin(t(nil, Y, R), Y, R).

delmin(t(Left, Root, Right), Y, t(Left1, Root, Right)) :-
 delmin(Left, Y, Left1).
```

Но для разработки операций добавления и удаления можно использовать другое изящное решение. Отношение `add` может быть определено недетерминированным способом, таким образом, чтобы вставка нового элемента выполнялась на любом уровне дерева, а не только на уровне листа. Ниже описаны правила выполнения такой операции вставки.

Чтобы ввести элемент X в бинарный словарь D, необходимо выполнить одно из следующих действий.

- Ввести X в корень дерева D (так, чтобы X стал новым корнем).
- Если корень D больше X, то вставить X в левое поддерево D, в противном случае вставить X в правое поддерево D.

Сложной частью этой операции является вставка элемента в корень дерева D. Сформулируем эту операцию как следующее отношение:

```
addroot(D, X, D1)
```

где X — элемент, который должен быть вставлен в корень D, а D1 — результирующий словарь, корнем которого является X. Зависимости между X, D и D1 показаны на



рис. 9.9. Остается найти ответ на вопрос о том, какими должны быть поддеревья L1 и L2 на рис. 9.9 (или, при использовании другого варианта вставки, R1 и R2). Ответ на этот вопрос можно получить, рассмотрев следующие ограничения.

- L1 и L2 должны быть бинарными словарями.
- Множество узлов в L1 и L2 равно множеству узлов в L.
- Все узлы в L1 меньше X и все узлы в L2 больше X.

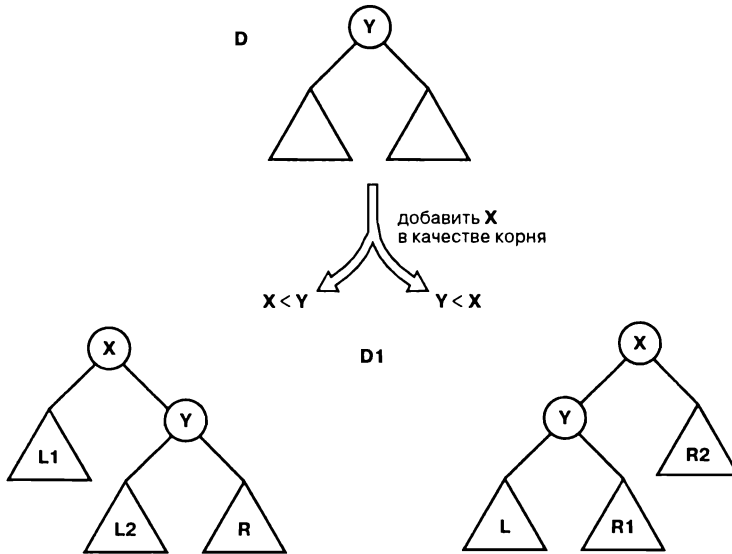


Рис. 9.9. Примеры вставки X в качестве корня бинарного словаря

Эти ограничения налагаются только одним отношением — `addroot`. А именно, если X должен быть введен в дерево L в качестве корня, то поддеревьями результирующего дерева как раз и должны быть L1 и L2. В языке Prolog L1 и L2 должны соответствовать следующей цели:

```
addroot(L, X, t(L1, X, L2))
```

Те же ограничения распространяются на R1 и R2:

```
addroot(R, X, t(R1, X, R2))
```

В листинге 9.6 приведена полная программа с определением операции “недетерминированной” вставки в бинарный словарь.

#### Листинг 9.6. Вставка в бинарный словарь на любом уровне дерева

```
% add(Tree, X, NewTree):
% вставка элемента X на любом уровне бинарного словаря Tree
% и получение словаря NewTree

add(Tree, X, NewTree) :-
 addroot(Tree, X, NewTree). % Добавление элемента X в качестве нового корня

add(t(L, Y, R), X, t(L1, Y, R)) :- % Вставка элемента X в левое поддерево
 gt(Y, X),
 add(L, X, L1).

add(t(L, Y, R), X, t(L, Y, R1)) :- % Вставка элемента X в правое поддерево
 gt(X, Y),
```

```

add(R, X, R1).

% addroot(Tree, X, NewTree):
% вставка элемента X в качестве корня в дерево Tree и получение дерева NewTree

addroot(nil, X, t(nil, X, nil)). % Вставка в пустое дерево

addroot(t(L, Y, R), X, t(L1, X, t(L2, Y, R))) :-
 gt(Y, X),
 addroot(L, X, t(L1, X, L2)).

addroot(t(L, Y, R), X, t(t(L, Y, R1), X, R2)) :-
 gt(X, Y),
 addroot(R, X, t(R1, X, R2)).

```

---

Важной отличительной особенностью этой процедуры вставки является то, что она не налагает ограничений на выбор уровня вставки. Поэтому такую процедуру `add` можно применить в обратном направлении, для удаления любого элемента из словаря. Например, следующий список целей формирует словарь `D`, содержащий элементы 3, 5, 1, 6, а затем удаляет элемент 5, что приводит к получению словаря `DD`:

```

add(nil, 3, D1), add(D1, 5, D2), add(D2, 1, D3),
add(D3, 6, D), add(DD, 5, D)

```

## 9.4. Отображение деревьев

Как и все объекты данных в языке Prolog, бинарное дерево `T` можно непосредственно вывести на внешнее устройство с помощью процедуры `write`. Но цель

```
write(T)
```

выводит лишь всю информацию, но не позволяет графически отобразить фактическую структуру дерева. Но задача воссоздания на бумаге фактической структуры дерева из термина Prolog, который представляет это дерево, может оказаться довольно утомительной. Поэтому часто возникает необходимость получить такую распечатку дерева, чтобы она графически показывала его структуру.

Ниже описан относительно простой метод отображения деревьев в такой форме. Весь его секрет состоит в том, чтобы дерево было отображено как растущее слева направо, а не сверху вниз, как обычно изображаются деревья. Дерево разворачивается влево, чтобы его корень стал крайним левым элементом, а листья сдвигаются вправо. Такой метод изображения деревьев демонстрируется на рис. 9.10.

Определим процедуру

```
show(T)
```

для вывода дерева `T` в форме, показанной на рис. 9.10. Эта процедура действует по описанному ниже принципу.

Чтобы вывести на внешнее устройство непустое дерево `T`, необходимо выполнить следующие действия.

1. Вывести правое поддереву `T`, обозначенное отступом вправо на некоторое расстояние `N`.
2. Вывести корень `T`.
3. Вывести левое поддереву `T`, обозначенное отступом вправо на некоторое расстояние `N`.

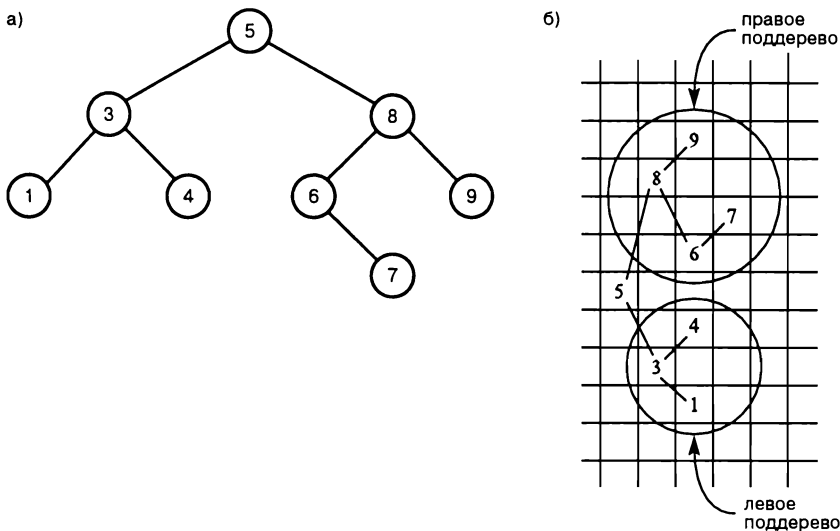


Рис. 9.10. Различные методы изображения деревьев: а) обычное изображение дерева; б) то же дерево, распечатанное процедурой *show* (дуги добавлены для наглядности)

Расстояние отступа  $N$ , которое может быть выбрано соответствующим образом, представляет собой дополнительный параметр процедуры вывода деревьев. После введения параметра  $N$  получим процедуру

```
show2(T, N)
```

для отображения дерева  $T$  с отступом на  $N$  пробелов от левого поля. Процедуры *show* и *show2* связаны между собой следующим отношением:

```
show(T) :- show2(T, 0).
```

Законченная программа, в которой применяются отступы на 2 пробела, показана в листинге 9.7. Принцип выполнения вывода в таком формате можно легко приспособить для отображения деревьев других типов.

### Листинг 9.7. Программа вывода бинарного дерева

```
% show(Tree): отображение бинарного дерева

show(Tree) :-
 show2(Tree, 0).

% show2(Tree, Indent): отображение дерева Tree с отступом Indent

show2(nil, _).

show2(t(Left, X, Right), Indent) :-
 Ind2 is Indent + 2,
 show2(Right, Ind2), % Отображение с отступом поддеревьев
 tab(Indent), write(X), nl, % Отображение правого поддерева
 show2(Left, Ind2). % Вывод элемента, соответствующего корню
 % Отображение левого поддерева
```

### Упражнение

9.10. Приведенная выше процедура отображения деревьев показывает дерево в непривычной ориентации, при которой корень находится слева, а листья дерева — справа. Напишите (более сложную) процедуру для отображения дерева в обычной ориентации, при которой корень находится сверху, а листья — внизу.

## 9.5. Графы

### 9.5.1. Представление графов

Структуры графов используются во многих приложениях, таких как представление отношений, ситуаций или проблем. Граф определяется как множество узлов и множество ребер, в котором каждое ребро соединяет пару узлов. Если ребра являются ориентированными, они именуются также *дугами*. Дуги представляются с помощью упорядоченных пар узлов. Такой граф называется *ориентированным*. Ребрам могут быть поставлены в соответствие стоимости, имена или обозначения любого рода, в зависимости от приложения. Примеры графов показаны на рис. 9.11.

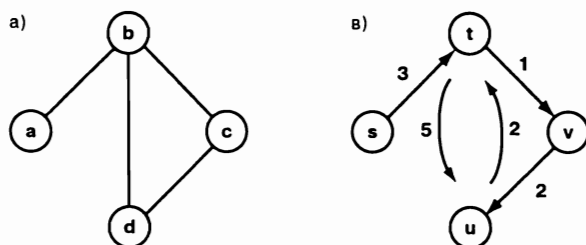


Рис. 9.11. Примеры графов: а) простой граф; б) ориентированный граф со стоимостями, назначенными дугам

Графы могут быть представлены на языке Prolog несколькими методами. Один из методов состоит в том, чтобы каждое ребро или каждая дуга были представлены отдельно, в виде одного предложения. Поэтому графы, показанные на рис. 9.11, могут быть представлены с помощью множеств предложений, например, следующим образом:

```
connected(a, b).
connected(b, c).
```

```
arc(s, t, 3).
arc(t, v, 1).
arc(u, t, 2).
```

Еще один метод состоит в том, что весь граф может быть представлен как один объект данных. Поэтому граф представляется как пара из двух множеств: узлы и ребра. Каждое множество узлов можно представить как список, а каждое ребро в множестве ребер — как пару узлов. Для объединения обоих множеств в пару выберем функтор `graph`, а для представления ребер будем применять функтор `e`. Таким образом, один из методов представления (неориентированного) графа, показанного на рис. 9.11, выглядит таким образом:

```
G1 = graph([a,b,c,d], [e(a,b), e(b,d), e(b,c), e(c,d)])
```

Для представления ориентированного графа можно выбрать функторы `digraph` и `a` (для дуг). Поэтому ориентированный граф, показанный на рис. 9.11, принимает следующий вид:

```
G2 = digraph([s,t,u,v], [a(s,t,3), a(t,v,1), a(t,u,5), a(u,t,2), a(v,u,2)])
```

Если каждый узел связан, по меньшей мере, еще с одним узлом, то можно исключить из этого представления список узлов, поскольку в таком случае множество узлов неявно задано списком ребер.

Еще один метод состоит в том, чтобы каждый узел был связан со списком узлов, смежных по отношению к этому узлу. В этом случае граф представляет собой список

пар, состоящий из узла и соответствующего ему списка смежности. Поэтому графы, рассматриваемые в качестве примеров, можно представить следующим образом:

```
G1 = [a -> [b], b -> [a,c,d], c -> [b,d], d -> [b,c]]
G2 = [s -> [t/3], t -> [u/5, v/1], u -> [t/2], v -> [u/2]]
```

Безусловно, приведенные выше символы “->” и “/” представляют собой инфиксные операторы.

Выбор наиболее подходящего представления зависит от приложения и от того, какие операции должны выполняться с графами. Две самые распространенные операции состоят в следующем.

- Поиск пути между двумя указанными узлами.
- Поиск в графе такого подграфа, который обладает некоторыми заданными свойствами.

Примером операции последнего типа является поиск остовного дерева графа. В следующих разделах рассматриваются некоторые простые программы поиска пути и формирования остовного дерева.

## 9.5.2. Поиск пути

Предположим, что  $G$  — граф, а  $A$  и  $Z$  — два узла в графе  $G$ . Определим следующее отношение:

```
path(A, Z, G, P)
```

где  $P$  — ациклический путь между узлами  $A$  и  $Z$  в графе  $G$ . Путь  $P$  представлен как список узлов в этом пути. Если  $G$  — граф, показанный на рис. 9.11, а, то имеют место следующие варианты пути:

```
path(a, d, G, [a,b,d])
path(a, d, G, [a,b,c,d])
```

Поскольку путь не должен содержать циклов, то любой узел может входить в состав пути не больше одного раза. Ниже описан один из методов поиска пути.

Чтобы найти ациклический путь  $P$  между узлами  $A$  и  $Z$  в графе  $G$ , необходимо выполнить следующие действия:

1. если  $A = Z$ , то  $P = [A]$ ;
2. в противном случае найти ациклический путь  $P_1$  от некоторого узла  $Y$  до  $Z$ , а затем найти пути от  $A$  до  $Y$ , избегая узлов, которые входят в путь  $P_1$ .

Эта формулировка указывает на то, что требуется еще одно отношение для поиска пути с учетом ограничения, согласно которому необходимо избегать использования некоторого подмножества узлов (обозначенного выше как  $P_1$ ). Поэтому определим еще одну процедуру следующим образом:

```
path1(A, Path1, G, Path)
```

Как показано на рис. 9.12, эта процедура имеет следующие параметры.

- $A$  — узел.
- $G$  — граф.
- $Path1$  — путь в графе  $G$ .
- $Path$  — ациклический путь в графе  $G$ , который проходит от  $A$  до начала  $Path1$  и продолжается вдоль  $Path1$  вплоть до его конца.

Отношения  $path$  и  $path1$  связаны между собой следующим отношением:

```
path(A, Z, G, Path) :- path1(A, [Z], G, Path).
```

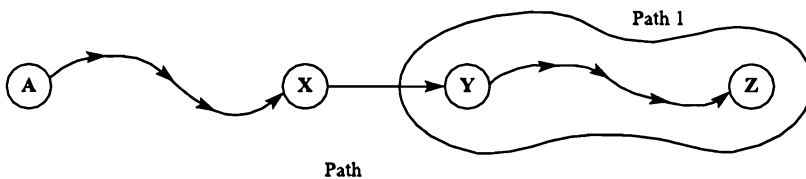


Рис. 9.12. Отношение *path1*, где *Path* — путь от *A* до *Z*; последний участок пути *Path* совпадает с *Path1*

Схема, приведенная на рис. 9.12, показывает, что отношение *path1* может быть определено рекурсивно. Как граничный случай может рассматриваться ситуация, при которой начальный узел пути *Path1* (узел *Y* на рис. 9.12) совпадает с начальным узлом пути *Path* (узлом *A*). Если же начальные узлы не совпадают, то должен быть узел *X*, такой, что:

1. *Y* является смежным по отношению к *X*; и
2. *X* не входит в состав пути *Path1*; и
3. *Path* должен соответствовать отношению *path1*(*A*, [*X* | *Path1*], *G*, *Path*).

Окончательный вариант программы показан в листинге 9.8. В этой программе отношение *member* представляет собой отношение проверки принадлежности к списку. Отношение

```
adjacent(X, Y, G)
```

означает, что в графе *G* существует дуга от *X* до *Y*. Определение этого отношения зависит от представления графа. Если *G* представлен в виде пары множеств (узлов и ребер) следующим образом:

```
G = graph(Nodes, Edges)
```

то должен применяться следующий вариант этого отношения:

```
adjacent(X, Y, graph(Nodes, Edges)) :-
 member(e(X,Y), Edges)
;
 member(e(Y,X), Edges).
```

### Листинг 9.8. Поиск ациклического пути *Path* от *A* до *Z* в графе *Graph*

```
% path(A, Z, Graph, Path): Path - ациклический путь от A до Z в графе Graph

path(A, Z, Graph, Path) :-
 path1(A, [Z], Graph, Path).

path1(A, [A | Path1], _, [A | Path1]).

path1(A, [Y | Path1], Graph, Path) :-
 adjacent(X, Y, Graph),
 not member(X, Path1), % Условие, исключающее образование цикла
 path1(A, [X, Y | Path1], Graph, Path).
```

Классической задачей теории графов является поиск *гамильтонова пути*; так называется ациклический путь, который охватывает все узлы в графе. Используя отношение *path*, эту операцию можно выполнить следующим образом:

```
hamiltonian(Graph, Path) :-
 path(_, _, Graph, Path),
 covers(Path, Graph).

covers(Path, Graph) :-
 not (node(N, Graph), not member(N, Path)).
```

где *node*(*N*, *Graph*) означает, что *N* — узел в графе *Graph*.

Пути могут быть поставлены в соответствие стоимости. *Стоимость пути* представляет собой сумму стоимостей дуг в этом пути. Если дугам не назначены стоимости, то можно вести речь не о стоимости, а о длине пути, считая, что каждая дуга в пути имеет стоимость, равную 1. Отношения `path` и `path1` можно модифицировать таким образом, чтобы в них учитывалась стоимость, введя дополнительный параметр, стоимость, для каждого пути следующим образом:

```
path(A, Z, G, P, C)
path1(A, P1, C1, G, P, C)
```

где `C` — стоимость пути `P` и `C1` — стоимость пути `P1`. В таком случае отношение `adjacent` также должно иметь дополнительный параметр — стоимость дуги. В листинге 9.9 приведена программа поиска пути, которая вычисляет путь и его стоимость.

**Листинг 9.9. Поиск пути в графе; Path — это ациклический путь со стоимостью Cost от A до Z в графе Graph**

```
% path(A, Z, Graph, Path, Cost):
% Path - ациклический путь со стоимостью Cost от A до Z в графе Graph

path(A, Z, Graph, Path, Cost) :-
 path1(A, [Z], 0, Graph, Path, Cost).

path1(A, [A | Path1], Cost1, Graph, [A | Path1], Cost1).

path1(A, [Y | Path1], Cost1, Graph, Path, Cost) :-
 adjacent(X, Y, CostXY, Graph),
 not member(X, Path1),
 Cost2 is Cost1 + CostXY,
 path1(A, [X, Y | Path1], Cost2, Graph, Path, Cost).
```

Эта процедура может также использоваться для поиска пути с минимальной стоимостью. Такой путь между двумя узлами (`node1` и `node2`) можно найти в некотором графе `Graph` с помощью следующих целей:

```
path(node1, node2, Graph, MinPath, MinCost),
not (path(node1, node2, Graph, _, Cost), Cost < MinCost)
```

Кроме того, можно найти путь с максимальной стоимостью между любой парой узлов в графе `Graph` с помощью следующих целей:

```
path(_, _, Graph, MaxPath, MaxCost),
not (path(_, _, Graph, _, Cost), Cost > MaxCost)
```

Необходимо отметить, что это — весьма неэффективный метод поиска пути с минимальной или максимальной стоимостью. В нем случайным образом исследуются все возможные пути, поэтому он полностью непригоден для больших графов, поскольку требует значительных затрат времени. Задача поиска пути часто возникает в проблематике искусственного интеллекта. Более приемлемые методы поиска оптимальных путей будут рассматриваться в главах 11 и 12.

### 9.5.3. Поиск остовного дерева графа

Граф называется связным, если в нем имеется путь от любого узла до любого другого узла. Предположим, что  $G = (V, E)$  — связный граф с множеством узлов  $V$  и множеством ребер  $E$ . Остовным деревом графа  $G$  является связный граф  $T = (V, E')$ , где  $E'$  — подмножество множества  $E$ , такое, что:

1.  $T$  является связным;
2. в  $T$  нет ни одного цикла.

Эти два условия гарантируют, что  $T$  представляет собой дерево. Для графа, показанного на рис. 9.11, а, имеются три остовных дерева, которые соответствуют следующим трем спискам ребер:

```
Tree1 = [a-b, b-c, c-d]
```

```
Tree2 = [a-b, b-d, d-c]
```

```
Tree3 = [a-b, b-d, b-c]
```

где каждый терм в форме  $X-Y$  обозначает ребро между узлами  $X$  и  $Y$ . В таком списке в качестве корня дерева может быть выбран любой узел. Остовные деревья представляют интерес, например, в проблематике сетей связи, поскольку они позволяют создавать соединения между любыми парами узлов с помощью минимального количества линий связи.

Определим следующую процедуру:

```
stree(G, T)
```

где  $T$  — остовное дерево графа  $G$ . Мы исходим из предположения, что граф  $G$  является связным. Алгоритмически можно представить процесс формирования остовного дерева следующим образом: начать с пустого множества ребер и постепенно добавлять из графа  $G$  все новые ребра, следя за тем, чтобы не создавался цикл, до тех пор, пока не возникнет ситуация, при которой больше нельзя будет добавлять ребра, поскольку это приведет к созданию цикла. Полученное в результате множество ребер определяет остовное дерево. За соблюдением условия отсутствия цикла можно следить с помощью следующего простого правила: любое ребро можно добавлять, только если один из его узлов уже находится в растущем дереве, а другой узел — еще нет. Программа, которая реализует этот замысел, приведена в листинге 9.10. Основным отношением этой программы является следующее:

```
spread(Tree1, Tree, G)
```

#### Листинг 9.10. Поиск остовного дерева графа: “алгоритмическая” программа; в этой программе принято предположение, что граф является связным

```
% Поиск остовного дерева графа
%
% Деревья и графы представлены списками их ребер.
% Например, Graph = [a-b, b-c, b-d, c-d]

% stree(Graph, Tree): Tree - остовное дерево графа Graph

stree(Graph, Tree) :-
 member(Edge, Graph),
 spread([Edge], Tree, Graph).

% spread(Tree1, Tree, Graph): дерево Tree1 "расширяется"
% до остовного дерева Tree графа Graph

spread(Tree1, Tree, Graph) :-
 addedge(Tree1, Tree2, Graph),
 spread(Tree2, Tree, Graph).

spread(Tree, Tree, Graph) :-
 not addedge(Tree, _, Graph). % Нельзя ввести ни одного ребра, не создав цикл
% addedge(Tree, NewTree, Graph):
% добавить ребро графа Graph в дерево Tree, не создавая цикл

addege(Tree, [A-B | Tree], Graph) :-
 adjacent(A, B, Graph), % Узлы A и B в графе Graph являются смежными
 node(A, Tree), % Узел A находится в дереве Tree
 not node(B, Tree). % Ребро A-B не создает цикл в дереве Tree

adjacent(Node1, Node2, Graph) :-
 member(Node1-Node2, Graph)
 ;
 member(Node2-Node1, Graph).
```



---

```
node(Node, Graph) :- % Node -узел графа Graph, если
 adjacent(Node, _, Graph). % он является смежным с любым узлом графа Graph
```

---

Все три параметра этого отношения представляют собой множества ребер.  $G$  — это связный граф,  $Tree1$  и  $Tree$  — подмножества графа  $G$ , такие, что оба они являются деревьями.  $Tree$  — это остовное дерево графа  $G$ , полученное путем добавления ребер графа  $G$  в дерево  $Tree1$  (в количестве от нуля и больше). Такую операцию можно описать словесно так, что “дерево  $Tree1$  было расширено до  $Tree$ ” (отсюда и название процедуры — `spread`).

Любопытно отметить, что может быть также создана работоспособная программа для формирования остовного дерева на основе иного, полностью декларативного подхода, при котором задаются математические определения. Предположим, что и графы, и деревья представлены в виде списков их ребер, как в программе из листинга 9.10. Для разработки программы необходимо ввести приведенные ниже определения.

1.  $T$  — остовное дерево  $G$ , если одновременно выполняются следующие условия:

- $T$  — подмножество  $G$ ,
- $T$  — дерево,
- $T$  “покрывает”  $G$ , т.е. каждый узел  $G$  находится также в  $T$ .

2. Множество ребер  $T$  представляет собой дерево, если одновременно выполняются следующие условия:

- $T$  связно,
- $T$  не имеет циклов.

С помощью программы `path` (см. предыдущий раздел) эти определения можно сформулировать на языке Prolog, как показано в листинге 9.11. Но следует отметить, что данная программа в такой форме не представляет практического интереса из-за ее неэффективности.

**Листинг 9.11. Поиск остовного дерева графа: “декларативная” программа; отношения `node` и `adjacent` приведены в листинге 9.10**

---

```
% Поиск остовного дерева
% Графы и деревья представлены как списки ребер

% stree(Graph, Tree): Tree - остовное дерево графа Graph

stree(Graph, Tree) :-
 subset(Graph, Tree),
 tree(Tree),
 covers(Tree, Graph).

tree(Tree) :-
 connected(Tree),
 not hasacycle(Tree).
% connected(Graph): есть путь между любыми двумя узлами в графе Graph

connected(Graph) :-
 not (node(A, Graph), node(B, Graph), not path(A, B, Graph, _)).

hasacycle(Graph) :-
 adjacent(Node1, Node2, Graph),
 path(Node1, Node2, Graph, [Node1, X, Y | _]). % Путь имеет длину больше 1

% covers(Tree, Graph): каждый узел графа Graph находится в дереве Tree

covers(Tree, Graph) :-
 not (node(Node, Graph), not node(Node, Tree)).
```

```
% subset(List1, List2): список List2 представляет собой подмножество List1
subset([], []).

subset([X | Set], Subset) :- % Элемент X не находится в подмножестве
 subset(Set, Subset).

subset([X | Set], [X | Subset]) :- % Элемент X включен в подмножество
 subset(Set, Subset).
```

---

## Упражнения

- 9.11. Рассмотрите процесс формирования остовных деревьев графов, в которых ребрам назначены стоимости. Допустим, что стоимость остовного дерева определена как сумма стоимостей всех ребер дерева. Напишите программу для поиска остовного дерева заданного графа с минимальной стоимостью.
- 9.12. Проведите эксперименты с программами формирования остовных деревьев, приведенными в листингах 9.10 и 9.11, и измерьте время их выполнения. Выявите причины неэффективности второй программы.

## Резюме

В этой главе рассматривались реализации на языке Prolog некоторых часто используемых структур данных и соответствующих операций над ними. К ним относятся структуры и операции, перечисленные ниже.

### ■ Списки:

- сортировка списков (пузырьковая сортировка, сортировка вставкой, быстрая сортировка);
- эффективность этих процедур.

### ■ Представление множеств в виде *бинарных деревьев* и *бинарных словарей*:

- поиск элемента в дереве;
- добавление элемента;
- удаление элемента;
- добавление в качестве листа, добавление в качестве корня;
- формирование сбалансированных деревьев, зависимость перечисленных выше операций от сбалансированности деревьев;
- отображение деревьев.

### ■ Графы:

- представление графов;
- поиск путей в графе;
- поиск остовного дерева графа.

## Дополнительные источники информации

В этой главе классические темы сортировки и сопровождения структур данных, применяемых для представления множеств, рассматривались с точки зрения языка Prolog. Эти темы представлены в книгах, которые в целом посвящены алгоритмам и структурам данных (например, [2], [3], [37], [61] и [73]). Программу Prolog для вставки на любом уровне бинарного дерева (см. раздел 9.3) впервые продемонстрировал автору (при личном общении) M. van Emden (М. ван Эмден).

# Усовершенствованные методы представления деревьев

*В этой главе...*

|                                                        |     |
|--------------------------------------------------------|-----|
| 10.1. Двоично-троичный словарь                         | 215 |
| 10.2. AVL-дерево — приближенно сбалансированное дерево | 221 |

В этой главе рассматриваются усовершенствованные методы представления наборов данных с помощью деревьев. Главной особенностью этих методов является то, что они позволяют поддерживать дерево в сбалансированном или примерно сбалансированном состоянии для предотвращения его вырождения в список. Такие схемы сопровождения сбалансированных деревьев даже в самом неблагоприятном случае гарантируют относительно быстрый доступ к данным за время, пропорциональное логарифму количества элементов. В настоящей главе представлены две такие схемы: двоично-троичные (2-3) и AVL-деревья. (Материал, приведенный в данной главе, не является обязательным для изучения других глав.)

## 10.1. Двоично-троичный словарь

Бинарное дерево называется *полностью сбалансированным*, если оба его поддерева имеют примерно одинаковую высоту (или размер) и также являются сбалансированными. Высота сбалансированного дерева приблизительно равна  $\log n$ , где  $n$  — количество узлов в дереве. Время, необходимое для вычисления отношений *in*, *add* и *delete* на бинарных словарях, возрастает пропорционально высоте дерева. Поэтому все эти операции на сбалансированных словарях могут быть выполнены за время, пропорциональное  $\log n$ . Логарифмический рост затрат времени на проверку принадлежности к множеству представляет собой значительное улучшение по сравнению со способом представления множеств в виде списков, при котором затраты времени растут пропорционально размеру набора данных. Но если дерево плохо сбалансировано, эффективность доступа к словарю снижается. В крайних случаях бинарный словарь вырождается в список, как показано на рис. 10.1. Форма словаря зависит от того, в какой последовательности в него вставлялись данные. В лучшем случае обеспечивается хорошая сбалансированность, при которой эффективность доступа пропорциональна  $\log n$ , а в худшем случае эффективность доступа пропорциональна  $n$ . Анализ показывает, что в среднем, при условии, что любая последовательность ввода данных является равновероятной, эффективность выполнения отношений *in*, *add* и *delete* все равно остается пропорциональной  $\log n$ . Поэтому, к счастью, эффективность в среднем ближе к наилучшему, чем к наихудшему случаю. Но существуют и другие довольно простые схемы обеспечения хорошей сбалансированности дерева независимо от последовательности данных. Такие схемы гарантируют эффективность

выполнения отношений `in`, `add` и `delete`, даже в худшем случае пропорциональную  $\log n$ . Одной из них являются двоично-троичные (сокращенно 2-3) деревья, а другой — AVL-деревья.

Двоично-троичное дерево определяется следующим образом. Оно является либо пустым, либо состоящим из единственного узла, либо представляет собой дерево, которое соответствует следующим условиям.

- Каждый внутренний узел имеет два или три дочерних узла.
- Все листья находятся на одном и том же уровне.

Двоично-троичный словарь представляет собой двоично-троичное дерево, в котором элементы данных хранятся в листьях, упорядоченных слева направо. (Пример такого дерева приведен на рис. 10.2.) Внутренние узлы содержат метки, которые обозначают минимальные элементы поддеревьев следующим образом.

- Если внутренний узел имеет два поддерева, то он содержит минимальный элемент второго поддерева.
- Если внутренний узел имеет три поддерева, то он содержит минимальные элементы второго и третьего поддеревьев.

Для поиска элемента  $X$  в двоично-троичном словаре необходимо начинать с корня и переходить на нижний уровень, руководствуясь метками на внутренних узлах. Предположим, что корень содержит метки  $M_1$  и  $M_2$ . В таком случае должны выполняться следующие действия:

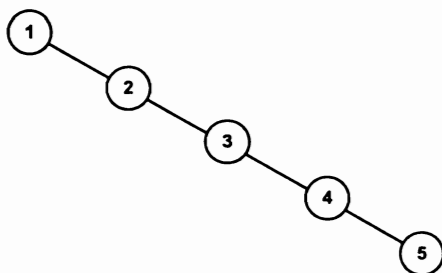


Рис. 10.1. Полностью несбалансированный бинарный словарь; эффективность доступа к нему снижается до уровня эффективности доступа к списку

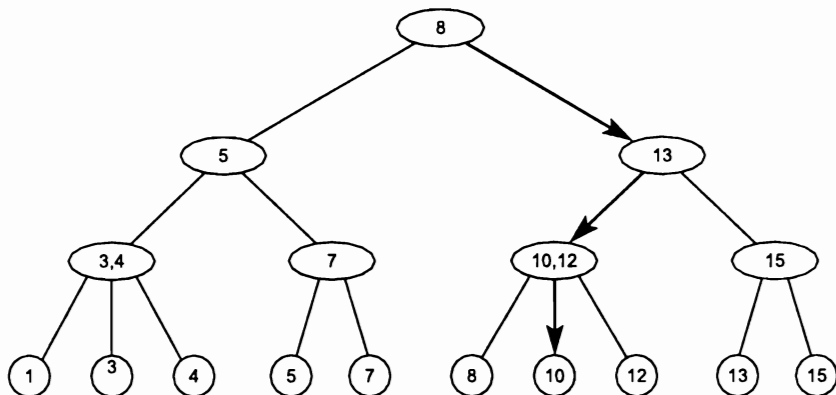


Рис. 10.2. Двоично-троичный словарь; указанный путь соответствует последовательности поиска элемента 10

- если  $X < M_1$ , то продолжать поиск в левом поддереве;
- иначе, если  $X < M_2$ , то продолжать поиск в среднем поддереве;
- иначе продолжать поиск в правом поддереве.

Еще один вариант состоит в том, что корень содержит только одну метку,  $M$ . В таком случае, если  $X < M$ , то поиск должен продолжаться в левом поддереве, а иначе — в правом поддереве. Такие операции поиска продолжаются до тех пор, пока не будет достигнут уровень листьев, и в этот момент либо успешно обнаруживается элемент  $X$ , либо поиск завершается неудачей.

Поскольку все листья находятся на одном и том же уровне, двоично-троичное дерево является идеально сбалансированным по отношению к значениям высоты своих поддеревьев. Все пути поиска от корня до любого листа имеют одинаковую длину, пропорциональную  $\log n$ , где  $n$  — количество элементов, хранящихся в дереве.

При вставке новых данных двоично-троичное дерево может также расти в ширину, а не только в глубину. Каждый внутренний узел, имеющий два дочерних узла, может включить дополнительный дочерний узел, что приводит к росту дерева в ширину. С другой стороны, если узел с тремя дочерними узлами принимает еще один дочерний узел, то разделяется на два узла, каждый из которых принимает два дочерних узла из общего количества, равного четырем. Созданный таким образом новый внутренний узел вставляется в дерево на более высоком уровне. Если это происходит на самом верхнем уровне, то возникает необходимость увеличить количество уровней дерева. Описанные выше действия проиллюстрированы на рис. 10.3.

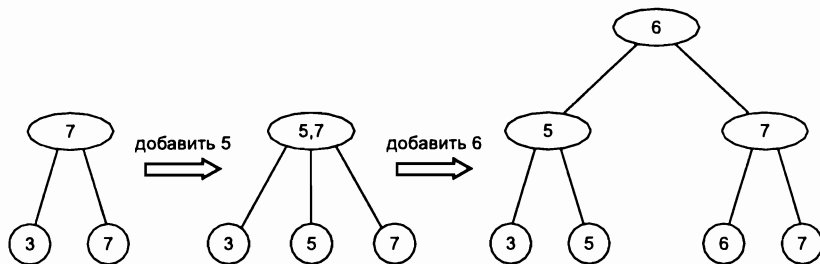


Рис. 10.3. Вставка элементов в двоично-троичный словарь; дерево вначале растет в ширину, а затем — в высоту

Операцию вставки элементов в двоично-троичный словарь можно запрограммировать как следующее отношение:

```
add23(Tree, X, NewTree)
```

где *NewTree* — дерево, полученное в результате вставки элемента  $X$  в дерево *Tree*. Основная работа по вставке элементов будет передана двум вспомогательным отношениям, которые оба именуются как *ins*. Первое из них имеет следующие три параметра:

```
ins(Tree, X, NewTree)
```

где *NewTree* — результат вставки  $X$  в дерево *Tree*. Деревья *Tree* и *NewTree* имеют одинаковую высоту. Но, безусловно, не всегда возможно сохранить после вставки прежнюю высоту. Поэтому предусмотрено еще одно отношение *ins* с пятью параметрами, позволяющее учесть эту ситуацию, следующим образом:

```
ins(Tree, X, NTa, Mb, NTb)
```

В данном случае после вставки элемента  $X$  в дерево *Tree* последнее разделяется на два дерева, *NTa* и *NTb*, имеющие такую же высоту, как и *Tree*. Здесь *Mb* является минимальным элементом дерева *NTb*. Пример ситуации, в которой дерево необходимо разбить на два, показан на рис. 10.4.

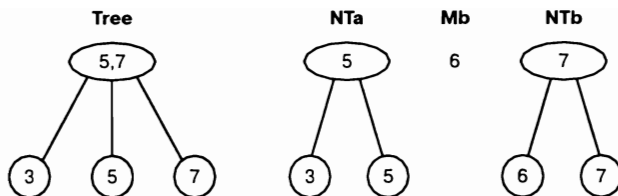


Рис. 10.4. Пример, в котором объекты соответствуют отношению `ins( Tree, 6, NTa, Mb, NTb)`

В разрабатываемой программе двоично-троичное дерево должно быть представлено, в зависимости от его формы, следующим образом.

- `nil` представляет пустое дерево.
- `l(X)` представляет дерево с одним узлом — листом с элементом `X`.
- `n2(T1, M, T2)` представляет дерево с двумя поддеревьями, `T1` и `T2`; `M` — минимальный элемент `T2`.
- `n3(T1, M2, T2, M3, T3)` представляет дерево с тремя поддеревьями, `T1`, `T2` и `T3`; `M2` — минимальный элемент `T2`, а `M3` — минимальный элемент `T3`.

Здесь `T1`, `T2` и `T3` являются двоично-троичными деревьями.

Отношение между `add23` и `ins` характеризуется тем, что если после вставки элемента дерево не растет вверх, то имеет место следующее правило:

```
add23(Tree, X, NewTree) :-
 ins(Tree, X, NewTree).
```

Но если высота дерева после вставки увеличивается, то отношение `ins` определяет два поддерева, `T1` и `T2`, которые затем объединяются в более крупное дерево:

```
add23(Tree, X, n2(T1, M, T2)) :-
 ins(Tree, X, T1, M, T2).
```

Отношение `ins` является более сложным, поскольку в нем должно учитываться много вариантов: вставка в пустое дерево, в дерево с одним узлом, в дерево типа `n2` или `n3`. Дополнительные промежуточные варианты возникают в результате вставки в первое, второе или третье поддерево. Соответственно, отношение `ins` определено как набор правил таким образом, что в каждом предложении, касающемся `ins`, рассматривается один из вариантов. Некоторые из этих вариантов приведены на рис. 10.5. Варианты, показанные на данном рисунке, можно представить на языке Prolog следующим образом.

## Вариант А

```
ins(n2(T1, M, T2), X, n2(NT1, M, T2)) :-
 gt(M, X), % M больше чем X
 ins(T1, X, NT1).
```

## Вариант В

```
ins(n2(T1, M, T2), X, n3(NT1a, Mb, NT1b, M, T2)) :-
 gt(M, X),
 ins(T1, X, NT1a, Mb, NT1b).
```

## Вариант В

```
ins(n3(T1, M2, T2, M3, T3), X, n2(NT1a, Mb, NT1b), M2, n2(T2, M3, T3)) :-
 gt(M2, X),
 ins(T1, X, NT1a, Mb, NT1b).
```

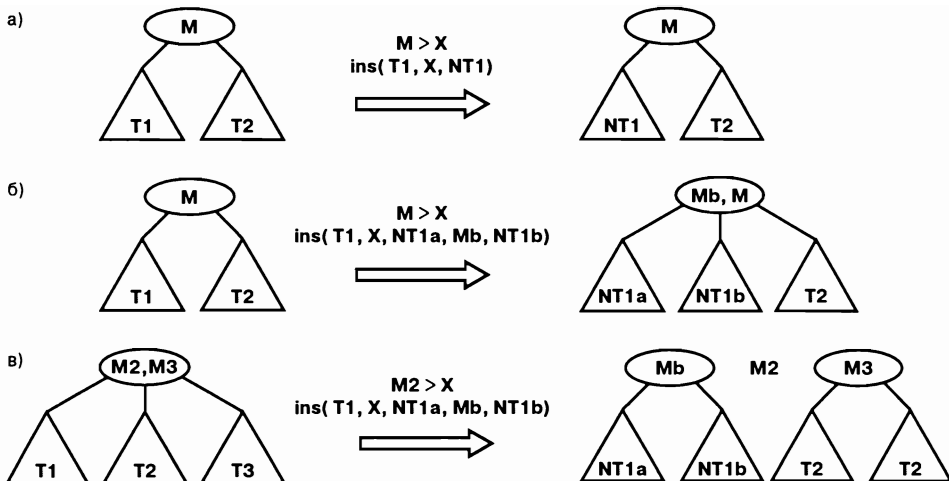


Рис. 10.5. Некоторые варианты использования отношения *ins*: а) *ins*( *n2* ( *T1*, *M*, *T2* ), *X*, *n2* ( *NT1*, *M*, *T2* )); б) *ins*( *n2* ( *T1*, *M*, *T2* ), *X*, *n3* ( *NT1a*, *Mb*, *NT1b*, *M*, *T2* )); в) *ins*( *n3* ( *T1*, *M2*, *T2*, *M3*, *T3* ), *X*, *n2* ( *NT1a*, *Mb*, *NT1b* ), *M2*, *n2* ( *T2*, *M3*, *T3* ))

В листинге 10.1 приведена полная программа вставки в двоично-троичный словарь, а в листинге 10.2 приведена программа отображения двоично-троичных деревьев.

#### Листинг 10.1. Программа вставки в двоично-троичный словарь; в этой программе попытки вставки дублирующегося элемента завершаются неудачей

% Вставка в двоично-троичный словарь

```
add23(Tree, X, Tree1) :- % Ввести X в дерево Tree, получив дерево Tree1
 ins(Tree, X, Tree1). % Дерево растёт в ширину

add23(Tree, X, n2(T1, M2, T2)) :- % Дерево растёт в высоту
 ins(Tree, X, T1, M2, T2).

del23(Tree, X, Tree1) :- % Удалить X из дерева Tree, получив дерево Tree1
 add23(Tree1, X, Tree).

ins(nil, X, l(X)).

ins(l(A), X, l(A), X, l(X)) :-
 gt(X, A).

ins(l(A), X, l(X), A, l(A)) :-
 gt(A, X).

ins(n2(T1, M , T2), X, n2(NT1, M, T2)) :-
 gt(M, X),
 ins(T1, X, NT1).

ins(n2(T1, M, T2), X, n3(NT1a, Mb, NT1b, M, T2)) :-
 gt(M, X),
 ins(T1, X, NT1a, Mb, NT1b).

ins(n2(T1, M, T2), X, n2(T1, M, NT2)) :-
 gt(X, M),
 ins(T2, X, NT2).

ins(n2(T1, M, T2), X, n3(T1, M, NT2a, Mb, NT2b)) :-
 gt(X, M),
```

```

ins(T2, X, NT2a, Mb, NT2b).

ins(n3(T1, M2, T2, M3, T3), X, n3(NT1, M2, T2, M3, T3)) :-
 gt(M2, X),
 ins(T1, X, NT1).

ins(n3(T1, M2, T2, M3, T3), X, n2(NT1a, Mb, NT1b), M2, n2(T2, M3, T3)) :-
 gt(M2, X),
 ins(T1, X, NT1a, Mb, NT1b).

ins(n3(T1, M2, T2, M3, T3), X, n3(T1, M2, NT2, M3, T3)) :-
 gt(X, M2), gt(M3, X),
 ins(T2, X, NT2).

ins(n3(T1, M2, T2, M3, T3), X, n2(T1, M2, NT2a), Mb, n2(NT2b, M3, T3)) :-
 gt(X, M2), gt(M3, X),
 ins(T2, X, NT2a, Mb, NT2b).

ins(n3(T1, M2, T2, M3, T3), X, n3(T1, M2, T2, M3, NT3)) :-
 gt(X, M3),
 ins(T3, X, NT3).

ins(n3(T1, M2, T2, M3, T3), X, n2(T1, M2, T2), M3, n2(NT3a, Mb, NT3b)) :-
 gt(X, M3),
 ins(T3, X, NT3a, Mb, NT3b).

```

**Листинг 10.2. Программа отображения двоично-троичного словаря (слева) и словарь, который приведен на рис. 10.2, отображенный с помощью этой программы (справа)**

‡ Отображение двоично-троичного словаря

```

show(T) :-
 show(T,0).

show(nil, _).

show(l(A),H) :-
 tab(H), write(A), nl.

show(n2(T1,M,T2), H) :-
 H1 is H+5,
 show(T2, H1),
 tab(H), write(--), nl,
 tab(H), write(M), nl,
 tab(H), write(--), nl,
 show(T1, H1).

show(n3(T1, M2, T2, M3, T3), H) :-
 H1 is H+5,
 show(T3, H1),
 tab(H), write(--), nl,
 tab(H), write(M3), nl,
 show(T2, H1),
 tab(H), write(M2), nl,
 tab(H), write(--), nl,
 show(T1, H1).

```

|    |    |
|----|----|
|    | 15 |
| -- |    |
| 15 |    |
| -- |    |
|    | 13 |
| -- |    |
| 13 |    |
| -- |    |
|    | 12 |
| -- |    |
| 12 |    |
| 10 |    |
| -- |    |
| 8  |    |
| -- |    |
| 8  |    |
| -- |    |
|    | 7  |
| -- |    |
| 7  |    |
| -- |    |
|    | 5  |
| -- |    |
| 5  |    |
| -- |    |
|    | 4  |
| -- |    |
| 4  |    |
| 3  |    |
| -- |    |
|    | 1  |



В данной программе иногда происходит ненужный перебор с возвратами. Например, если выполнение отношения `ins` с тремя параметрами завершается неудачей, то вызывается отношение `ins` с пятью параметрами и при этом отменяется часть выполненной работы. Эту причину снижения эффективности можно легко устранить, например, переопределив отношение `ins` следующим образом:

```
ins2(Tree, X, NewTrees)
```

где `NewTrees` — список, имеющий длину 1 или 3, который соответствует таким условиям:

```
NewTrees = [NewTree], если ins(Tree, X, NewTree)
```

```
NewTrees = [NTa, Mb, NTb], если ins(Tree, X, NTa, Mb, NTb)
```

В связи с этим отношение `add23` должно быть переопределено таким образом:

```
add23(T, X, T1) :-
 ins2(T, X, Trees),
 combine(Trees, T1).
```

Отношение `combine` должно формировать одно дерево, `T1`, из списка `Trees`.

## Упражнения

### 10.1. Определите отношение

```
in(Item, Tree)
```

для поиска элемента `Item` в двоично-троичном словаре `Tree`.

### 10.2. Откорректируйте программу, приведенную в листинге 10.1, для предотвращения перебора с возвратами (определите отношения `ins2` и `combine`).

## 10.2. AVL-дерево — приближенно сбалансированное дерево

AVL-дерево — это бинарное дерево, которое обладает следующими свойствами.

1. Его левое и правое поддеревья отличаются по высоте не больше чем на 1.
2. Оба поддерева являются AVL-деревьями.

Это определение допускает возможность формирования немного несбалансированных деревьев. Можно показать, что высота AVL-дерева всегда грубо пропорциональна  $\log n$ , где  $n$  — количество узлов в дереве (даже в худшем случае). Это позволяет гарантировать эффективность операций `in`, `add` и `del`, пропорциональную логарифму количества элементов.

Операции с AVL-словарями аналогичны операциям с бинарными словарями, если не считать некоторых дополнений, позволяющих поддерживать приблизительную сбалансированность дерева. Если дерево выходит из состояния приблизительной сбалансированности после вставки или удаления элемента, то некоторый дополнительный механизм снова восстанавливает требуемую степень сбалансированности. Для эффективной реализации этого механизма необходимо сопровождать определенную информацию о сбалансированности дерева. По сути требуется знать только разницу между высотами поддеревьев дерева, которая может быть равна -1, 0 или +1. Но для упрощения выполняемых операций желательно сопровождать информацию о полной высоте деревьев, а не только о разнице высот.

Определим отношение вставки следующим образом:

```
addavl(Tree, X, NewTree)
```

где и `Tree`, и `NewTree` — AVL-словари, такие, что дерево `NewTree` представляет собой дерево `Tree` со вставленным элементом `X`. AVL-деревья будут представлены с помощью термов в следующей форме:

```
t(Left, A, Right)/Height
```

где  $A$  — корень,  $Left$  и  $Right$  — поддеревья, а  $Height$  — высота дерева. Пустое дерево представлено в виде  $nil/0$ . Теперь рассмотрим операцию вставки элемента  $X$  в непустой AVL-словарь, как показано ниже.

$Tree = t(L, A, R)/H$

Начнем описание с изучения только того варианта, в котором  $X$  больше  $A$ . После этого вставим  $X$  в дерево  $R$  и получим следующее отношение:

$addavl(R, X, t(R1, B, R2)/Hb)$

На рис. 10.6 показаны следующие компоненты, из которых должно быть создано дерево  $NewTree$ :

$L, A, R1, B, R2$

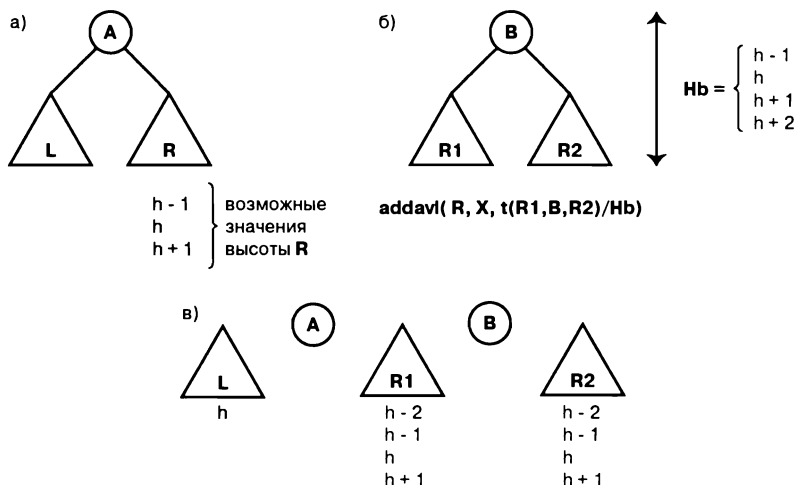


Рис. 10.6. Решение задачи вставки элемента в AVL-дерево: а) AVL-дерево перед вставкой  $X$ ,  $X > A$ ; б) AVL-дерево после вставки  $X$  в дерево  $R$ ; в) компоненты, из которых должно быть сформировано новое дерево

Каковыми могут быть значения высоты  $L$ ,  $R$ ,  $R1$  и  $R2$ ? Деревья  $L$  и  $R$  могут отличаться по высоте не больше чем на 1. На рис. 10.6 показано, какими могут быть значения высоты деревьев  $R1$  и  $R2$ . Поскольку в дерево  $R$  вставлен только один элемент,  $X$ , то не более чем одно из поддеревьев ( $R1$  или  $R2$ ) может иметь высоту  $h + 1$ .

В том случае, если  $X$  меньше  $A$ , ситуация является аналогичной, за исключением того, что левое и правое поддерева меняются местами. Поэтому в любом случае необходимо сформировать дерево  $NewTree$  из трех деревьев (назовем их  $Tree1$ ,  $Tree2$  и  $Tree3$ ) и двух отдельных элементов,  $A$  и  $B$ . Теперь рассмотрим вопрос о том, как объединить эти пять компонентов, чтобы сформировать дерево  $NewTree$ , представляющее собой AVL-словарь. Порядок компонентов в дереве  $NewTree$  слева направо должен быть следующим:

$Tree1, A, Tree2, B, Tree3$

Необходимо рассмотреть следующие варианты.

1. Среднее дерево,  $Tree2$ , является более высоким, чем оба других дерева.
2. Дерево  $Tree1$  является, по меньшей мере, таким же высоким, как деревья  $Tree2$  и  $Tree3$ .
3. Дерево  $Tree3$  является, по меньшей мере, таким же высоким, как деревья  $Tree2$  и  $Tree1$ .

На рис. 10.7 показано, как можно сформировать дерево NewTree в каждом из этих вариантов. В варианте 1 среднее дерево (Tree2) необходимо разделить на части и включить их в дерево NewTree. Три правила, которые приведены на рис. 10.7, можно легко перевести на язык Prolog в виде следующего отношения:

```
combine(Tree1, A, Tree2, B, Tree3, NewTree)
```

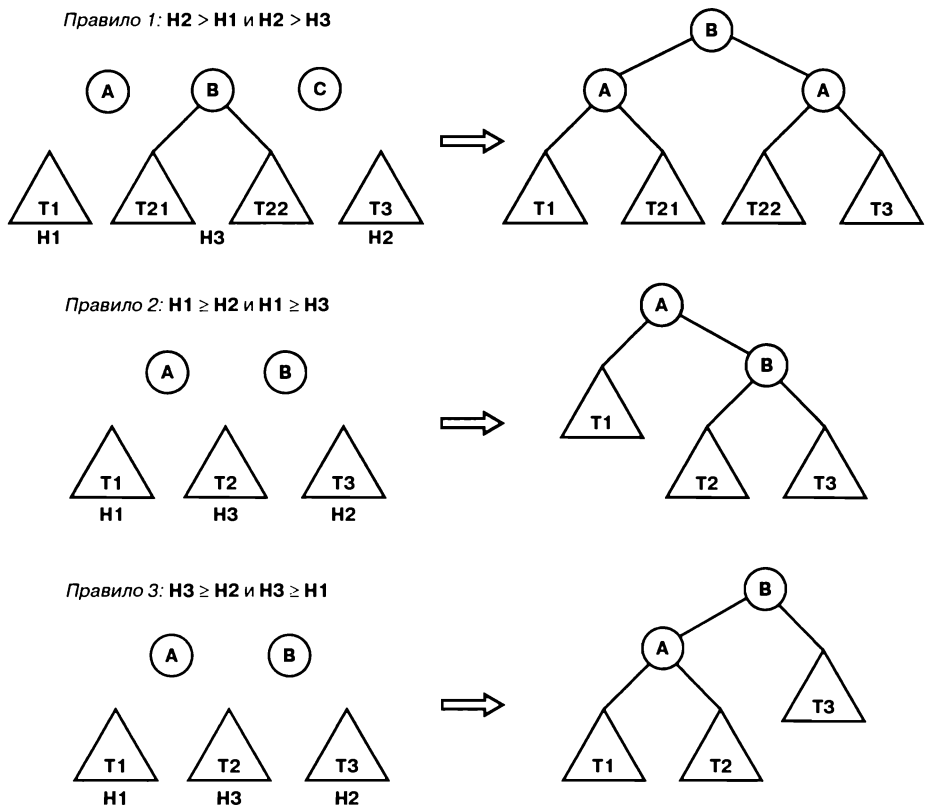


Рис. 10.7. Три правила формирования AVL-деревьев

Последний параметр, NewTree, представляет собой AVL-дерево, сформированное из пяти компонентов, которые заданы первыми пятью параметрами. Например, правило 1 принимает следующий вид:

```
combine(
 T1/H1, A, t(T21,B,T22) / H2, C, T3/H3, % Пять компонентов
 t(t(T1/H1,A,T21)/Ha, B, t(T22,C,T3/H3)/Hc)/Hb) :- % Их сочетание
 H2 > H1, H2 > H3, % Среднее дерево - самое высокое
 Ha is H1 + 1, % Высота левого поддерева
 Hc is H3 + 1, % Высота правого поддерева
 Hb is Ha + 1. % Высота всего дерева
```

Полная версия программы addavl, которая вычисляет также значения высоты дерева и поддеревьев, приведена в листинге 10.3.

**Листинг 10.3. Программа вставки в AVL-словарь; в этой программе попытка вставки дублирующегося элемента оканчивается неудачей (определение отношения combine см. на рис. 10.7)**

```
% addavl(Tree, X, NewTree): вставка в AVL-словарь
% Tree = t(Left, Root, Right)/HeightOfTree
% Пустое дерево - nil/0

addavl(nil/0, X, t(nil/0,X,nil/0)/1). % Ввести элемент X в пустое дерево
addavl(t(L,Y,R)/Hy, X, NewTree) :- % Ввести элемент X в непустое дерево
 gt(Y, X),
 addavl(L, X, t(L1,Z,L2)/_), % Ввести элемент X в левое поддерево
 combine(L1, Z, L2, Y, R, NewTree). % Объединить компоненты дерева NewTree

addavl(t(L,Y,R)/Hy, X, NewTree) :-
 gt(X, Y),
 addavl(R, X, t(R1,Z,R2)/_), % Ввести в правое поддерево
 combine(L, Y, R1, Z, R2, NewTree).

% combine(Tree1, A, Tree2, B, Tree3, NewTree):
% Объединить поддерева Tree1, Tree2, Tree3 и узлы A и B в AVL-дерево
combine(T1/H1, A, t(T21,B,T22)/H2, C, T3/H3,
 t(t(T1/H1,A,T21)/Ha, B, t(T22,C,T3/H3)/Hc)/Hb) :-
 H2 > H1, H2 > H3, % Среднее поддерево - самое высокое
 Ha is H1 + 1,
 Hc is H3 + 1,
 Hb is Ha + 1.

combine(T1/H1, A, T2/H2, C, T3/H3,
 t(T1/H1, A, t(T2/H2,C,T3/H3)/Hc)/Ha) :-
 H1 >= H2, H1 >= H3, % Высокое левое поддерево
 max1(H2, H3, Hc),
 max1(H1, Hc, Ha).

combine(T1/H1, A, T2/H2, C, T3/H3,
 t(t(T1/H1,A,T2/H2)/Ha, C, T3/H3)/Hc) :-
 H3 >= H2, H3 >= H1, % Высокое правое поддерево
 max1(H1, H2, Ha),
 max1(Ha, H3, Hc).

max1(U, V, M) :- % M равно 1 + максимальное из двух значение, U и V
 U > V, !, M is U + 1;
 M is V + 1.
```

В процессе работы этой программы учитываются значения высоты деревьев. Но, как было указано выше, возможен более краткий способ представления. В действительности необходимо знать лишь степень нарушения сбалансированности (разницу высот), которая может составлять только -1, 0 или +1. Тем не менее недостатком такого сокращенного способа представления является некоторое усложнение правил соединения компонентов.

## Упражнения

### 10.3. Определите отношение

```
avl(Tree)
```

для проверки того, является ли AVL-деревом бинарное дерево Tree; это означает, что все поддерева одного уровня могут отличаться друг от друга по высоте не больше чем на 1. Допустим, что бинарные деревья представлены с помощью термов в форме `t( Left, Root, Right)` или `nil`.

**10.4.** Проведите трассировку выполнения алгоритма вставки в AVL-дерево, начиная с пустого дерева и последовательно вставляя элементы 5, 8, 9, 3, 1, 6, 7. Как изменяется корневой элемент во время этого процесса?

## Резюме

- *Двоично-троичные и AVL-деревья*, рассматриваемые в данной главе, относятся к разновидностям сбалансированных деревьев.
- *Сбалансированные или приближенно сбалансированные деревья* гарантируют эффективное выполнение трех основных операций с деревьями: поиск, добавление и удаление элемента. Все эти операции могут быть выполнены за время, пропорциональное  $\log n$ , где  $n$  — количество узлов в дереве.

## Дополнительные источники информации

Двоично-троичные деревья были подробно описаны, например, в [2] и [3]. При этом в [3] показана также реализация таких деревьев на языке Pascal. В [172] приведена программа Pascal для работы с AVL-деревьями. Двоично-троичные деревья представляют собой частный случай более общих В-деревьев. Эти и некоторые другие варианты структур данных, относящихся к двоично-троичным и AVL-деревьям, рассматриваются, кроме прочих источников, в [61] вместе с описанием различных результатов исследования поведения этих структур. Программа вставки в AVL-дерево, в которой используется только информация о смещении уровней в дереве (т.е. о разности высот поддеревьев,  $-1$ ,  $0$  или  $+1$ ), а не полная высота, опубликована в [159].



## Часть II

# Применение языка Prolog в области искусственного интеллекта

---

### *В этой части...*

|                                                                                |     |
|--------------------------------------------------------------------------------|-----|
| Глава 11. Основные стратегии решения проблем                                   | 228 |
| Глава 12. Эвристический поиск по заданному критерию                            | 247 |
| Глава 13. Декомпозиция задач и графы AND/OR                                    | 277 |
| Глава 14. Логическое программирование в ограничениях                           | 301 |
| Глава 15. Представление знаний и экспертные системы                            | 326 |
| Глава 16. Командный интерпретатор экспертной системы                           | 357 |
| Глава 17. Планирование                                                         | 383 |
| Глава 18. Машинное обучение                                                    | 408 |
| Глава 19. Индуктивное логическое программирование                              | 446 |
| Глава 20. Качественные рассуждения                                             | 478 |
| Глава 21. Обработка лингвистической информации с помощью грамматических правил | 510 |
| Глава 22. Ведение игры                                                         | 532 |
| Глава 23. Метапрограммирование                                                 | 559 |

# Основные стратегии решения проблем

*В этой главе...*

|                                                |     |
|------------------------------------------------|-----|
| 11.1. Вводные понятия и примеры                | 228 |
| 11.2. Поиск в глубину и итеративное углубление | 232 |
| 11.3. Поиск в ширину                           | 238 |
| 11.4. Анализ основных методов поиска           | 242 |

Основной темой данной главы является использование для представления задач общей схемы, называемой *пространством состояний*. Пространство состояний — это граф, в котором узлы соответствуют проблемным ситуациям, а решение заданной проблемы сводится к поиску пути в этом графе. В этой главе рассматриваются примеры формулировок проблем с использованием подхода на основе пространства состояний и обсуждаются общие методы решения проблем, представленных в такой форме. Решение проблемы сводится к поиску в графе и изучению альтернатив. Основными стратегиями изучения альтернатив, представленными в этой главе, являются поиск в глубину, поиск в ширину и итеративное углубление.

## 11.1. Вводные понятия и примеры

Рассмотрим пример, приведенный на рис. 11.1. Задача состоит в том, что необходимо найти план переупорядочения кубиков, поставленных друг на друга, как показано на этом рисунке. Разрешается передвигать одновременно только один кубик. Кубик можно передвигать, только если на нем не стоит другой кубик. Такой кубик можно поставить на стол или на другой кубик (кубики, поставленные друг на друга, образуют столбик). Чтобы определить требуемый план, нужно найти последовательность действий, позволяющих осуществить указанную перестановку кубиков.

Эту задачу можно рассматривать как задачу изучения возможных вариантов. В первоначальной проблемной ситуации имеется только один вариант: поставить кубик С на стол. А после того как кубик С поставлен на стол, появляются три следующих варианта:

- поставить А на стол;
- поставить А на С;
- поставить С на А.

Безусловно, что в процессе поиска альтернативных решений не следует уделять большого внимания предыдущему шагу, в котором кубик С был поставлен на стол, поскольку в данной ситуации он представлял собой единственно возможное действие.



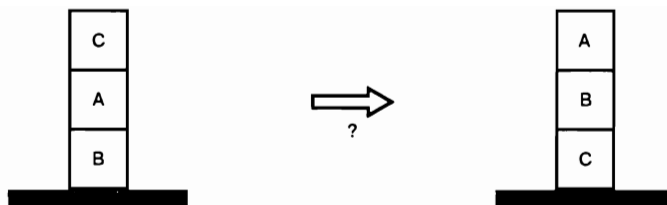


Рис. 11.1. Задача перестановки кубиков

Как показывает данный пример, при анализе подобной проблемы приходится сталкиваться с двумя основными понятиями.

1. Проблемные ситуации.
2. Допустимые шаги, или действия, которые преобразуют одни проблемные ситуации в другие.

Проблемные ситуации и допустимые действия образуют ориентированный граф, называемый *пространством состояний*. Пространство состояний для задачи, рассматриваемой в данном примере, показано на рис. 11.2. Узлы этого графа соответствуют проблемным ситуациям, а дуги — допустимым переходам между состояниями. Задача поиска плана решения эквивалентна поиску пути между заданной начальной ситуацией (*начальным узлом*) и некоторой указанной конечной ситуацией, называемой также *целевым узлом*.

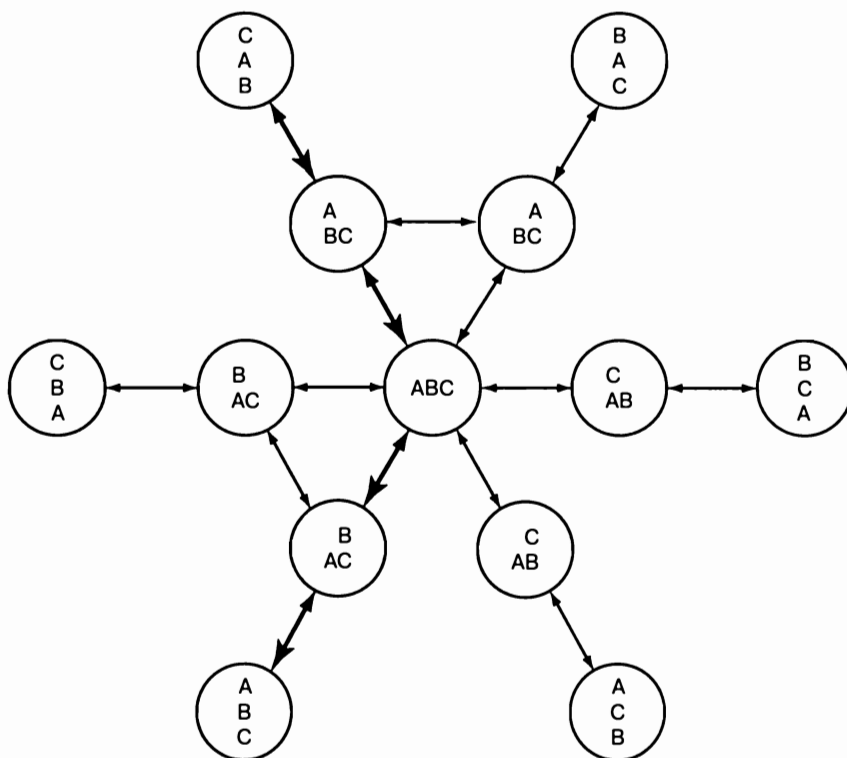


Рис. 11.2. Представление задачи перестановки кубиков в виде пространства состояний; полужирными стрелками обозначен путь решения задачи, представленной на рис. 11.1

На рис. 11.3 приведен еще один пример задачи. Здесь рассматриваются головоломка “игра в восемь” и ее представление в виде проблемы поиска пути. Головоломка состоит из восьми скользящих фишек, пронумерованных цифрами от 1 до 8 и размещенных в коробке с размерами 3×3, с девятью ячейками. Одна из ячеек всегда пуста, и в эту пустую ячейку может быть передвинута по горизонтали или по вертикали любая соседняя фишка. Это правило можно выразить иначе: пустую ячейку можно перемещать по коробке, меняя ее местами с любой из соседних фишек. Конечной ситуацией является некоторое особое расположение фишек, как показано, например, на рис. 11.3.

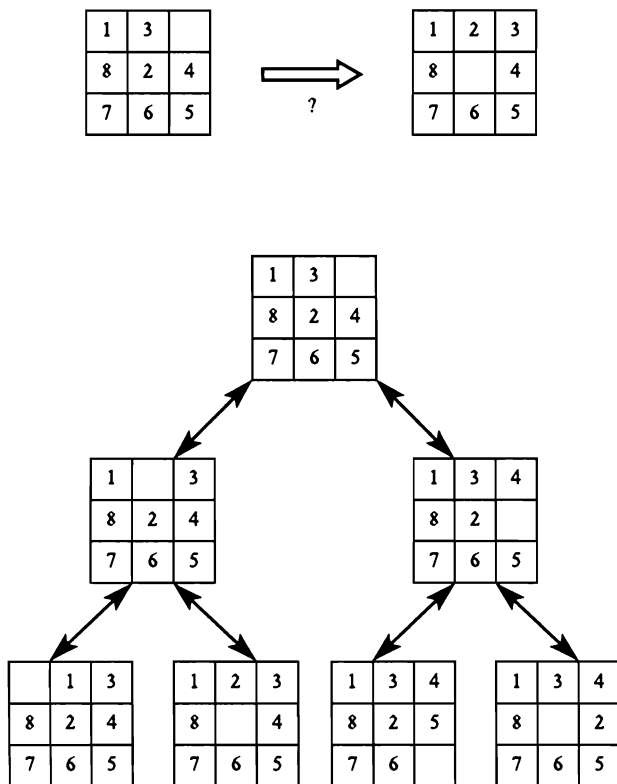


Рис. 11.3. Головоломка “игра в восемь” и соответствующее ей представление в виде пространства состояний

Подытожим понятия, представленные в этих примерах. Пространство состояний указанной задачи определяет “правила игры”, согласно которым узлы в пространстве состояний соответствуют ситуациям, а дуги — допустимым шагам, или действиям, или этапам решения. Любая конкретная задача определяется следующими составляющими.

- Пространство состояний.
- Начальный узел.
- Целевое состояние (состояние, которое должно быть достигнуто); *целевыми узлами* называются такие узлы, которые соответствуют этому состоянию.

Допустимым шагам (или действиям) могут быть поставлены в соответствие стоимости. Например, стоимости, назначенные действиям по перестановке кубиков в за-

даче по их переупорядочению, могут указывать, что одни кубики сложнее переставлять, чем другие. В задаче с коммивояжером шаги соответствуют прямым поездкам из города в город. Естественно, что стоимости этих шагов представляют расстояния между городами.

В тех случаях, если шагам назначаются стоимости, обычно требуется осуществить *оптимизацию* — найти решение с минимальной стоимостью. *Стоимостью решения* является сумма стоимостей всех дуг на пути решения. Но даже если стоимость дуг не указана, может рассматриваться задача оптимизации, в которой необходимо найти кратчайшие решения.

Прежде чем перейти к рассмотрению некоторых программ, в которых реализованы классические алгоритмы поиска в пространстве состояний, рассмотрим, как пространство состояний может быть представлено в программе Prolog.

Пространство состояний представляется отношением

$s(X, Y)$

которое принимает истинное значение, если в данном пространстве состояний имеется допустимый переход из узла  $X$  в узел  $Y$ . В таком случае узел  $Y$  принято называть *преемником* узла  $X$ . Если с допустимыми шагами связаны стоимости, то вводится третий параметр — стоимость шага:

$s(X, Y, \text{Cost})$

Это отношение может быть представлено в программе явно, в виде множества фактов. Но такой вариант при наличии типичного пространства состояний с какой-либо значительной сложностью может оказаться практически неприемлемым или невозможным. Поэтому, как правило, отношение определения преемника,  $s$ , задается неявно путем указания правил вычисления узлов, которые являются преемниками указанного узла.

Обычно возникает еще один важный вопрос — как представлять проблемные ситуации, иными словами, сами узлы. Это представление должно быть компактным, но должно также обеспечивать эффективное выполнение необходимых операций, в частности, вычисление отношения, определяющего преемника, и, возможно, соответствующие стоимости.

В качестве примера рассмотрим задачу перестановки кубиков, показанную на рис. 11.1. Проанализируем более общий случай, при котором разрешается применять любое количество кубиков, расположенных в виде одного или нескольких столбиков. Количество столбиков может быть ограничено некоторым заданным максимумом для того, чтобы задача стала более интересной. К тому же это ограничение может оказаться вполне реальным, поскольку робот, который манипулирует кубиками, может иметь в своем распоряжении только ограниченное рабочее пространство.

Проблемную ситуацию можно представить как список столбиков. Каждый столбик, в свою очередь, представляется в виде списка кубиков в этом столбике, упорядоченного таким образом, чтобы верхний кубик в столбике представлял собой голову списка. Пустые столбики представлены пустыми списками. Поэтому первоначальную ситуацию задачи (см. рис. 11.1) можно определить следующим образом:

$[[c, a, b], [], []]$

Целевой ситуацией является любое расположение кубиков, при котором в одном из столбиков находятся все кубики, расположенные по порядку. Возможны три таких ситуации:

$[[a, b, c], [], []]$

$[], [a, b, c], []]$

$[], [], [a, b, c]$

Отношение определения преемника можно запрограммировать в соответствии со следующим правилом: ситуация 2 является преемником ситуации 1, если в ситуации 1 имеются два столбика,  $\text{Stack1}$  и  $\text{Stack2}$ , и верхний кубик можно перенести со столбика  $\text{Stack1}$  на столбик  $\text{Stack2}$ . Поскольку все ситуации представляются как

списки кубиков в столбиках, такую формулировку задачи можно перевести на язык Prolog следующим образом:

```
s(Stacks, [Stack1, [Top1 | Stack2] | OtherStacks]) :- % Перенести верхний
 % кубик Top1 на столбик Stack2
 del([Top1 | Stack1], Stacks, Stacks1), % Найти первый столбик
 del(Stack2, Stacks1, OtherStacks). % Найти второй столбик

del(X, [X | L], L).
del(X, [Y | L], [Y | L1]) :-
 del(X, L, L1).
```

Целевое состояние для данного примера задачи является следующим:

```
goal(Situation) :-
 member([a,b,c], Situation).
```

Алгоритмы поиска реализуются в программе в виде отношения

```
solve(Start, Solution)
```

где Start — начальный узел в пространстве состояний, а Solution — путь от узла Start до любого целевого узла. Для рассматриваемой задачи перестановки кубиков соответствующий вызов может иметь вид

```
?- solve([[c,a,b], [],[]], Solution).
```

В результате успешного поиска переменная Solution конкретизируется списком перестановок кубиков. Этот список представляет собой план преобразования начального состояния в такое состояние, что все три кубика находятся в одном столбике и расположены как [a,b,c].

## 11.2. Поиск в глубину и итеративное углубление

При использовании формулировки задачи в виде пространства состояний может быть предусмотрено много подходов к поиску пути решения. Двумя основными стратегиями поиска являются поиск в глубину и поиск в ширину. В этом разделе рассматриваются стратегия поиска в глубину и ее вариант, называемый *итеративным углублением*.

Начнем разработку данного алгоритма и его вариантов с анализа описанной ниже простой идеи.

Чтобы найти путь решения Sol от заданного узла N до некоторого целевого узла, необходимо реализовать в программе следующие операции:

- если N — целевой узел, то Sol = [N];
- иначе, если существует узел-преемник N1 узла N, такой, что имеется путь Sol1 от узла N1 до целевого узла, то Sol = [ N | Sol1].

Эта формулировка может быть переведена на язык Prolog следующим образом:

```
solve(N, [N]) :-
 goal(N).

solve(N, [N | Sol1]) :-
 s(N, N1),
 solve(N1, Sol1).
```

По сути, эта программа представляет собой реализацию стратегии поиска в глубину. Ее называют *поиском в глубину* с учетом того, в каком порядке рассматриваются варианты в пространстве состояний. Каждый раз, когда программе поиска в глубину предоставляется возможность продолжить поиск путем перехода к одному из нескольких узлов, она всегда предпочитает выбрать из них самый глубокий. Са-

мым глубоким узлом является тот, который расположен дальше всех от начального узла. На рис. 11.4 показано, в каком порядке посещаются узлы. Этот порядок соответствует трассировке выполнения программы Prolog при поиске ответа на следующий вопрос:

```
?- solve(a, Sol).
```

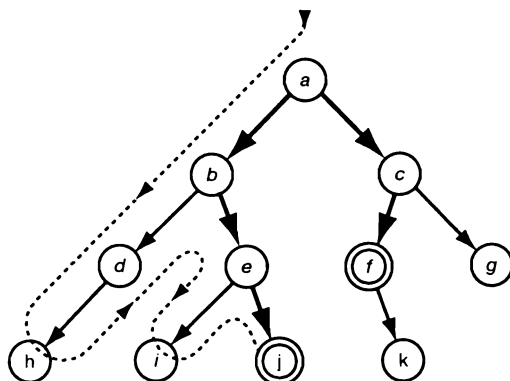


Рис. 11.4. Простое пространство состояний: *a* — начальный узел, *f* и *j* — целевые узлы. Порядок, в котором программа, реализующая стратегию поиска в глубину, посещает узлы в этом пространстве состояний, будет следующим: *a, b, d, h, e, i, j*. Найденным решением является  $\{a, b, e, j\}$ . При переборе с возвратами обнаруживается еще одно решение —  $\{a, c, f\}$

Поиск в глубину в наибольшей степени приемлем для рекурсивного стиля программирования на языке Prolog. Причина этого состоит в том, что сама система Prolog при выполнении цели проверяет варианты по принципу поиска в глубину.

Стратегия поиска в глубину является простой и удобной для программной реализации и может успешно применяться в определенных случаях. Программы решения задачи с восемью ферзями (см. главу 4), по сути, были примерами поиска в глубину. Формулировка задачи с восемью ферзями на основе пространства состояний, которая может использоваться в приведенной выше процедуре `solve`, состоит в следующем.

- Узлами являются позиции на доске, в которых на подряд идущих вертикальных рядах доски помещено от нуля или больше ферзей.
- Узел-преемник формируется путем помещения еще одного ферзя на следующий вертикальный ряд доски таким образом, чтобы он не нападал ни на одного из имеющихся ферзей.
- Начальным узлом является пустая доска, представленная пустым списком.
- Целевым узлом является любая позиция с восемью ферзями (правило выбора преемника гарантирует, что ни один ферзь не нападает на другого).

Если позиция на доске представлена как список координат *Y* ферзей, то эту формулировку можно представить в виде программы следующим образом:

```
s(Queens, [Queen | Queens]) :-
 member(Queen, [1,2,3,4,5,6,7,8]), % Поместить ферзя Queen на любой
 % горизонтальный ряд

 noattack(Queen, Queens).

goal([_,_,_,_,_,_,_,_]). % Позиция с 8 ферзями
```

Отношение `noattack` требует, чтобы ферзь `Queen` не нападал ни на одного из ферзей в списке `Queens`; это отношение можно легко представить в программе, как показано в главе 4. Вопрос

```
?- solve([], Solution).
```

будет вырабатывать список позиций на доске со все возрастающим количеством ферзей. В конечном итоге в этом списке будет представлена безопасная конфигурация с восемью ферзями. Такой вопрос позволяет также найти альтернативные решения с помощью перебора с возвратами.

Стратегия поиска в глубину часто успешно реализуется, как и в этом примере, но имеется также много ситуаций, при которых рассматриваемая простая процедура `solve` может столкнуться с затруднениями. Действительно ли это произойдет или нет, зависит от пространства состояний. Для того чтобы нарушить работу процедуры `solve` при решении задачи, представленной на рис. 11.4, достаточно внести небольшое изменение в формулировку этой задачи: добавить дугу от `h` до `d`, создав тем самым цикл (рис. 11.5). В таком случае поиск будет осуществляться следующим образом: программа начнет работу с узла `a` и спустится к `h`, следуя по крайней левой ветви графа. В этот момент, в отличие от рис. 11.4, узел `h` имеет преемника, `d`. Поэтому в процессе выполнения программы происходит не возврат из узла `h`, а переход к узлу `d`. Затем будет найден преемник `d`, узел `h` и т.д., в результате чего программа начнет переходить по циклу от `d` к `h` и наоборот.

Очевидным усовершенствованием программы поиска в глубину является введение механизма распознавания цикла. В соответствии с ним любой узел, который уже встречался в пути от начального узла к текущему узлу, больше не должен рассматриваться. Это условие можно сформулировать с помощью следующего отношения:

```
depthfirst(Path, Node, Solution)
```

Как показано на рис. 11.6, `Node` — это состояние, из которого необходимо найти путь к целевому состоянию, `Path` — это путь (список узлов) между начальным узлом и узлом `Node`, `Solution` — это путь `Path`, проходящий через `Node` к целевому узлу.

Для упрощения программирования пути в рассматриваемой программе представлены в виде списков с узлами в обратном порядке. Параметр `Path` может использоваться для двух назначений.

1. Для исключения вероятности того, чтобы в программе рассматривались преемники узла `Node`, которые встречались ранее (распознавание циклов).
2. Для формирования пути решения `Solution`.

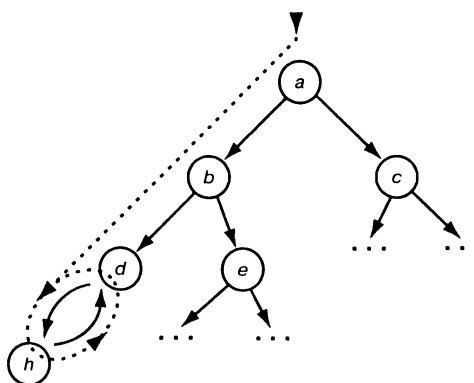


Рис. 11.5. Начиная с узла `a`, процесс поиска в глубину оканчивается возникновением цикла между узлами `d` и `h` следующим образом: `a, b, d, h, d, h, d, ...`

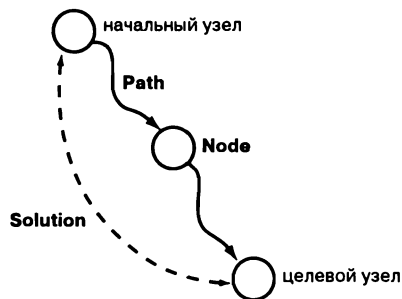


Рис. 11.6. Отношение `depthfirst(Path, Node, Solution)`

Соответствующая программа поиска в глубину приведена в листинге 11.1.

#### Листинг 11.1. Программа поиска в глубину, которая позволяет предотвратить возникновение циклов

```
% solve(Node, Solution):
% Решение Solution представляет собой ациклический путь (узлы в котором
% указаны в обратном порядке) между узлом Node и целью

solve(Node, Solution) :-
 depthfirst([], Node, Solution).

% depthfirst(Path, Node, Solution):
% решение Solution формируется в результате продления пути [Node | Path]
% до целевого узла

depthfirst(Path, Node, [Node | Path]) :-
 goal(Node).

depthfirst(Path, Node, Sol) :-
 s(Node, Node1),
 not member(Node1, Path),
 depthfirst([Node | Path], Node1, Sol). % Предотвращение цикла
```

Благодаря применению механизма обнаружения циклов рассматриваемая процедура поиска в глубину позволяет находить пути решения в пространствах состояний, подобных приведенному на рис. 11.5. Тем не менее есть такие пространства состояний, в которых эта программа вполне может потерпеть неудачу. В частности, многие пространства состояний являются бесконечными. В подобном пространстве состояний программа поиска в глубину может пропустить целевой узел, проходя по бесконечной ветви графа. Затем программа неопределенно долго будет исследовать эту бесконечную часть пространства, не приближаясь к цели. На первый взгляд может показаться, что причиной возникновения ловушки такого рода может стать пространство состояний задачи с восемью ферзями, которое определено в этом разделе. Но по стечению обстоятельств данное пространство является конечным, поскольку максимальное количество вариантов выбора координат  $Y$  клеток, на которых могут быть безопасно расставлены восемь ферзей, является ограниченным.

Для предотвращения бессмысленного прохождение по бесконечным (нециклическим) ветвям можно ввести еще одно усовершенствование в основную процедуру поиска в глубину: ограничить глубину поиска. В таком случае процедура поиска в глубину будет иметь следующие параметры:

```
depthfirst2(Node, Solution, Maxdepth)
```

Поиск не должен проводиться в глубину, превышающую Maxdepth. Это ограничение можно запрограммировать путем уменьшения предельного значения глубины при каждом рекурсивном вызове и контроля над тем, чтобы этот предел оставался положительным. Результирующая программа приведена в листинге 11.2.

### Листинг 11.2. Программа поиска в глубину с ограничением глубины поиска

```
% depthfirst2(Node, Solution, Maxdepth):
% Решение Solution представляет собой путь от узла Node до целевого узла,
% длина которого не превышает Maxdepth

depthfirst2(Node, [Node], _) :-
 goal(Node).

depthfirst2(Node, [Node | Sol], Maxdepth) :-
 Maxdepth > 0,
 s(Node, Node1),
 Max1 is Maxdepth - 1,
 depthfirst2(Node1, Sol, Max1).
```

При использовании программы поиска в глубину (см. листинг 11.2) возникает определенное затруднение, связанное с тем, что подходящий предел необходимо устанавливать заранее. Если этот предел будет установлен слишком низким (т.е. меньшим по сравнению с длиной любого пути решения), поиск завершится неудачей. А если предел будет установлен слишком высоким, поиск потребует лишних затрат времени. Для преодоления этой сложности можно выполнять поиск в глубину итеративно, постепенно изменяя предел глубины, начиная с очень низкого предела глубины и постепенно увеличивая этот предел до тех пор, пока не будет найдено решение. Этот метод называется *итеративным углублением*. Его можно реализовать путем модификации программы, приведенной в листинге 11.2, следующим образом: предусмотреть вызов процедуры depthfirst2 из другой процедуры, которая при каждом рекурсивном вызове будет увеличивать предел на 1.

Тем не менее существует более изящная реализация, которая основана на использовании следующей процедуры:

```
path(Node1, Node2, Path)
```

где Path — ациклический путь в обратном порядке между узлами Node1 и Node2 в пространстве состояний. Предположим, что этот путь представлен как список узлов в обратном порядке. В таком случае процедуру path можно записать следующим образом:

```
path(Node, Node, [Node]). % Путь с одним узлом

path(FirstNode, LastNode, [LastNode | Path]) :-
 path(FirstNode, OneButLast, Path), % Путь, который включает все узлы,
 % кроме предпоследнего
 s(OneButLast, LastNode), % Последний этап пути
 not member(LastNode, Path). % Отсутствие цикла
```

Попытаемся найти некоторые пути, начинающиеся с узла a в пространстве состояний, приведенном на рис. 11.4, как показано ниже.

```
?- path(a, Last, Path).
Last = a
Path = [a];
Last = b
Path = [b,a];
Last = c
Path = [c,a];
Last = d
Path = [d,b,a];
...
```



Процедура `path`, в которой задан некоторый начальный узел, находит все возможные ациклические пути с постоянно возрастающей длиной. Именно это и требуется в подходе с использованием итеративного углубления: находить пути, имеющие все большую и большую длину, до тех пор, пока не будет найден путь, который оканчивается в целевом узле. Применение этой процедуры позволяет сразу же создать следующую программу поиска в глубину с итеративным углублением:

```
depth_first_iterative_deepening(Node, Solution) :-
 path(Node, GoalNode, Solution),
 goal(GoalNode).
```

Такая процедура действительно является очень удобной с точки зрения практики, при условии, что комбинаторная сложность рассматриваемой задачи не требует использования эвристик, характерных для этой задачи. Данная процедура проста, и, даже притом что она не выполняет каких-либо достаточно “разумных” действий, ее использование не требует больших ресурсов времени или пространства. По сравнению с некоторыми другими стратегиями поиска, такими как поиск в ширину, основным преимуществом итеративного углубления является то, что для реализации этой стратегии необходим относительно небольшой объем памяти. На любом этапе выполнения программы требования к памяти, по сути, ограничиваются необходимостью хранить данные об одном пути, между начальным узлом поиска и текущим узлом. Пути формируются, проверяются и отбрасываются, в отличие от некоторых других процедур поиска (таких как поиск в ширину), при которых во время поиска одновременно хранится информация о многих возможных путях. Недостаток стратегии итеративного углубления является продолжением ее основного достоинства: при увеличении предела глубины во время каждой итерации ранее вычисленные пути необходимо перевычислять и продлевать до нового предела. Но при решении типичных задач поиска такое повторное вычисление не слишком сильно влияет на общую продолжительность вычислений. Как правило, основной объем вычислений выполняется на самых низких уровнях поиска, поэтому повторные вычисления на верхних уровнях занимают относительно небольшую часть в общем объеме времени.

## Упражнения

### 11.1. Напишите процедуру поиска в глубину (с распознаванием циклов)

```
depthfirst1(CandidatePath, Solution)
```

для поиска пути решения `Solution` как продолжения пути `CandidatePath`. Допустим, что оба пути должны быть представлены как списки узлов в обратном порядке, чтобы целевой узел был головой списка `Solution`.

### 11.2. Напишите процедуру поиска в глубину, в которой объединяются механизмы распознавания циклов и ограничения глубины, взятые из процедур, приведенных в листингах 11.1 и 11.2.

### 11.3. Процедура `depth_first_iterative_deepening/2`, приведенная в данном разделе, может попасть в бесконечный цикл, если в рассматриваемом пространстве состояний отсутствует путь к решению. Она продолжает поиск все более длинных путей к решению, даже если становится очевидно, что не существуют путей длиннее чем те, по которым уже проводился поиск. Та же самая проблема может возникнуть, если пользователь потребует предоставить ему альтернативные решения после того, как все решения уже будут найдены. Напишите программу поиска с итеративным углублением, которая ищет пути с длиной $i+1$ , только если был найден по меньшей мере один путь с длиной $i$ .

### 11.4. Проведите эксперименты с программами поиска в глубину, приведенными в этом разделе, при решении задачи планирования в мире кубиков, показанной на рис. 11.1.

### 11.5. Напишите процедуру

```
show(Situation)
```

для отображения проблемного состояния Situation в мире кубиков. Предположим, что Situation — это список столбиков, а каждый столбик, в свою очередь, представляет собой список кубиков. Цель

```
show([[a], [e,d], [c,b]])
```

должна отображать соответствующую ситуацию, например, как показано ниже.

```

 e c
 d b
a
=====

```

## 11.3. Поиск в ширину

В отличие от стратегии поиска в глубину, стратегия поиска в ширину предусматривает посещение в первую очередь тех узлов, которые являются ближайшими к начальному узлу. Это приводит к осуществлению процесса поиска, который, как правило, развивается в большей степени в ширину, чем в глубину (рис. 11.7).

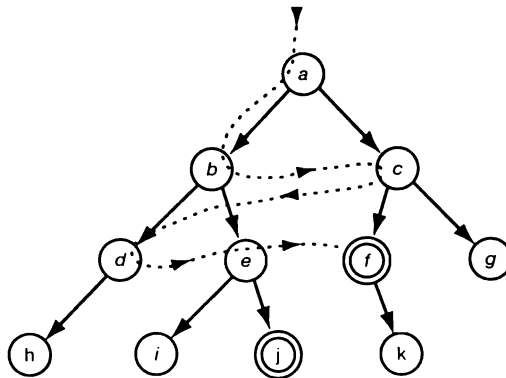


Рис. 11.7. Простое пространство состояний: *a* — начальный узел, *f* и *j* — целевые узлы. Порядок, в котором программа, реализующая стратегию поиска в ширину, посещает узлы в этом пространстве состояний, является следующим: *a*, *b*, *c*, *d*, *e*, *f*. Более короткое решение,  $[a, c, f]$ , обнаруживается прежде, чем длинное,  $[a, b, e, j]$

Составление программы поиска в ширину сложнее по сравнению с поиском в глубину. Причина такого усложнения состоит в том, что необходимо сопровождать целое множество возможных альтернативных узлов, а не рассматривать только один узел, как при поиске в глубину. Такое множество возможных узлов представляет собой целую грань дерева поиска, которая постепенно сдвигается вниз. Но даже такое множество узлов является недостаточным, если в процессе поиска необходимо также выделить путь к решению. Поэтому вместо сопровождения множества возможных узлов приходится сопровождать множество возможных путей. Таким образом, отношение

```
breadthfirst(Paths, Solution)
```

принимает истинное значение, если некоторый путь из множества возможных путей Paths может быть продлен до целевого узла. Таким продленным путем является путь Solution.

Для представления множества возможных путей будет использоваться следующий способ: это множество будет представлено как список путей, а каждый путь — как список узлов в обратном порядке. Это означает, что голова в списке узлов представляет собой узел, сформированный в самую последнюю очередь, а последним элементом в списке является начальный узел поиска. При инициализации поиска исходное множество, состоящее из одного возможного элемента, задается следующим образом:

```
[[StartNode]]
```

Общая схема поиска в ширину приведена ниже.

Для осуществления поиска в ширину по заданному множеству возможных путей необходимо выполнить следующие действия:

- если голова первого пути представляет собой целевой узел, то считать этот путь решением задачи;
- в противном случае удалить первый путь из множества возможных путей и сформировать множество, состоящее из всех возможных одношаговых продолжений этого пути, добавить это множество продолжений к концу множества возможных путей и выполнить поиск в ширину на этом модифицированном множестве.

Применительно к примеру задачи, приведенному на рис. 11.7, этот процесс разбивается, как описано ниже.

1. Начать с первоначального множества возможных путей:

```
[[a]]
```

2. Сформировать продолжения пути [a]:

```
[[b,a], [c,a]]
```

(Обратите внимание, что узлы во всех путях представлены в обратном порядке.)

3. Удалить из множества первый возможный путь, [b,a], и сформировать продолжения этого пути:

```
[[d,b,a], [e,b,a]]
```

Добавить список продолжений к концу множества возможных путей:

```
[[c,a], [d,b,a], [e,b,a]]
```

4. Удалить путь [c,a] и добавить его продолжения к концу множества возможных путей, что приводит к получению следующего множества:

```
[[d,b,a], [e,b,a], [f,c,a], [g,c,a]]
```

На последующих этапах продлеваются пути [d,b,a] и [e,b,a] и модифицированное множество возможных путей приобретает вид

```
[[f,c,a], [g,c,a], [h,d,b,a], [i,e,b,a], [j,e,b,a]]
```

Теперь процесс поиска находит путь [f,c,a], который содержит целевой узел, f. Поэтому данный путь возвращается как решение.

Программа, которая осуществляет указанный процесс, приведена в листинге 11.3. В этой программе все одношаговые продолжения вырабатываются с использованием встроенной процедуры bagof. Выполняется также проверка для предотвращения образования циклических путей. Обратите внимание, что в том случае, если продолжение пути невозможно, процедура bagof завершается неудачей и поэтому предусмотрен альтернативный вызов процедуры breadthfirst. Здесь member и conc, соответственно, представляют собой отношения проверки принадлежности к списку и конкатенации списков.

### Листинг 11.3. Реализация алгоритма поиска в ширину

```
% solve(Start, Solution):
% Решение Solution представляет собой путь (узлы в котором представлены
% в обратном порядке) от узла Start до целевого узла

solve(Start, Solution) :-
 breadthfirst([[Start]], Solution).

% breadthfirst([Path1, Path2, ...], Solution):
% Решение Solution представляет собой результат продления одного из путей
% до целевого узла

breadthfirst([[Node | Path] | _], [Node | Path]) :-
 goal(Node).

breadthfirst([Path | Paths], Solution) :-
 extend(Path, NewPaths),
 conc(Paths, NewPaths, Paths1),
 breadthfirst(Paths1, Solution).

extend([Node | Path], NewPaths) :-
 bagof([NewNode, Node | Path],
 (s(Node, NewNode), not member(NewNode, [Node | Path])),
 NewPaths),
 !.
% Предложение, выполняемое после неудачного завершения вызова предиката bagof,
% который означает, что узел Node не имеет преемников
extend(Path, []).
```

Один из недостатков этой программы обусловлен низкой эффективностью операции `conc`. Этот недостаток можно устранить с помощью способа представления списков в виде разностных пар (см. главу 8). В таком случае множество возможных путей представляется в виде пар списков, `Paths` и `Z`, как показано ниже.

`Paths - Z`

Применив этот способ представления в программе, приведенной в листинге 11.3, ее можно постепенно преобразовать в программу, показанную в листинге 11.4. Оставляем самостоятельное выполнение этого преобразования в качестве упражнения для читателя.

**Листинг 11.4. Более эффективная программа поиска в ширину по сравнению с приведенной в листинге 11.3; это усовершенствование основано на использовании способа представления списка возможных путей в виде разностных пар. Процедура `extend` приведена в листинге 11.3**

```
% solve(Start, Solution):
% Решение Solution представляет собой путь (узлы в котором заданы
% в обратном порядке) от узла Start до целевого узла

solve(Start, Solution) :-
 breadthfirst([[Start] | Z] - Z, Solution).

breadthfirst([[Node | Path] | _] - _, [Node | Path]) :-
 goal(Node).

breadthfirst([Path | Paths] - Z, Solution) :-
 extend(Path, NewPaths),
 conc(NewPaths, Z1, Z),
 Paths \== Z1,
 breadthfirst(Paths - Z1, Solution).
% Добавить список NewPaths к концу списка
% Множество возможных путей - не пустое
```

- 11.6.** Предположим, что пространство состояний представляет собой дерево с единообразным ветвлением  $b$ , а путь к решению имеет длину  $d$ . Для частного случая,  $b = 2$  и  $d = 3$ , определите, сколько узлов формируется в наихудшем случае при поиске в ширину и при итеративном углублении (учитывая также повторно формируемые узлы). Обозначьте как  $N(b, d)$  количество узлов, формируемых при итеративном углублении в общем случае. Найдите рекурсивную формулу, позволяющую получить значение  $N(b, d)$  с помощью значения  $N(b, d - 1)$ .
- 11.7.** Перепишите программу поиска в ширину, приведенную в листинге 11.3, используя способ представления списка возможных путей в виде разностной пары, и покажите, что в результате может быть получена программа, показанная в листинге 11.4. Определите, в чем состоит назначение следующей цели, применяемой в листинге 11.4:

```
Paths \== Z1
```

Определите, что происходит, если эта цель в программе исключена; используйте пространство состояний, показанное на рис. 11.7. Подсказка: различие обнаруживается только при попытке найти другие решения после того, как их уже не осталось.

- 11.8.** Как можно воспользоваться программами поиска, приведенными в данном разделе, для проведения поиска от множества начальных узлов, а не от одного начального узла?
- 11.9.** Как можно применить программы поиска, приведенные в этой главе, для поиска в обратном направлении, при котором поиск начинается с целевого узла и продолжается в направлении начального узла (или одного из начальных узлов, при наличии нескольких начальных узлов)? Подсказка: переопределите отношение  $s$ . В каких ситуациях обратный поиск является более выгодным по сравнению с прямым?
- 11.10.** Иногда имеет смысл проводить двунаправленный поиск; т.е. двигаться с обоих концов — от начального и от целевого узлов. Поиск оканчивается, когда оба пути соединяются. Определите пространство поиска (отношение  $s$ ) и целевое отношение для заданного графа таким образом, чтобы рассматриваемые процедуры поиска, по сути, выполняли двунаправленный поиск.
- 11.11.** В трех процедурах поиска, `find1`, `find2` и `find3`, которые определены ниже, используются разные стратегии поиска. Определите эти стратегии.

```
find1(Node, [Node]) :-
 goal(Node).
```

```
find1(Node, [Node | Path]) :-
 s(Node, Node1),
 find1(Node1, Path).
```

```
find2(Node, Path) :-
 conc(Path, _, _), % Обычная процедура conc/3 для конкатенации списков
 find1(Node, Path).
```

```
find3(Node, Path) :-
 goal(Goal),
 find3(Node, [Goal], Path).
```

```
find3(Node, [Node | Path], [Node | Path]).
```

```
find3(Node, [Node2 | Path2], Path) :-
 s(Node1, Node2),
 find3(Node, [Node1, Node2 | Path2], Path).
```

**11.12.** Изучите следующую программу поиска и опишите используемую в ней стратегию поиска:

```
% search(Start, Path1 - Path2): найти путь от начального узла S до целевого
% Путь решения Solution представлен в виде двух списков - Path1 и Path2
search(S, P1 - P2) :-
 similar_length(P1, P2), % Списки имеют приблизительно равную длину
 goal(G),
 path2(G, P2, N),
 path1(S, P1, N).

path1(N, [N], N).

path1(First, [First | Rest], Last) :-
 s(First, Second),
 path1(Second, Rest, Last).
path2(N, [N], N).

path2(First, [First | Rest], Last) :-
 s(Second, First),
 path2(Second, Rest, Last).

similar_length(List1, List2) :- % Списки приблизительно равной длины
 equal_length(List2, List), % Списки равной длины
 (List1 = List; List1 = [_ | List]).

equal_length([], []).

equal_length([X1 | L1], [X2 | L2]) :-
 equal_length(L1, L2).
```

**11.13.** Проведите эксперименты по использованию различных стратегий поиска для решения задачи планирования в мире кубиков.

**11.14.** Программы поиска в ширину, приведенные в этой главе, предусматривают проверку только повторяющихся узлов, которые появляются в одном и том же возможном пути. В графах один и тот же узел может быть достигнут с помощью разных путей. Такая ситуация данными программами не обнаруживается, поэтому они повторно осуществляют поиск на уровнях ниже тех, на которых находятся подобные узлы. Измените программы, приведенные в листингах 11.3 и 11.4, для предотвращения такой ненужной работы.

## 11.4. Анализ основных методов поиска

В данном разделе проводятся анализ и сравнение основных методов поиска. Вначале рассмотрим их применение для поиска в графах, затем прокомментируем оптимальность вырабатываемых ими решений. Наконец, проанализируем предъявляемые ими требования к ресурсам времени и пространства.

Примеры, которые рассматривались до сих пор, могут создать ложное впечатление, что рассматриваемые программы поиска могут применяться только в пространствах состояний, которые представляют собой деревья, а не графы в любой форме. Но при поиске в графе он, по сути, развертывается в дерево таким образом, что некоторые пути, возможно, копируются в другие части дерева. Такая ситуация иллюстрируется на рис. 11.8. Поэтому рассматриваемые программы работают также с графами, но могут без какой-либо необходимости продублировать часть работы в тех случаях, если некоторый узел может быть достигнут разными путями. Такое развитие событий можно предотвратить, проверяя, не произошло ли повторное появление некоторого узла во всех возможных путях, а не только в том пути, в котором был сформирован этот узел. Безусловно, что такая проверка возможна в рассматриваемых программах поиска в ширину только в тех случаях, если для проверки доступны альтернативные пути.

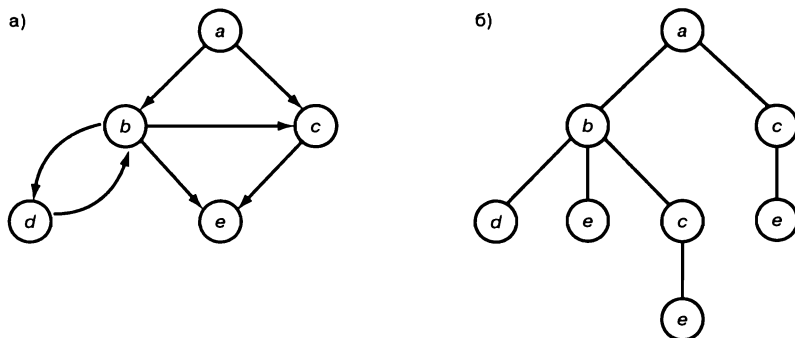


Рис. 11.8. Пример дублирования узлов: а) пространство состояний, в котором  $a$  представляет собой начальный узел; б) дерево всех возможных ациклических путей из узла  $a$ , которое, по сути, формируется программой поиска в ширину, приведенной в листинге 11.3

Рассматриваемые программы поиска в ширину вырабатывают один за другим пути решений, упорядоченные по длине, причем вначале формируются кратчайшие решения. Это важно, если необходимо обеспечить получение оптимальных решений (с точки зрения длины). Стратегия поиска в ширину гарантирует первоочередную выработку кратчайших решений. Это утверждение, безусловно, не относится к стратегии поиска в глубину. Но во время поиска в глубину с итеративным углублением такой поиск выполняется с постепенным увеличивающимися пределами глубины, и поэтому наблюдается тенденция к тому, что в первую очередь обнаруживаются кратчайшие решения. Таким образом, при итеративном углублении происходит своего рода моделирование поиска в ширину.

Но в приведенных в данной главе программах не учитываются какие-либо стоимости, связанные с дугами в пространстве состояний. Если в качестве критерия оптимизации рассматривается минимальная стоимость пути решения (а не его длина), поиск в ширину не может обеспечить выбор оптимального пути решения. Для оптимизации стоимости предназначен поиск по заданному критерию, который рассматривается в главе 12.

Типичной проблемой, связанной с поиском, является комбинаторная сложность. При наличии нетривиальных проблемных областей количество вариантов, подлежащих рассмотрению, становится столь значительным, что наибольшее значение приобретает существенное увеличение затрат ресурсов на исследование данных вариантов. Можно легко проанализировать причины того, почему это происходит. Для упрощения такого анализа предположим, что пространство состояний представляет собой дерево с единообразным ветвлением  $b$ . Это означает, что каждый узел в дереве, за исключением листьев, имеет точно  $b$  преемников. Предположим, что кратчайший путь решения имеет длину  $d$ , и в дереве на глубине  $d$  или меньшей отсутствуют листья. Количество альтернативных путей длины  $d$  от начального узла равно  $b^d$ . При поиске в ширину количество рассматриваемых путей пропорционально  $b^d$ . Это обозначается как  $O(b^d)$ . Итак, количество возможных путей очень быстро возрастает при увеличении их длины, что приводит к так называемому *комбинаторному взрыву*.

Теперь сравним затраты ресурсов на осуществление основных алгоритмов поиска. Потребность в ресурсах времени обычно определяется в зависимости от количества узлов, формируемых алгоритмом поиска, а затраты ресурсов пространства обычно определяются в зависимости от максимального количества узлов, которые должны храниться в памяти во время поиска.

Рассмотрим поиск в ширину в дереве с коэффициентом ветвления  $b$  и кратчайшим путем решения длиной  $d$ . Количество узлов на последовательно углубляющихся

уровнях дерева возрастает экспоненциально с глубиной, поэтому количество узлов, вырабатываемых при поиске в ширину, можно определить следующим образом:

$$1 + b + b^2 + b^3 + \dots$$

Общее количество узлов, вплоть до глубины решения  $d$ , составляет  $O(b^d)$ , поэтому затраты ресурсов времени при поиске в ширину измеряются значением  $O(b^d)$ . Кроме того, при поиске в ширину информация обо всех возможных путях хранится в памяти, поэтому затраты ресурсов пространства также измеряются значением  $O(b^d)$ .

Задача анализа неограниченного поиска в глубину является менее очевидной, поскольку при таком поиске система может полностью пропустить правильный путь решения длиной  $d$  и неограниченно долго продолжать поиск в бесконечном поддереве. Для упрощения анализа рассмотрим поиск в глубину, ограниченный максимальной глубиной  $d_{\max}$ , такой, что  $d \leq d_{\max}$ . Затраты времени при этом измеряются значением  $O(b^{d_{\max}})$ . Но затраты ресурсов пространства составляют всего  $O(d_{\max})$ . При поиске в глубину, по сути, сопровождается лишь путь между начальным и текущим узлами поиска, рассматриваемый в настоящее время. Преимущество поиска в глубину по сравнению с поиском в ширину состоит в том, что он требует намного меньших затрат ресурсов пространства, а его недостатком является то, что он не гарантирует получение оптимального решения.

При итеративном углублении выполняется поиск в глубину  $(d + 1)$  со все возрастающей глубиной:  $0, 1, \dots, d$ . Поэтому затраты ресурсов пространства при этом поиске измеряются значением  $O(d)$ . Посещение начального узла происходит  $(d + 1)$  раз, дочерних узлов начального узла —  $d$  раз и т.д. В худшем случае количество формируемых узлов измеряется следующим выражением:

$$(d+1)*1 + d*b + (d-1)*b^2 + \dots + 1*b^d$$

Это выражение также измеряется значением  $O(b^d)$ . Но фактически затраты на повторное формирование узлов верхних уровней по сравнению с поиском в ширину весьма малы. Можно показать, что отношение между количеством узлов, вырабатываемых при итеративном углублении и при поиске в ширину, составляет приблизительно  $b/(b - 1)$ . При  $b \geq 2$  такие дополнительные расходы на итеративное углубление относительно невелики, если к тому же учесть весьма значительное уменьшение потребности в пространстве по сравнению с поиском в ширину. В этом смысле итеративное углубление сочетает в себе наилучшие свойства поиска в ширину (гарантированное достижение оптимального решения) и поиска в глубину (экономия пространства) и поэтому на практике часто является наиболее предпочтительным среди всех основных методов поиска.

Рассмотрим также двунаправленный поиск (упражнения 11.10–11.12). В тех случаях, если этот метод поиска является применимым (известен целевой узел), он может привести к значительной экономии. Предположим, что имеется граф поиска с единообразным ветвлением  $b$  в обоих направлениях, а двунаправленный поиск реализован как поиск в ширину в обоих направлениях. Допустим, что кратчайший путь решения имеет длину  $d$ , поэтому двунаправленный поиск окончится, когда встретятся оба пути поиска в ширину; это означает, что оба эти пути пройдут приблизительно до середины между начальным и целевым узлами. Таким образом, встреча путей произойдет, когда они пройдут от своих соответствующих узлов на глубину, составляющую примерно  $d/2$ . Поэтому затраты ресурсов при выполнении поиска в каждом из направлений измеряются значением, которое ориентировочно равно  $b^{d/2}$ . При таких благоприятных обстоятельствах двунаправленный поиск позволяет найти путь решения длиной  $d$  с потреблением примерно таких же ресурсов, как при поиске в ширину, но в результате решения более простой задачи с длиной пути решения, равной  $d/2$ . В табл. 11.1 приведены итоговые данные, позволяющие провести сравнение основных методов поиска.



Таблица 11.1. Приближенные оценки затрат ресурсов, связанных с использованием основных методов поиска. Здесь  $b$  — коэффициент ветвления,  $d$  — длина кратчайшего пути решения,  $d_{\max}$  — предел глубины при поиске в глубину,  $d \leq d_{\max}$

| Метод поиска                      | Время          | Пространство | Самое короткое гарантируемое решение |
|-----------------------------------|----------------|--------------|--------------------------------------|
| Поиск в ширину                    | $b^d$          | $b^d$        | да                                   |
| Поиск в глубину                   | $b^{d_{\max}}$ | $d_{\max}$   | нет                                  |
| Итеративное углубление            | $b^d$          | $d$          | да                                   |
| Двунаправленный, если он возможен | $b^{d/2}$      | $b^{d/2}$    | да                                   |

Рассматриваемые в данной главе основные методы поиска не обеспечивают выполнения каких-либо обоснованных действий, позволяющих преодолеть комбинаторный взрыв. В них все возможные пути рассматриваются как в равной степени перспективные и не используется какая-либо информация, относящаяся к конкретной задаче, для управления ведением этого поиска в более перспективном направлении. В данном смысле эти методы не являются достаточно информационно насыщенными. Поэтому описанные в этой главе простые методы поиска не позволяют решать крупномасштабные задачи. При решении подобных задач необходимо использовать информацию, относящуюся к конкретной проблеме, для обеспечения целенаправленного поиска. Такая руководящая информация называется *эвристикой*. Алгоритмы, в которых применяются эвристики, обеспечивают выполнение эвристического поиска. Подобный метод поиска представлен в следующей главе.

## Резюме

- *Пространство состояний* используется в качестве формализованной структуры для представления проблем.
- Пространство состояний — это ориентированный граф, узлы которого соответствуют проблемным ситуациям, а дуги — возможным действиям. Конкретная задача определяется с помощью начального узла и целевого состояния. В таком случае решение задачи соответствует одному из путей в графе. Поэтому решение задачи сводится к поиску пути в графе.
- *Задачи оптимизации* можно моделировать, назначая стоимости дугам в пространстве состояний.
- Тремя основными стратегиями поиска, которые позволяют систематически исследовать пространство состояний, являются *поиск в глубину*, *поиск в ширину* и *итеративное углубление*.
- Разработка программы поиска в глубину может быть осуществлена проще всего, но такая программа восприимчива к образованию циклов. Для предотвращения циклов применяются два простых метода: *ограничение глубины поиска* и *проверка на наличие повторяющихся узлов*.
- Реализация стратегии поиска в ширину сложнее, поскольку требует сопровождения множества возможных путей. Проще всего такое множество можно представить как список списков.
- *Поиск в ширину* всегда позволяет в первую очередь найти кратчайший путь решения, но это утверждение не относится к стратегии поиска в глубину.
- Для поиска в ширину требуется больше пространства, чем для поиска в глубину. На практике пространство часто является наиболее важным и ограниченным ресурсом.
- Метод *поиска в глубину с итеративным углублением* сочетает в себе наилучшие свойства поиска в глубину и в ширину.

- В случае больших пространств состояний возникает опасность *комбинаторного взрыва*. Простые стратегии поиска плохо подходят для преодоления возникающих при этом сложностей. Поэтому в подобных случаях необходимо руководствоваться эвристическими методами.
- В настоящей главе представлены следующие понятия:
  - пространство состояний;
  - начальный узел, целевое условие, путь решения;
  - стратегия поиска;
  - поиск в глубину;
  - поиск в ширину;
  - поиск с итеративным углублением;
  - двунаправленный поиск;
  - эвристический поиск.

## Дополнительные источники информации

Описания основных стратегий поиска можно найти в любой книге по искусственному интеллекту, например [133] и [171]. В [81] показано, как можно использовать логику для реализации соответствующих принципов поиска. В [78] приведен сравнительный анализ преимуществ итеративного углубления.

# Эвристический поиск по заданному критерию

*В этой главе...*

|                                                                                       |     |
|---------------------------------------------------------------------------------------|-----|
| 12.1. Поиск по заданному критерию                                                     | 247 |
| 12.2. Применение поиска по заданному критерию для решения головоломки “игра в восемь” | 256 |
| 12.3. Применение поиска по заданному критерию при планировании                        | 261 |
| 12.4. Методы экономии пространства для поиска по заданному критерию                   | 265 |

Использование методов решения задач, организованных по принципу поиска в графе, обычно приводит к возникновению проблемы комбинаторной сложности из-за быстрого увеличения количества вариантов. Эвристический поиск позволяет эффективно справиться с этой проблемой. Один из способов применения эвристической информации о проблеме состоит в определении числовых эвристических оценок для узлов в пространстве состояний. Такая оценка узла указывает, насколько перспективным является тот или иной узел с точки зрения достижения целевого узла. Идея состоит в том, что поиск должен всегда продолжаться от наиболее перспективного узла в множестве возможных узлов. Программы поиска по заданному критерию, приведенные в этой главе, основаны на данном принципе.

## 12.1. Поиск по заданному критерию

Программа поиска по заданному критерию может быть создана как усовершенствование программы поиска в ширину. Поиск по заданному критерию также начинается от начального узла, и в процессе его происходит сопровождение информации о множестве возможных путей. При поиске в ширину для продолжения всегда выбирается кратчайший возможный путь (т.е. путь через наименее глубокие концевые узлы, достигнутые в ходе поиска), а в программе поиска по заданному критерию применяется усовершенствованный вариант этого принципа: определяется эвристическая оценка для каждого возможного узла и для продолжения пути в соответствии с этой оценкой выбирается наилучший возможный узел.

В дальнейшем предполагается, что функция стоимости определена для дуг в пространстве состояний, а  $c(n, n')$  — это стоимость перемещения от узла  $n$  к его предшественнику  $n'$  в пространстве состояний.

Допустим, что для эвристической оценки применяется функция  $f$ , такая, что для каждого узла  $n$  в пространстве состояний функция  $f(n)$  служит для оценки “сложности достижения”  $n$ . В соответствии с этим наиболее перспективным из всех текущих возможных узлов является тот, для которого значение функции  $f$  становит-

ся минимальным. В настоящей главе применяется сформированная особым образом функция  $f$ , которая дает возможность использовать широко известный алгоритм  $A^*$ . Функция  $f(n)$  будет сформирована таким образом, чтобы она оценивала стоимость наилучшего пути решения от начального узла  $s$  до целевого узла, при том условии, что этот путь пройдет через узел  $n$ . Предположим, что такой путь имеется, и целевым узлом, для которого стоимость этого пути становится минимальной, является узел  $t$ . В таком случае оценку  $f(n)$  можно сформировать как сумму двух термов (рис. 12.1) следующим образом:

$$f(n) = g(n) + h(n)$$

где  $g(n)$  — оценка стоимости оптимального пути от  $s$  до  $n$ , а  $h(n)$  — оценка стоимости оптимального пути от  $n$  до  $t$ .

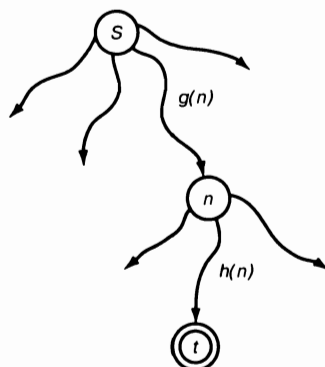


Рис. 12.1. Формирование эвристической оценки  $f(n)$  стоимости пути с наименьшей стоимостью от  $s$  до  $t$  через  $n$ :  
 $f(n) = g(n) + h(n)$

После обнаружения узла  $n$  в процессе поиска возникает следующая ситуация: путь от  $s$  до  $n$  уже должен быть найден и его стоимость может быть вычислена как сумма стоимости дуг в этом пути. Такой путь от  $s$  до  $n$  не обязательно должен быть оптимальным (может существовать лучший путь от  $s$  до  $n$ , еще не обнаруженный в этом процессе поиска), но стоимость найденного пути, выраженная первым термом  $g(n)$ , может служить в качестве оценки минимальной стоимости пути от  $s$  до  $n$ . Задача определения второго терма,  $h(n)$ , сложнее, поскольку “мир” между узлами  $n$  и  $t$  к этому времени еще не был исследован в процессе поиска. Поэтому  $h(n)$ , как правило, представляет собой в полном смысле слова *эвристическую гипотезу*, основанную на имеющейся в распоряжении алгоритма общей информации о данной конкретной задаче. Поскольку значение  $h$  зависит от проблемной области, универсального метода формирования функции  $h$  не существует. Конкретные примеры того, как могут выдвигаться подобные эвристические гипотезы, будут приведены ниже. Но на данный момент предположим, что функция  $h$  задана, и сосредоточимся на изучении программы поиска по заданному критерию.

Работу программы поиска по заданному критерию можно представить себе следующим образом. Процесс поиска состоит из множества конкурирующих подпроцессов, каждый из которых исследует свой собственный вариант; иными словами, исследует собственное поддерево. В поддеревьях имеются свои поддеревья; они исследуются подпроцессами подпроцессов и т.д. Среди всех этих конкурирующих процессов в любой момент времени активным является только один — тот, который имеет дело с вариантом, являющимся в данный момент *наиболее перспективным*; таковым называется вариант с наименьшим значением функции  $f$  (или кратко

*f*-значением). Остальные процессы должны находиться в состоянии ожидания до тех пор, пока текущие *f*-оценки не изменятся таким образом, что более перспективным станет какой-то иной вариант. После этого активность переключается на этот вариант. Можно представить себе, что такой механизм активизации и перехода в неактивное состояние действует следующим образом: процессу, работающему над вариантом, имеющим в настоящее время наивысший приоритет, выделяются некоторые ресурсы и процесс остается активным до тех пор, пока эти ресурсы не будут исчерпаны. В этот период активности процесс продолжает разворачивать свое поддерево и сообщает решение, если был обнаружен целевой узел. Ресурсы для этого этапа выполнения выделяются на основании эвристической оценки ближайшего конкурирующего варианта.

Пример такого поведения показан на рис. 12.2. Предположим, что дана некоторая дорожная карта и задача состоит в том, чтобы найти кратчайший маршрут между начальным городом *s* и целевым городом *t*. В процессе оценки стоимости оставшегося расстояния в маршруте от города *X* до цели используется расстояние по прямой, обозначенное как  $\text{dist}(X, t)$ . Поэтому справедлива следующая формула:

$$f(X) = g(X) + h(X) = g(X) + \text{dist}(X, t)$$

В данном примере можно представить себе, что поиск по заданному критерию состоит из двух процессов, в каждом из которых исследуется один из двух возможных путей: в процессе 1 — путь через узел *a*, в процессе 2 — путь через узел *e*. На начальных этапах процесс 1 является более активным, поскольку *f*-значения вдоль его пути меньше, чем вдоль другого пути. А в тот момент, когда процесс 1 находится в узле *c*, а процесс 2 — все еще в узле *e*, ситуация изменяется следующим образом:

$$f(c) = g(c) + h(c) = 6 + 4 = 10$$

$$f(e) = g(e) + h(e) = 2 + 7 = 9$$

Поэтому  $f(e) < f(c)$ , и теперь процесс 2 переходит к узлу *f*, а процесс 1 ожидает. Но здесь возникает такая ситуация:

$$f(f) = 7 + 4 = 11$$

$$f(c) = 10$$

$$f(c) < f(f)$$

Поэтому процесс 2 останавливается, а процесс 1 получает разрешение продолжить свою работу, но только до пункта *d*, когда обнаруживается, что  $f(d) = 12 > 11$ . Теперь процесс 2, активизированный в этот момент, беспрепятственно достигает цели *t*.

В процессе поиска, организованном таким образом, начиная с начального узла продолжается выработка новых узлов-преемников и путь всегда продлевается в наиболее перспективном направлении в соответствии с *f*-значениями. В течение этого процесса формируется дерево поиска, корнем которого является начальный узел поиска. Поэтому рассматриваемая программа поиска по заданному критерию продолжает разворачивать дерево поиска до тех пор, пока не будет найдено решение. Такое дерево представлено в данной программе с помощью термов в двух описанных ниже формах.

1. Терм  $l(N, F/G)$  представляет отдельный узел дерева (лист); где *N* — узел в пространстве состояний, *G* — значение функции  $g(N)$  (стоимость пути, пройденного от начального узла до *N*), *F* — значение функции  $f(N) = G + h(N)$ .
2. Терм  $t(N, F/G, Subs)$  представляет дерево с непустыми поддеревьями; где *N* — корень дерева, *Subs* — список его поддеревьев, *G* — значение функции  $g(N)$ , *F* — “обновленное” *f*-значение *N* (под этим подразумевается *f*-значение наиболее перспективного преемника *N*); список *Subs* упорядочен в соответствии с возрастающими *f*-значениями поддеревьев.

Например, снова рассмотрим процесс поиска, проиллюстрированный на рис. 12.2. В тот момент, когда был развернут узел *s*, дерево поиска состояло из трех узлов: корня *s* и двух его дочерних узлов, *a* и *e*. В рассматриваемой программе такое дерево поиска представлено следующим термом:

$$t(s, 7/0, [l(a, 7/2), l(e, 9/2)])$$

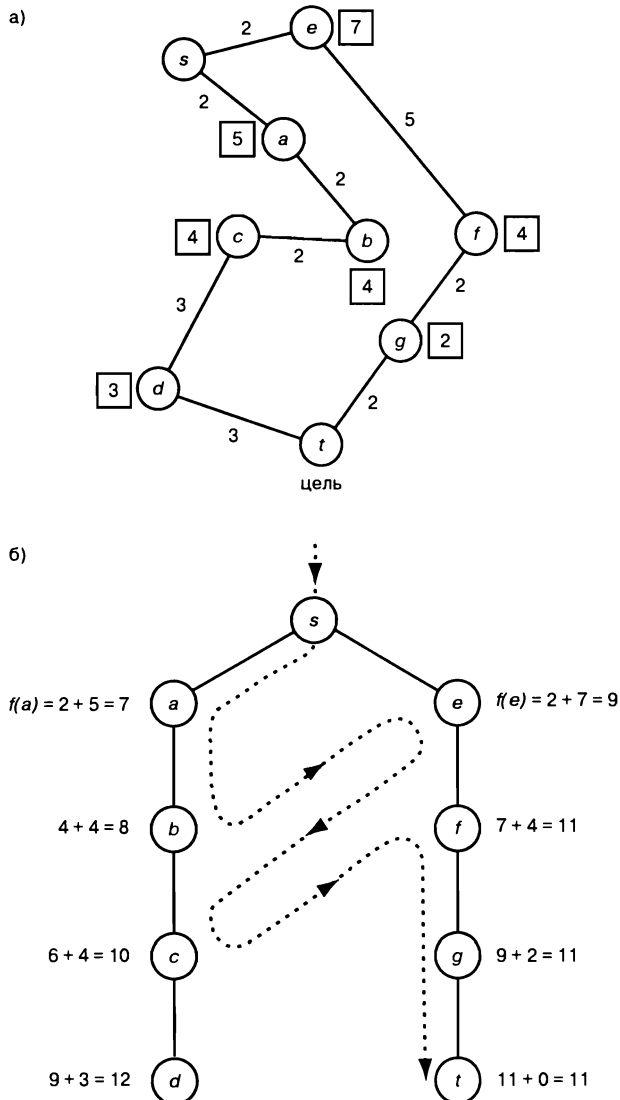


Рис. 12.2. Поиск кратчайшего маршрута от  $s$  до  $t$  по карте: а) карта, на которой указаны расстояния между городами; числа в квадратах указывают расстояние по прямой до  $t$ ; б) порядок, в котором происходит изучение карты при поиске по заданному критерию. При определении эвристических оценок за основу взяты расстояния по прямой. Пунктирная линия показывает, как происходит переключение активности между альтернативными путями. Сплошная линия показывает упорядочение узлов в соответствии с их  $f$ -значениями, иными словами, порядок, в котором узлы развертывались (а не порядок, в котором они формировались)

Здесь  $f$ -значение корня  $s$  равно 7, т.е. соответствует  $f$ -значению наиболее перспективного приемника корня — узла  $a$ . Теперь дерево поиска развертывается путем

развертывания наиболее перспективного поддерева  $a$ . Ближайшим конкурентом узла  $a$  является  $e$ , для которого  $f$ -значение равно 9. Таким образом, узлу  $a$  разрешено развернуться, при условии, что  $f$ -значение  $a$  не будет превышать 9. Поэтому формируются узлы  $b$  и  $c$ . Значение  $f(c) = 10$ , поэтому предел развертывания превышен и варианту  $a$  больше не разрешено расти. В данный момент дерево поиска имеет следующий вид:

```
t(s, 9/0, [l(e,9/2), t(a,10/2, [t(b,10/4, [l(c,10/6)])])])
```

Следует отметить, что теперь  $f$ -значение узла  $a$  равно 10, а узла  $s$  — 9. Эти значения обновлены в связи с тем, что были сформированы новые узлы,  $b$  и  $c$ . В настоящее время наиболее перспективным преемником узла  $s$  является  $e$ , для которого  $f$ -значение равно 9.

Обновление  $f$ -значений необходимо для того, чтобы программа имела возможность распознавать на каждом уровне дерева поиска наиболее перспективное поддерево (т.е. дерево, которое содержит наиболее перспективный конечный узел). Такая модификация  $f$ -оценок приводит фактически к обобщению определения функции  $f$ . В результате этого обобщения определение функции  $f$  распространяется с узлов на деревья. Для дерева, состоящего из отдельного узла (листа),  $n$ , было принято такое первоначальное определение:

$$f(n) = g(n) + h(n)$$

А для дерева  $T$ , корнем которого является узел  $n$ , а поддеревьями узла  $n$  — деревья  $S_1, S_2$  и т.д., имеет место следующее определение:

$$f(T) = \min_i f(S_i)$$

Программа поиска по заданному критерию, разработанная в соответствии с этими соглашениями, приведена в листинге 12.1. Ниже даны некоторые дополнительные пояснения к этой программе.

Основной процедурой программы является процедура `expand`, которая имеет следующие шесть параметров:

```
expand(P, Tree, Bound, Tree1, Solved, Solution)
```

Она развертывает текущее дерево (поддерево), при условии, что  $f$ -значение этого дерева остается меньшим или равным значению `Bound`. Ниже приведены определения параметров процедуры `expand`.

- `P`. Путь между начальным узлом и деревом `Tree`.
- `Tree`. Текущее дерево (поддерево) поиска.
- `Bound`. Текущий  $f$ -предел для развертывания дерева `Tree`.
- `Tree1`. Дерево 3, развернутое в пределах `Bound`; следовательно,  $f$ -значение для `Tree1` больше чем `Bound` (если только в процессе этого развертывания не найден целевой узел).
- `Solved`. Индикатор, который может принимать значение “yes”, “no” или “never”.
- `Solution`. Путь решения от начального узла “через `Tree1`” до целевого узла в пределах `Bound` (если такой целевой узел существует).

### Листинг 12.1. Программа поиска по заданному критерию

```
% bestfirst(Start, Solution):
% Решение Solution - это путь от узла Start до целевого узла

bestfirst(Start, Solution) :-
 expand([], l(Start, 0/0), 9999, _, yes, Solution).
% Предполагается, что любое f-значение не превышает 9999

% expand(Path, Tree, Bound, Tree1, Solved, Solution):
```

```

% Path — это путь между начальным узлом поиска и поддеревом, Tree1 — дерево
% Tree, развернутое в пределах Bound; если цель найдена, то Solution — путь
% решения и Solved = yes

% Случай 1. Цель — лист-узел, сформировать путь поиска
expand(P, l(N, _), _, _, yes, [N|P]) :-
 goal(N).

% Случай 2. Лист-узел; f-значение меньше чем Bound. Сформировать
% узлы-преемники и развернуть их в пределах Bound
expand(P, l(N,F/G), Bound, Tree1, Solved, Sol) :-
 F =< Bound,
 (bagof(M/C, (s(N,M,C), not member(M,P)), Succ),
 !, % Узел N имеет преемников
 succlist(G, Succ, Ts), % Создать поддерева Ts
 bestf(Ts, F1), % f-значение лучшего преемника
 expand(P, t(N,F1/G,Ts), Bound, Tree1, Solved, Sol)
);
 Solved = never % Узел N не имеет преемников — тупиковый конец
) .

% Случай 3. Не лист-узел; f-значение меньше чем Bound. Развернуть наиболее
% перспективное поддерево; в зависимости от результатов процедура
% continue примет решение в отношении дальнейших действий
expand(P, t(N,F/G,[T|Ts]), Bound, Tree1, Solved, Sol) :-
 F =< Bound,
 bestf(Ts, BF), min(Bound, BF, Bound1), % Bound1 = min(Bound,BF)
 expand([N|P], T, Bound1, T1, Solved1, Sol),
 continue(P, t(N,F/G,[T1|Ts]), Bound, Tree1, Solved1, Solved, Sol).

% Случай 4. Не лист-узел с пустыми поддеревьями. Это — тупиковое направление,
% в котором не может быть найдено решение
expand(_, t(_,_,[]), _, _, never, _) :- !.

% Случай 5. Получено f-значение больше чем Bound. Дерево больше не может расти
expand(_, Tree, Bound, Tree, no, _) :-
 f(Tree, F), F > Bound.

% continue(Path, Tree, Bound, NewTree, SubtreeSolved, TreeSolved, Solution)
continue(_, _, _, _, yes, yes, Sol).

continue(P, t(N,F/G,[T1|Ts]), Bound, Tree1, no, Solved, Sol) :-
 insert(T1, Ts, NTs),
 bestf(NTs, F1),
 expand(P, t(N,F1/G,NTs), Bound, Tree1, Solved, Sol).

continue(P, t(N,F/G,[_ |Ts]), Bound, Tree1, never, Solved, Sol) :-
 bestf(Ts, F1),
 expand(P, t(N,F1/G,Ts), Bound, Tree1, Solved, Sol).

% succlist(G0, [Node1/Cost1, ...], [l(BestNode,BestF/G), ...]):
% упорядочить список листьев поиска по их F-значениям
succlist(_, [], []).

succlist(G0, [N/C | NCs], Ts) :-
 G is G0 + C,

```



```

h(N, H), % Эвристический терм h(N)
F is G + H,
succlist(G0, NCs, Ts1),
insert(l(N,F/G), Ts1, Ts).

% Вставить дерево T в список деревьев Ts с сохранением упорядоченности
% по отношению к f-значениям

insert(T, Ts, [T | Ts]) :-
 f(T, F), bestf(Ts, F1),
 F <= F1, !.

insert(T, [T1 | Ts], [T1 | Ts1]) :-
 insert(T, Ts, Ts1).

% Извлечь f-значение

f(l(_,F/_), F). % f-значение листа

f(t(_,F/_,_), F). % f-значение дерева

bestf([T|_], F) :- % Лучшее f-значение в списке деревьев
 f(T, F).

bestf([], 9999). % Деревья отсутствуют: неприемлемое f-значение

min(X, Y, X) :-
 X <= Y, !.

min(X, Y, Y).

```

Параметры *P*, *Tree* и *Bound* являются “входными” по отношению к процедуре *expand*; это означает, что при каждом вызове процедуры *expand* они уже конкретизированы. Процедура *expand* вырабатывает результаты трех типов, которые различаются по значению параметра *Solved*, как описано ниже.

**1. Solved = yes.**

*Solution* — путь решения, найденный в результате развертывания дерева *Tree* в пределах *Bound*.

Переменная *Treel* является неконкретизированной.

**2. Solved = no.**

*Treel* — дерево *Tree*, развернутое так, что его *f*-значение превысило *Bound* (рис. 12.3).

Переменная *Solution* является неконкретизированной.

**3. Solved = never.**

Переменные *Treel* и *Solution* являются неконкретизированными.

Последний случай показывает, что дерево *Tree* — это бесперспективный (тупиковый) вариант и программе больше не следует предоставлять другого шанса на повторную активизацию его исследования. Такая ситуация возникает, если *f*-значение дерева *Tree* меньше или равно *Bound*, но дерево не может расти, поскольку в нем ни один узел больше не имеет преемника (т.е. является листом) или имеет преемника, но такого, который создает цикл.

Некоторые предложения, касающиеся отношения *expand*, заслуживают дополнительного описания. В частности, предложение, которое относится к наиболее сложному случаю, когда *Tree* имеет поддеревья, т.е. предложение

```
Tree = t(N, F/G, [T | Ts])
```

обеспечивает выполнение следующих действий. Прежде всего разворачивается наиболее перспективное поддереву,  $T$ . Этому разворачиванию назначается не предел  $\text{Bound}$ , а, возможно, некоторое более низкое значение, в зависимости от  $f$ -значений других конкурирующих поддеревьев,  $T_s$ . Это позволяет гарантировать, что поддерево, растущее в настоящее время, всегда является наиболее перспективным. Затем процесс разворачивания переключается с одного поддерева на другие в соответствии с их  $f$ -значениями. После разворачивания наилучшего поддерева из всех возможных вспомогательная процедура `continue` вырабатывает решение в отношении дальнейших действий; это решение зависит от типа результата, полученного на данном этапе разворачивания. Если найдено решение, оно возвращается, в противном случае разворачивание продолжается. В предложении, которое касается данной ситуации,

`Tree = l( N, F/G)`

вырабатываются узлы-преемники узла  $N$  наряду со значениями стоимостей дуг между  $N$  и узлами-преемниками. Процедура `succlist` создает список поддеревьев, исходящих из данных узлов-преемников, вычисляя также их  $g$ - и  $f$ -значения (рис. 12.4). Затем результирующее дерево продолжает разворачиваться до тех пор, пока позволяет предел  $\text{Bound}$ . Если, с другой стороны, преемников не было, то происходит отказ от какого-либо дальнейшего использования данного листа путем конкретизации значения `Solved = 'never'`.

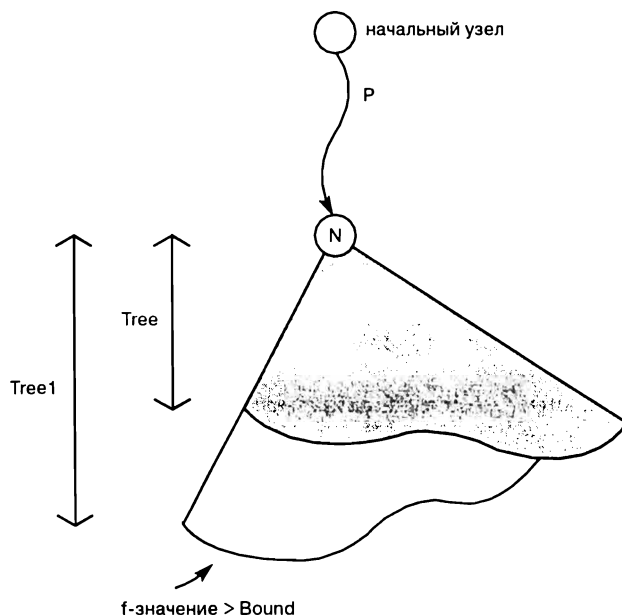


Рис. 12.3. Отношение `expand`: разворачивание дерева `Tree` до тех пор, пока  $f$ -значение не превысит  $\text{Bound}$ , приводит к созданию дерева `Tree1`

Другие отношения описаны ниже.

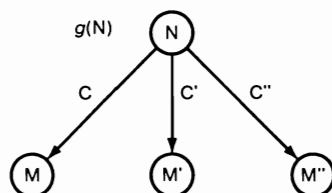
`s( N, M, C)`

где  $M$  — узел-преемник узла  $N$  в пространстве состояний,  $C$  — стоимость дуги от  $N$  до  $M$ .

`h( N, H)`

где  $H$  — эвристическая оценка стоимости наилучшего пути от узла  $N$  до целевого узла.

Примеры применения данной программы поиска по заданному критерию будут даны в следующем разделе. Но вначале приведем некоторые общие, заключительные комментарии к этой программе. Она представляет собой один из вариантов реализации эвристического алгоритма, известного в литературе как алгоритм  $A^*$  (см. ссылки на дополнительные источники, приведенные в конце этой главы). Алгоритм  $A^*$  заслуживает большого внимания, поскольку он представляет собой один из фундаментальных алгоритмов искусственного интеллекта. Ниже приведен один из важных результатов математического анализа алгоритма  $A^*$ .



$$g(M) = g(N) + C$$

$$f(M) = g(M) + h(M)$$

Рис. 12.4. Зависимости между  $g$ -значением узла  $N$ , а также  $f$ - и  $g$ -значениями его дочерних узлов в пространстве состояний

Алгоритм поиска называется *допустимым*, если он всегда вырабатывает оптимальное решение (т.е. путь с минимальной стоимостью), при условии, что решение вообще существует. Рассматриваемая здесь реализация, в которой все решения вырабатываются с помощью перебора с возвратами, может считаться допустимой, если оптимальным является первое же из найденных решений. Предположим, что для каждого узла  $n$  в пространстве состояний как  $h^*(n)$  обозначена стоимость оптимального пути от  $n$  до целевого узла. В теореме, касающейся допустимости алгоритма  $A^*$ , утверждается следующее: допустимым является любой алгоритм  $A^*$ , в котором используется эвристическая функция  $h$ , такая, что для всех узлов  $n$  в пространстве состояний справедливо следующее утверждение:

$$h(n) \leq h^*(n)$$

Этот результат имеет большое практическое значение. Даже если неизвестно точное значение  $h^*$ , достаточно найти нижнюю границу  $h^*$  и использовать ее в качестве  $h$  в алгоритме  $A^*$ . Это служит достаточной гарантией того, что алгоритм  $A^*$  вырабатывает оптимальное решение. В частности, может рассматриваться тривиальная нижняя граница, а именно:

$$h(n) = 0$$

для всех  $n$  в пространстве состояний.

Такой вариант действительно гарантирует допустимость. Но недостаток использования условия  $h = 0$  состоит в том, что оно не имеет эвристического потенциала и не обеспечивает какого-либо управления процессом поиска. Алгоритм  $A^*$ , в котором используется  $h = 0$ , ведет себя аналогично алгоритму поиска в ширину. Мало того, он фактически сводит поиск в ширину к такому случаю, что функция стоимости дуг принимает значение  $c(n, n') = 1$  для всех дуг  $(n, n')$  в пространстве состояний. Такое отсутствие эвристического потенциала приводит к значительному увеличению потребности в ресурсах. Поэтому желательно иметь такое значение  $h$ , которое является нижним пределом  $h^*$  (для обеспечения допустимости), а также в максимально возможной степени приближается к  $h^*$  (для обеспечения эффективности). В идеале,

если была бы известна стоимость  $h^*$ , то в алгоритме можно было бы использовать само значение  $h^*$ , поскольку алгоритм  $A^*$ , в котором используется значение  $h^*$ , находит оптимальное решение непосредственно, вообще без какого-либо перебора с возвратами.

## Упражнения

- 12.1. Определите отношения  $s$ ,  $goal$  и  $h$  для задачи поиска маршрута, показанной на рис. 12.2, с учетом особенностей рассматриваемой проблемы. Изучите поведение приведенной в данной главе программы  $A^*$  при решении этой задачи.
- 12.2. Следующее утверждение на первый взгляд напоминает теорему допустимости: “Для любой задачи поиска, если  $A^*$  находит оптимальное решение, то  $h(n) \leq h^*(n)$  для всех узлов  $n$  в пространстве состояний”. Является ли это утверждение правильным?
- 12.3. Предположим, что  $h_1$ ,  $h_2$  и  $h_3$  — три допустимые эвристические функции ( $h_i \leq h^*$ ), которые поочередно применяются алгоритмом  $A^*$  в одном и том же пространстве состояний. Объедините эти три функции еще в одну эвристическую функцию,  $h$ , которая также допустима и направляет поиск не хуже любой из трех функций  $h_i$ , отдельно взятых.
- 12.4. Подвижный робот перемещается на плоскости  $x$ - $y$  между препятствиями. Все препятствия представляют собой прямоугольники, выровненные вдоль осей  $y$  и  $x$ . Робот может перемещаться только в направлениях  $x$  и  $y$  и имеет такие небольшие размеры, что может быть приближенно представлен в виде точки. Робот должен планировать пути перемещения без столкновений с препятствиями от его текущей позиции до некоторой указанной целевой позиции. Робот стремится свести к минимуму длину пути и количество изменений направления движения (допустим, что стоимость одной смены направления равна стоимости перемещения на одну единицу длины). В роботе для поиска оптимальных путей используется алгоритм  $A^*$ . Определите предикаты  $s(\text{State}, \text{NewState}, \text{Cost})$  и  $h(\text{State}, H)$  для использования в программе  $A^*$  в процессе решения этой задачи поиска (желательно, чтобы эти предикаты были допустимыми). Предположим, что целевая позиция для робота определяется с помощью предиката  $goal(Xg/Yg)$ , где  $Xg$  и  $Yg$  — координаты  $x$  и  $y$  целевой точки. Препятствия представлены с помощью следующего предиката:  
`obstacle( Xmin/Ymin, Xmax/Ymax)`  
где  $Xmin/Ymin$  — нижний левый угол препятствия, а  $Xmax/Ymax$  — его верхний правый угол.

## 12.2. Применение поиска по заданному критерию для решения головоломки “игра в восемь”

Чтобы можно было применить программу поиска по заданному критерию, приведенную в листинге 12.1, для решения некоторой конкретной задачи, необходимо определить отношения, характеризующие рассматриваемую проблему. Эти отношения характеризуют конкретную проблему (“правила игры”), а также воплощают в себе эвристическую информацию о том, как решить данную задачу. Такая эвристическая информация предоставляется в форме эвристической функции.

Предикаты, касающиеся конкретной проблемы, задаются в виде следующего отношения:

```
s(Node, Node1, Cost)
```

Это отношение является истинным, если в пространстве состояний между узлами Node и Node1 имеется дуга со стоимостью Cost. Отношение

```
goal(Node)
```

является истинным, если Node — целевой узел в пространстве состояний. А в отношении

```
h(Node, H)
```

переменная H — эвристическая оценка стоимости пути с минимальной стоимостью от узла Node до целевого узла.

В этом и следующем разделах такие отношения будут определены для двух примеров проблемных областей: головоломка “игра в восемь” (описанная в разделе 11.1) и задача составления расписаний.

Отношения, касающиеся проблемной области для головоломки “игра в восемь”, приведены в листинге 12.2. Любой из узлов в пространстве состояний представляет собой некоторую конфигурацию фишек в коробке. В данной программе эта конфигурация представлена в виде списка текущих положений фишек. Каждая позиция обозначается парой координат, X/Y. Порядок элементов в списке является следующим.

1. Текущая позиция пустого квадрата.
2. Текущая позиция фишки 1.
3. Текущая позиция фишки 2.
4. ...

Целевая ситуация (см. рис. 11.3), в которой все фишки находятся на своих исходных клетках, определяется следующим предложением:

```
goal([2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]).
```

## Листинг 12.2. Относящиеся к проблемной области процедуры для головоломки “игра в восемь”, которые должны использоваться в программе поиска по заданному критерию, приведенной в листинге 12.1

```
/* Относящиеся к проблемной области процедуры для головоломки "игра в восемь"
```

Текущая позиция представлена списком координат фишек, в котором на первом месте указаны координаты пустой клетки

Пример

|   |   |   |   |
|---|---|---|---|
| 3 | 1 | 2 | 3 |
| 2 | 8 |   | 4 |
| 1 | 7 | 6 | 5 |
|   | 1 | 2 | 3 |

Для представления этой позиции служит следующий список:

```
[2/2, 1/3, 2/3, 3/3, 3/2, 3/1, 2/1, 1/1, 1/2]
```

Принято считать, что пустая клетка — это “пустая” фишка, которая может передвигаться по горизонтали или по вертикали на место, занимаемое любой из ее соседних фишек; это означает, что “пустая” фишка и ее соседняя фишка могут поменяться местами

```
*/
% s(Node, SuccessorNode, Cost)
s([Empty | Tiles], [Tile | Tiles1], 1) :- % Стоимости всех дуг равны 1
 swap(Empty, Tile, Tiles, Tiles1). % Поменять местами пустую фишку
 % Empty и фишку Tile в списке Tiles

swap(Empty, Tile, [Tile | Ts], [Empty | Ts]) :-
 mandist(Empty, Tile, 1). % Манхэттенское расстояние равно 1
```

```

swap(Empty, Tile, [T1 | Ts], [T1 | Ts1]) :-
 swap(Empty, Tile, Ts, Ts1).

mandist(X/Y, X1/Y1, D) :- % D - это манхэттенское расстояние
 % между двумя клетками
 dif(X, X1, Dx),
 dif(Y, Y1, Dy),
 D is Dx + Dy.

dif(A, B, D) :- % D равно |A-B|
 D is A-B, D >= 0, !
 ;
 D is B-A.

% Эвристическая оценка h представляет собой сумму расстояний от каждой фишки
% до ее "исходной" клетки плюс утроенное значение "оценки упорядоченности"

h([Empty | Tiles], H) :-
 goal([Empty1 | GoalSquares]),
 totdist(Tiles, GoalSquares, D), % Суммарное расстояние от исходных клеток
 seq(Tiles, S), % Оценка упорядоченности
 H is D + 3*S.

totdist([], [], 0).

totdist([Tile | Tiles], [Square | Squares], D) :-
 mandist(Tile, Square, D1),
 totdist(Tiles, Squares, D2),
 D is D1 + D2.

% seq(TilePositions, Score): оценка упорядоченности

seq([First | OtherTiles], S) :-
 seq([First | OtherTiles], First, S).

seq([Tile1, Tile2 | Tiles], First, S) :-
 score(Tile1, Tile2, S1),
 seq([Tile2 | Tiles], First, S2),
 S is S1 + S2.

seq([Last], First, S) :-
 score(Last, First, S).

score(2/2, _, 1) :- !. % Оценка фишки, стоящей в центре, равна 1

score(1/3, 2/3, 0) :- !. % Оценка фишки, за которой следует
 % допустимый преемник, равна 0

score(2/3, 3/3, 0) :- !.
score(3/3, 3/2, 0) :- !.
score(3/2, 3/1, 0) :- !.
score(3/1, 2/1, 0) :- !.
score(2/1, 1/1, 0) :- !.
score(1/1, 1/2, 0) :- !.
score(1/2, 1/3, 0) :- !.

score(_, _, 2). % Оценка фишки, за которой следует
 % недопустимый преемник, равна 2
goal([2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2]). % Исходные клетки для фишек

% Показать путь решения как список позиций на доске

showsol([]).

showsol([P | L]) :-
 showsol(L),

```

```

nl, write('---'),
showpos(P).

% Показать позицию на доске

showpos([S0,S1,S2,S3,S4,S5,S6,S7,S8]) :-
 member(Y, [3,2,1]), % Последовательность координат Y
 nl, member(X, [1,2,3]), % Последовательность координат X
 member(Tile-X/Y, % Фишка в клетке X/Y
 [' '-S0,1-S1,2-S2,3-S3,4-S4,5-S5,6-S6,7-S7,8-S8]),
 write(Tile),
 fail % Выполнить перебор с возвратом к следующей клетке
 ;
 true. % Обработка всех клеток закончена

% Начальные позиции для некоторых задач

start1([2/2,1/3,3/2,2/3,3/3,3/1,2/1,1/1,1/2]). % Требуется 4 хода
start2([2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3]). % Требуется 5 ходов
start3([2/2,2/3,1/3,3/1,1/2,2/1,3/3,1/1,3/2]). % Требуется 18 ходов

% Пример запроса: ?- start1(Pos), bestfirst(Pos, Sol), showsol(Sol).

```

При определении этих отношений применяется следующее вспомогательное отношение:

```
mandist(S1, S2, D).
```

где *D* — *манхэттенское расстояние* между клетками *S1* и *S2*; оно измеряется как сумма расстояний между *S1* и *S2* в горизонтальном и вертикальном направлениях.

Необходимо найти путь к решению минимальной длины. Поэтому определим стоимость всех дуг в пространстве состояний как равную 1. В программе, приведенной в листинге 12.2, заданы также три примера начальных позиций, которые составлены по диаграммам, показанным на рис. 12.5.

|   |   |   |
|---|---|---|
| 1 | 3 | 4 |
| 8 |   | 2 |
| 7 | 6 | 5 |

а)

|   |   |   |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 |   | 5 |

б)

|   |   |   |
|---|---|---|
| 2 | 1 | 6 |
| 4 |   | 8 |
| 7 | 5 | 3 |

в)

Рис. 12.5. Три начальные позиции для головоломки “игра в восемь”: а) требует 4 хода; б) требует 5 ходов; в) требует 18 ходов

Эвристическая функция *h* определяется в программе следующим образом:

```
h(Pos, H)
```

где *Pos* — позиция на доске, *H* — сочетание двух описанных ниже критериев.

1. *totdist*. Суммарное расстояние восьми фишек в позиции *Pos* от тех клеток, называемых *исходными*, в которых должны находиться фишки после решения головоломки. Например, в начальной позиции головоломки, приведенной на рис. 12.5, а, *totdist* = 4.
2. *seq*. Оценка упорядоченности, определяющая ту степень, в какой фишки, стоящие в текущей позиции, уже упорядочены по отношению к той последовательности, которая должна быть достигнута в целевой конфигурации. Значе-

ние  $seq$  вычисляется как сумма оценок для каждой фишки в соответствии со следующими правилами:

- фишка в центре имеет оценку 1;
- фишка, стоящая на нецентральной клетке, имеет оценку 0, если за ней в направлении по часовой стрелке следует соответствующий ей преемник (например, если за фишкой 1 следует фишка 2);
- такая фишка имеет оценку 2, если за ней не следует соответствующий ей преемник.

Например, для начальной позиции головоломки, приведенной на рис. 12.5, *а*,  $seq = 6$ .

Эвристическая оценка,  $H$ , вычисляется как  $H = totdist + 3 * seq$ .

Эта эвристическая функция действует успешно в том смысле, что очень эффективно направляет поиск к цели. Например, при решении задач, показанных на рис. 12.5, *а*, *б*, не происходило даже разворачивание ни одного узла, выходящего за пределы кратчайшего пути решения, до того, как было найдено первое решение. Это означает, что в таких случаях кратчайшие пути решения отыскиваются непосредственно, без какого-либо перебора с возвратами. Почти сразу же решается даже трудная задача, приведенная на рис. 12.5, *в*. Но недостаток этой эвристики состоит в том, что она не является допустимой: она не гарантирует, что кратчайший путь решения всегда будет найден до обнаружения какого-либо более длинного решения. Дело в том, что данная функция  $h$  не удовлетворяет условию допустимости, при котором  $h \leq h^*$  для всех узлов. Например, для начальной позиции, приведенной на рис. 12.5, *а*, имеет место следующее:

$$h = 4 + 3 * 6 = 22, \quad h^* = 4$$

С другой стороны, допустимым является отдельно взятый критерий “суммарного расстояния”, поскольку для всех позиций имеет место следующее:

$$totdist \leq h^*$$

Допустимость этого отношения можно легко проверить с помощью таких рассуждений: если условия этой головоломки будут упрощены таким образом, что фишки можно будет переносить друг над другом, то каждая фишка сможет переходить к своей исходной клетке по траектории, длина которой точно равна манхэттенскому расстоянию между начальной клеткой фишки и ее исходной клеткой. Поэтому оптимальное решение в упрощенной головоломке будет точно равно длине  $totdist$ . Но в первоначальном варианте задачи фишки могут мешать друг другу и одна из них может находиться на пути другой, что препятствует движению фишек по кратчайшим траекториям, а это гарантирует, что длина оптимального решения равна или больше чем  $totdist$ .

## Упражнение

12.5. Модифицируйте программу поиска по заданному критерию, приведенную в листинге 12.1, таким образом, чтобы в ней подсчитывалось количество узлов, выработанных в процессе поиска. Один из простых способов состоит в том, чтобы текущее значение количества узлов вносилось в базу данных как факт и обновлялось с помощью предикатов `retract` и `assert` после каждой выработки новых узлов. Проведите эксперименты с различными эвристическими функциями для головоломки “игра в восемь” для оценки их эвристического потенциала, который характеризуется количеством выработанных узлов.



# 12.3. Применение поиска по заданному критерию при планировании

Рассмотрим следующую проблему планирования заданий. Предположим, что дано множество заданий,  $t_1, t_2, \dots$ , и соответствующих им значений продолжительности выполнения,  $D_1, D_2, \dots$ . Задания предназначены для выполнения множеством идентичных процессоров  $m$ . Любое задание может быть выполнено любым процессором, но каждый процессор способен одновременно выполнять только одно задание. Между заданиями определено отношение предшествования, которое указывает, какие задания, если они имеются, должны быть завершены, прежде чем можно будет начать выполнение какого-то другого задания. Проблема планирования состоит в распределении заданий по процессорам таким образом, чтобы не нарушалось отношение предшествования и все задания в целом были обработаны за кратчайшее возможное время. Время окончания последнего задания в расписании называется *временем завершения расписания*. Среди всех возможных расписаний необходимо найти такое, при котором время завершения становится минимальным.

На рис. 12.6 показана такая задача планирования заданий и приведены два допустимых расписания, одно из которых является оптимальным. Этот пример демонстрирует интересное свойство оптимальных расписаний, которое состоит в том, что они могут включать *время простоя* для процессоров. В оптимальном расписании, приведенном на рис. 12.6, процессор 2 после выполнения задания  $t_2$  ожидает в течение двух единиц времени, хотя он мог бы сразу же начать выполнение задания  $t_7$ .

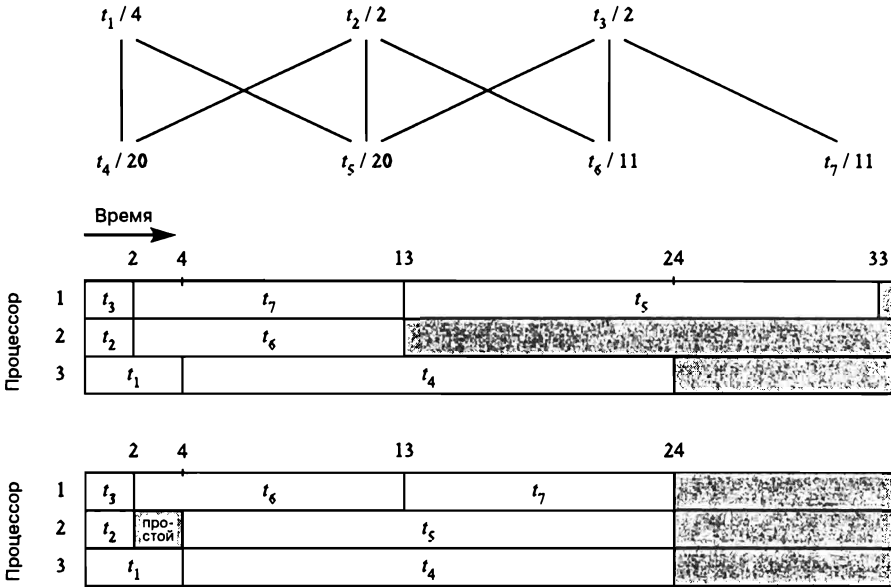


Рис. 12.6. Задача планирования заданий, в условия которой входят семь заданий и три процессора. В верхней части этого рисунка показаны отношение предшествования заданий и продолжительность заданий. Например, для задания  $t_5$  требуется 20 единиц времени и его выполнение может начаться только после завершения трех других заданий,  $t_1, t_2$  и  $t_3$ . Показаны два допустимых расписания: оптимальное, с временем завершения 24, и неоптимальное, с временем завершения 33. При решении данной задачи любое оптимальное расписание должно включать время простоя (см. [34], с. 86). Этот рисунок адаптирован и включен в настоящую книгу с разрешения издательства Prentice Hall, Englewood Cliffs, New Jersey

Один из способов формирования расписания, грубо говоря, состоит в следующем. Работа начинается с пустого расписания (с нераспределенными интервалами времени для каждого процессора), и в это расписание постепенно одно за другим вносятся задания до тех пор, пока не будут внесены все задания. Обычно на каждом из этих этапов внесения заданий в расписание имеется несколько вариантов, поскольку всегда ожидает обработки несколько возможных заданий. Поэтому задача составления расписания представляет собой задачу поиска. В соответствии с этим задачу составления расписания можно сформулировать как задачу поиска в пространстве состояний следующим образом:

- состояния представляют собой частично составленные расписания;
- состояние-преемник некоторого частично составленного расписания можно получить, введя в это расписание еще не запланированное задание; еще одна возможность состоит в том, что процессор, который выполнил свое текущее задание, может быть оставлен в состоянии простоя;
- начальным состоянием является пустое расписание;
- целевым состоянием является любое расписание, которое включает все задания, рассматриваемые в данной задаче;
- стоимость решения (которая должна быть минимизирована) представляет собой время завершения целевого расписания;
- согласно этим условиям, стоимость перехода между двумя (частично составленными) расписаниями, время завершения которых соответственно равно  $F_1$  и  $F_2$ , представляет собой разность  $F_2 - F_1$ .

В этот ориентировочный сценарий необходимо внести некоторые уточнения. Во-первых, принято заполнять расписание по возрастанию значений времени таким образом, чтобы задания вносились в расписание слева направо. Кроме того, при внесении каждого задания следует проверять, соблюдается ли ограничение предшествования. К тому же нет смысла оставлять какой-либо процессор в состоянии простоя на неопределенно долгое время, если все еще имеются какие-то возможные задания, ожидающие выполнения. Поэтому решено оставлять процессор в состоянии простоя только до тех пор, пока какой-то другой процессор не завершит свое текущее задание, а затем снова рассматривать возможность назначения задания простаивающему процессору.

Теперь необходимо выбрать способ представления проблемных ситуаций, т.е. частично составленных расписаний. Для решения задачи составления расписания требуется следующая информация.

1. Список ожидающих заданий и значений времени их выполнения.
2. Текущие назначения процессоров.

Кроме того, для удобства можно также учитывать следующую информацию.

3. Время завершения (частично составленного) расписания, другими словами, последнее значение времени окончания обработки работы процессорами текущих назначений.

Список ожидающих заданий и значений времени их выполнения будет представлен в программе в виде списка в следующей форме:

```
[Task1/D1, Task2/D2, ...]
```

Текущие назначения процессоров представляются в виде списка заданий, обрабатываемых в настоящее время; это означает, что должны использоваться пары с определением заданий и времени их завершения в следующей форме:

```
Task/FinishingTime
```

В списке имеется  $m$  таких пар, по одной для каждого процессора. Новое задание всегда вносится в расписание в тот момент, когда завершается выполнение первого предшествующего ему задания. Для этого список текущих назначений поддержива-

ются в отсортированном виде в соответствии с возрастающими значениями времени завершения. Три компонента частично составленного расписания (ожидающие задания, текущие назначения и время завершения) объединяются в программе в одно выражение:

WaitingList \* ActiveTasks \* FinishingTime

Кроме этой информации, имеется ограничение предшествования, которое задано в программе в виде отношения:

prec( TaskX, TaskY)

Теперь рассмотрим эвристическую оценку. Для этого будет использоваться довольно простая эвристическая функция, которая не обеспечивает достаточно эффективного управления при осуществлении алгоритма поиска в условиях крупномасштабных задач. Тем не менее эта функция является допустимой и поэтому гарантирует получение оптимального расписания. Но следует отметить, что для решения большинства задач составления расписаний требуется гораздо более мощная эвристическая функция.

Рассматриваемая эвристическая функция представляет собой оптимистическую оценку времени завершения частично составленного расписания, дополненного всеми ожидающими в настоящее время заданиями. Эта оптимистическая оценка вычисляется исходя из предположения, что ослаблены следующие ограничения на действительные расписания.

1. Удалено ограничение предшествования.
2. Принято (нереальное) допущение, что задание может выполняться в режиме распределения по нескольким процессорам и что сумма значений времени выполнения этого задания всеми соответствующими процессорами равна первоначально заданному времени выполнения данного задания на одном процессоре.

Предположим, что значения времени выполнения ожидающих в настоящее время заданий равны  $D_1, D_2, \dots$ , а значения времени завершения текущих назначений процессоров равны  $F_1, F_2, \dots$ . Такая оптимистическая оценка времени завершения,  $Finall$ , позволяющего выполнить все активные в настоящее время и все ожидающие задания, определяется по следующей формуле:

$$Finall = (\sum_i D_i + \sum_j F_j) / m$$

где  $m$  — количество процессоров. Предположим, что время завершения текущего частично составленного расписания определяется выражением:

$$Fin = \max_j (F_j)$$

В таком случае эвристическая оценка  $H$  (дополнительное время, требуемое для дополнения частично составленного расписания ожидающими заданиями) может быть определена следующим образом:

если  $Finall > Fin$ , то  $H = Finall - Fin$ , иначе  $H = 0$

Полная программа, в которой в соответствии с описанными выше условиями дано определение отношений пространства состояний для задачи планирования заданий, приведена в листинге 12.3. Кроме того, в этом листинге имеется спецификация конкретной задачи составления расписания, показанной на рис. 12.6. Эти определения могут также использоваться в программе поиска по заданному критерию, приведенной в листинге 12.1. Одним из оптимальных решений, сформированных с помощью поиска по заданному критерию в проблемной области, определенной таким образом, является оптимальное расписание, показанное на рис. 12.6.

**Листинг 12.3. Отношения, касающиеся рассматриваемой проблемной области, для задачи планирования заданий. Конкретная задача составления расписания, приведенная на рис. 12.6, определена также с помощью ее графа предшествования и начального (пустого) расписания, применяемого в качестве начального узла для поиска**

```

/* Отношения, касающиеся рассматриваемой проблемной области,
 для задачи планирования заданий

Узлы в этом пространстве состояний представляют собой частичные расписания,
заданные следующим образом:

[WaitingTask1/D1, WaitingTask2/D2, ...] * H
[Task1/F1, Task2/F2, ...] * FinTime

Первый лист определяет ожидающие задания и значения их продолжительности;
второй список определяет задания, выполняемые в настоящее время, и значения
времени их завершения, упорядоченные так, что F1 <= F2, F2 <= F3
Fintime – это самое последнее время завершения при текущем назначении
процессоров
*/

% s(Node, SuccessorNode, Cost)

s(Tasks1 * [_/F | Active1] * Fin1, Tasks2 * Active2 * Fin2, Cost) :-
 del(Task/D, Tasks1, Tasks2), % Выбрать ожидающую задачу
 not (member(T/_, Tasks2), before(T, Task)), % Проверить правильность
 % упорядочения,
 not (member(T1/F1, Active1), F < F1, before(T1, Task)), % в том числе
 % активных задач
 Time is F + D, % Время завершения активных задач
 insert(Task/Time, Active1, Active2, Fin1, Fin2),
 Cost is Fin2 - Fin1.

s(Tasks * [_/F | Active1] * Fin, Tasks * Active2 * Fin, 0) :-
 insertidle(F, Active1, Active2). % Оставить процессор в состоянии простоя

before(T1, T2) :-
 % Согласно правилам упорядочения задание
 % T1 предшествует заданию T2
 prec(T1, T2).

before(T1, T2) :-
 prec(T, T2),
 before(T1, T).

insert(S/A, [T/B | L], [S/A, T/B | L], F, F) :- % Списки заданий упорядочены
 A <= B, !.

insert(S/A, [T/B | L], [T/B | L1], F1, F2) :-
 insert(S/A, L, L1, F1, F2).

insert(S/A, [], [S/A], _, A).

insertidle(A, [T/B | L], [idle/B, T/B | L]) :- % Оставить процессор в
 A < B, !. % состоянии простоя до наступления
 % следующего, очередного времени завершения

insertidle(A, [T/B | L], [T/B | L1]) :-
 insertidle(A, L, L1).

del(A, [A | L], L). % Удалить элемент из списка

del(A, [B | L], [B | L1]) :-
 del(A, L, L1).

goal([] * _ * _). % Целевое состояние – ни одно из заданий не находится
 % в состоянии ожидания

```

```

% Эвристическая оценка частичного расписания основана на оптимистической оценке
% последнего времени завершения данного частично составленного расписания,
% дополненного всеми ожидающими заданиями

h(Tasks * Processors * Fin, H) :-
 totaltime(Tasks, Totttime), % Общая продолжительность ожидающих заданий
 sumnum(Processors, Ftime, N), % Ftime - сумма значений времени завершения
 % для всех процессоров, N - количество этих значений
 Finall is (Totttime + Ftime)/N,
 (Finall > Fin, !, H is Finall - Fin
 ;
 H = 0
).

totaltime([], 0).

totaltime([_/D | Tasks], T) :-
 totaltime(Tasks, T1),
 T is T1 + D.

sumnum([], 0, 0).

sumnum([_/T | Procs], FT, N) :-
 sumnum(Procs, FT1, N1),
 N is N1 + 1,
 FT is FT1 + T.

% Граф предшествования заданий

prec(t1, t4). prec(t1, t5). prec(t2, t4). prec(t2, t5).
prec(t3, t5). prec(t3, t6). prec(t3, t7).

% Начальный узел

start([t1/4, t2/2, t3/2, t4/20, t5/20, t6/11, t7/11] *
 [idle/0, idle/0, idle/0] * 0).

% Пример запоса: start(Problem), bestfirst(Problem, Sol).

```

---

## Проект

Как известно, в целом задачи составления расписания характеризуются комбинаторной сложностью. Описанная выше простая эвристическая функция не может служить достаточно мощным средством управления процессом поиска. Предложите другие функции и проведите с ними эксперименты.

## 12.4. Методы экономии пространства для поиска по заданному критерию

### 12.4.1. Потребность алгоритма A\* в ресурсах времени и пространства

Применение эвристических средств управления поиском по заданному критерию обычно позволяет сократить область поиска таким образом, чтобы в этом процессе посещались только узлы небольшой части пространства состояний задачи. Такое уменьшение области поиска можно рассматривать как сокращение эффективного объема ветвления в процессе поиска. Таким образом, если “среднее” ветвление в про-

странстве состояний задачи равно  $b$ , то применение эвристических средств управления поиском фактически приводит к возникновению среднего ветвления  $b'$ , причем  $b'$  обычно значительно меньше чем  $b$ .

Несмотря на существенное уменьшение затрат на выполнение поиска, порядок сложности алгоритма  $A^*$  (и его реализации, приведенной в листинге 12.1) продолжает оставаться экспоненциально зависимым от глубины поиска. Такое утверждение остается справедливым в отношении потребности в ресурсах и времени, и пространства, поскольку данный алгоритм предусматривает необходимость сопровождения информации обо всех сформированных узлах в памяти. В практических приложениях наиболее важным может стать один из этих ресурсов (время или пространство), и это зависит от конкретных обстоятельств. Но в большинстве практических ситуаций более важным ресурсом является пространство. Алгоритм  $A^*$  способен за считанные минуты израсходовать все доступное пространство памяти. А после этого поиск фактически не может продолжаться, притом что пользователи часто готовы ждать окончания работы этого алгоритма в течение многих часов или даже суток, поскольку для них весьма важны его результаты.

Разработано несколько вариантов алгоритма  $A^*$ , позволяющих экономить пространство за счет времени. По своему основному замыслу они аналогичны поиску в глубину с итеративным углублением (см. главу 11). Потребность в ресурсах пространства сокращается от экспоненциальной до линейной зависимости от глубины поиска. Но за это приходится платить тем, что в пространстве поиска повторно формируются ранее сформированные узлы.

В следующих разделах рассматриваются два метода экономии пространства в контексте поиска по заданному критерию. Первый из них называется  $IDA^*$  (Iterative Deepening  $A^*$  — алгоритм  $A^*$  с итеративным углублением), а второй упоминается под названием RBFS (Recursive Best-First Search — рекурсивный поиск по заданному критерию).

## 12.4.2. Метод $IDA^*$

Метод  $IDA^*$  (Iterative Deepening  $A^*$  — алгоритм  $A^*$  с итеративным углублением) аналогичен поиску в глубину с итеративным углублением, за исключением описанного ниже отличия. При итеративном углублении поиск в глубину осуществляется со все возрастающими пределами глубины. При каждой итерации поиск в глубину ограничивается текущим пределом глубины. Но в методе  $IDA^*$  последовательный поиск в глубину ограничивается текущим пределом значений оценок узлов (т.е. эвристических  $f$ -значений узлов). Поэтому основной механизм поиска в методе  $IDA^*$  также основан на поиске в глубину, при котором потребность в ресурсах пространства является очень низкой. Метод  $IDA^*$  можно сформулировать с помощью приведенной ниже алгоритмической конструкции.

```
Bound := f(StartNode);
```

```
Repeat
```

```
 выполнить поиск в глубину, начиная с узла StartNode, с соблюдением условия, что узел N развертывается, только если $f(N) \leq \text{Bound}$;
```

```
 if при этом поиске в глубину встречается целевой узел
```

```
 then сообщить о том, что "решение найдено",
```

```
 else
```

```
 вычислить NewBound как минимум f -значений узлов, достигнутых непосредственно после выхода за пределы Bound:
```

```
 NewBound = $\min \{f(N) \mid \text{узел } N, \text{ сгенерированный на этом этапе поиска, } f(N) > \text{Bound}\}$
```

```
 Bound := NewBound
```

```
until "решение найдено"
```

В качестве иллюстрации работы этого алгоритма рассмотрим применение алгоритма  $A^*$  к задаче поиска маршрута (рис. 12.2, б), при условии, что  $f(s) = 6$ . При этом метод  $IDA^*$  действует следующим образом:

Bound = f(s) = 6

Выполнить поиск в глубину, ограниченный значением  $f \leq 6$ . На этом этапе поиска развертывается узел s, формируются узлы a и e и обнаруживается следующее:

f(a) = 7 > Bound

f(e) = 9 > Bound

NewBound = min { 7, 9 } = 7

Выполнить поиск в глубину, ограниченный значением  $f \leq 7$ , начиная с узла s.

Узлами с f-значениями, непосредственно превышающими этот предел, являются b и e.

NewBound = min { f(b), f(e) } = min { 8, 9 } = 8

В последующем предел Bound изменяется следующим образом: 9 (f(e)), 10 (f(c)), 11 (f(f)). Для каждого из этих значений выполняется поиск в глубину.

При выполнении поиска в глубину со значением Bound = 11 возникает ситуация "решение найдено".

Хотя этот пример предназначен только для иллюстрации работы метода IDA\*, он оставляет впечатление, что реализованный в нем алгоритм является громоздким, поскольку поиск в глубину повторяется несколько раз. Но при решении крупных задач поиска экономия пространства, достигаемая с помощью метода IDA\*, может оказаться весьма значительной, тогда как издержки, связанные с повторением поиска, остаются вполне приемлемыми. Величина этих издержек зависит от свойств пространства поиска, в частности от свойств функции оценки f. В благоприятных ситуациях многие узлы имеют равные f-значения. В подобных ситуациях на каждом очередном этапе поиска в глубину рассматривается много новых узлов, количество которых превышает количество повторно формируемых узлов. Поэтому дополнительные издержки сравнительно малы. В неблагоприятных ситуациях f-значения, как правило, не являются одинаковыми для многих узлов. В крайнем случае каждый узел может иметь отличное от других f-значение. При этом возникает необходимость использовать целый ряд последовательных f-пределов, когда при каждом новом поиске в глубину формируется только один новый узел, а все остальные узлы представляют собой повторно сформированные узлы (которые были сформированы на предыдущих этапах и удалены из памяти). В таких крайних случаях дополнительные издержки, связанные с использованием метода IDA\*, действительно становятся неприемлемыми.

Еще одно интересное свойство метода IDA\* касается допустимости. Предположим, что функция f(N) определена как  $g(N) + h(N)$  для всех узлов N. Если функция h является допустимой (т.е.  $h(N) \leq h^*(N)$  для всех N), то метод IDA\* гарантирует нахождение оптимального решения.

Одним из возможных недостатков метода IDA\* является то, что он не обеспечивает исследование узлов в порядке оценок по заданному критерию (т.е. в порядке возрастания f-значений). Например, предположим, что f — это функция оценки, которая не обязательно представлена в форме  $f = g + h$ . Если функция f не монотонна, то упорядочение по заданному критерию не гарантируется. Функция f называется *монотонной*, если ее значение монотонно возрастает вдоль путей в пространстве состояний. Иными словами, f является монотонной, если для всех пар узлов N и N' справедливо утверждение: если  $s(N, N')$ , то  $f(N) \leq f(N')$ . Причина нарушения упорядоченности по заданному критерию состоит в том, что при использовании немонотонной функции f значение f-предела может стать настолько большим, что узлы с разными f-значениями будут впервые развертываться при этом поиске в глубину. Процедура поиска в глубину будет развертывать узлы до тех пор, пока для них значение функции f не превысит f-предела, без учета того, в каком порядке они развертываются. В принципе мы заинтересованы в соблюдении упорядоченности по заданному критерию, поскольку надеемся на то, что функция f отражает качество решений.

Один из простых способов реализации метода IDA\* на языке Prolog показан в листинге 12.4. В этой программе широко применяется механизм перебора с возвратами Prolog. Значение f-предела сопровождается как факт в форме

```
next_bound(Bound)
```

который обновляется с помощью предикатов assert и retract. При каждой итерации предел для поиска в глубину определяется из этого факта. Затем (в результате

применения предикатов `retract` и `assert` к этому факту) предел для следующей итерации инициализируется значением 99999. Предполагается, что эта большая величина превышает любое возможное  $f$ -значение. Поиск в глубину реализован в программе таким образом, что развертывание узла  $N$  допускается, только если  $f(N) \leq \text{Bound}$ . Если же, с другой стороны,  $f(N) > \text{Bound}$ , то значение  $f(N)$  сравнивается со значением `NextBound` (которое хранится в виде факта `next_bound(NextBound)`). Если  $f(N) < \text{NextBound}$ , то факт `next_bound` обновляется для сохранения в нем значения  $f(N)$ .

## Упражнения

**12.6.** Сформируйте пространство состояний и функцию  $f$ , при использовании которых метод IDA\* не развертывает узлы в порядке соблюдения заданного критерия.

**12.7.** Примените программу IDA\*, приведенную в листинге 12.4, к головоломке “игра в восемь” с использованием определений отношения выбора преемника  $s/3$  и отношения  $\text{totdist}/3$ , показанных в листинге 12.2. В качестве эвристической функции  $h$  (для обеспечения допустимости) применяйте только суммарное расстояние ( $\text{totdist}/3$ ). Определите также предикат  $f/2$  (необходимый для программы IDA\*) таким образом, чтобы  $f(N) = g(N) + h(N)$ . Для обеспечения возможности вычисления  $g(N)$  храните  $g$ -значение узла явно в составе представления этого узла (например,  $N = G:\text{TilePositions}$ ). Проведите эксперименты с головоломкой “игра в восемь”, которая определена в листинге 12.2. Попробуйте также применить начальное состояние  $[1/2, 3/3, 3/1, 1/3, 3/2, 1/1, 2/3, 2/1, 2/2]$ . Сравните значения времени выполнения и длины пути решения, найденные с помощью алгоритма A\* (с использованием эвристической функции листинга 12.2) и метода IDA\* (с использованием только суммарного расстояния).

### Листинг 12.4. Реализация алгоритма IDA\*

```
% idastar(Start, Solution):
% выполнение поиска по алгоритму IDA*; Start - это начальный узел,
% Solution - путь решения

idastar(Start, Solution) :-
 retract(next_bound(_)), fail % Удалить из базы данных значение
 % следующего предела
;
 asserta(next_bound(0)), % Инициализировать значение предела
 idastar0(Start, Solution).

idastar0(Start, Sol) :-
 retract(next_bound(Bound)), % Текущий предел
 asserta(next_bound(99999)), % Инициализировать значение следующего предела
 f(Start, F), % f-значение начального узла
 df([Start], F, Bound, Sol) % Найти решение; в случае неудачи
 % изменить значение предела
;
 next_bound(NextBound),
 NextBound < 99999, % Значение предела является конечным
 idastar0(Start, Sol). % Предпринять попытку с новым значением предела

% df(Path, F, Bound, Sol):
% выполнить поиск в глубину в пределах Bound. Здесь Path - путь от начального
% до текущего узла (представленный в виде списка узлов в обратном порядке),
% F представляет собой f-значение текущего узла, т.е. головы списка Path

df([N | Ns], F, Bound, [N | Ns]) :-
 F <= Bound, % Успешное завершение - решение найдено
 goal(N).
```



```

df([N | Ns], F, Bound, Sol) :-
 F <= Bound, % f-значение узла N не выходит за пределы
 s(N, N1), not member(N1, Ns), % Развернуть N
 f(N1, F1),
 df([N1,N | Ns], F1, Bound, Sol).

df(_, F, Bound, _) :-
 F > Bound, % За пределами Bound
 update_next_bound(F), % Только обновить следующее значение предела
 fail. % и инициировать неудачное завершение

update_next_bound(F) :-
 next_bound(Bound),
 Bound =< F, ! % Не изменять следующее значение предела
 ;
 retract(next_bound(Bound)), !, % Более низкое следующее значение предела
 asserta(next_bound(F)).

```

---

### 12.4.3. Метод RBFS

Метод IDA\* основан на продуктивной идее и является очень удобным для реализации, но в неблагоприятных ситуациях связанные с ним издержки повторного формирования узлов становятся неприемлемыми. Поэтому применяется лучший, хотя и более сложный метод экономии пространства, получивший название RBFS (Recursive Best-First Search — рекурсивный поиск по заданному критерию). Реализация метода RBFS весьма напоминает программу A\*, приведенную в листинге 12.1 (которая также является рекурсивной, в том же смысле, что и RBFS!). Различие между программами A\* и RBFS состоит в том, что первая обеспечивает хранение в памяти всех ранее сформированных узлов, а последняя предусматривает хранение только текущего пути поиска одноранговых узлов вдоль этого пути. Если программа RBFS на время приостанавливает поиск в поддереве (поскольку оно больше не соответствует заданному критерию), то удаляет из памяти это поддерево для экономии пространства. Поэтому потребность в ресурсах пространства для программы RBFS (как и при использовании метода IDA\*) зависит от глубины поиска только линейно. При использовании метода RBFS единственное, что сохраняется в памяти об удаленном поддереве, — обновленное  $f$ -значение для корня этого поддерева. Такие  $f$ -значения обновляются по методу сопровождения резервных копий  $f$ -значений таким же образом, как и в программе A\*. Чтобы подчеркнуть различия между “статической” функцией оценки  $f$  и резервными копиями ее значений, используются приведенные ниже форматы записей (для узла  $N$ ).

- $f(N)$ . Значение узла  $N$ , возвращенное функцией оценки (всегда остается одним и тем же во время поиска).
  - $F(N)$ . Резервная копия  $f$ -значения (изменяется во время поиска, поскольку зависит от узлов-потомков узла  $N$ ).
- Значение  $F(N)$  определяется следующим образом.
- $F(N) = f(N)$ , если узел  $N$  (никогда) не развертывался в процессе поиска.
  - $F(N) = \min\{ F(N_i) \mid N_i - \text{дочерний узел узла } N \}$  в противном случае.

Как и программа A\*, программа RBFS также предусматривает исследование поддеревьев, соответствующих указанному  $f$ -пределу. Этот предел определяется по  $F$ -значениям одноуровневых узлов вдоль текущего пути поиска (по минимальному  $F$ -значению для одноуровневых узлов, т.е. по  $F$ -значению узла, который является ближайшим конкурентом текущего узла). Предположим, что узел  $N$  в настоящее время является наилучшим узлом в дереве поиска (т.е. имеет минимальное  $F$ -значение). В таком случае узел  $N$  развертывается и дочерние узлы узла  $N$  исследуются вплоть до некоторого  $f$ -предела  $\text{Bound}$ . При превышении этого предела (что обнаруживается с помощью выражения  $F(N) > \text{Bound}$ ) все узлы, сформированные на

уровне ниже  $N$ , удаляются из памяти. Но обновленное значение  $F(N)$  сохраняется и используется при определении того, как продолжать поиск.

В программе  $F$ -значения могут быть не только определены путем создания резервных копий значений, полученных от дочерних узлов рассматриваемого узла, но и унаследованы от родительских узлов этого узла. Такое наследование происходит следующим образом. Предположим, что имеется узел  $N$ , для которого настало время быть развернутым в процессе поиска. Если  $F(N) > f(N)$ , то известно, что узел  $N$  уже был развернут ранее и значение  $F(N)$  определено с помощью дочерних узлов узла  $N$ , но эти дочерние узлы уже удалены из памяти. Теперь предположим, что дочерний узел  $N_i$  узла  $N$  снова сформирован и статическое значение  $f(N_i)$  для узла  $N_i$  также снова вычислено. Теперь значение  $F(N_i)$  определяется следующим образом:

если  $f(N_i) < F(N)$ , то  $F(N_i) = F(N)$ , иначе  $F(N_i) = f(N_i)$

Это выражение может быть записано короче следующим образом:

$$F(N_i) = \max\{F(N), f(N_i)\}$$

Таким образом, в том случае, если  $f(N_i) < f(N)$ , то  $F$ -значение узла  $N_i$  фактически наследуется от родительского узла  $N$  узла  $N_i$ . Это можно показать с помощью следующих рассуждений: когда узел  $N_i$  был сформирован (и удален) ранее, значение  $F(N_i)$  обязательно было больше или равно  $F(N)$ . В противном случае, согласно правилу создания резервной копии, значение  $F(N)$  должно было быть меньше.

Для иллюстрации процесса функционирования программы RBFS рассмотрим задачу поиска маршрута (см. рис. 12.2). На рис. 12.7 приведены отдельные снимки текущего пути (включая одноуровневые узлы вдоль этого пути), хранящиеся в памяти. При поиске происходит переключение (как и в программе  $A^*$ ) между альтернативными путями. Но после каждого такого переключения предыдущий путь удаляется из памяти для экономии пространства. На рис. 12.7 числа, стоящие рядом с узлами, представляют собой  $F$ -значения этих узлов. На снимке  $A$  узел  $a$  является наилучшим из возможных узлов ( $F(a) < F(e)$ ). Поэтому происходит поиск в поддереве ниже узла  $a$  со значением  $\text{Bound} = 9$  (т.е. со значением, равным  $F(e)$ ; им характеризуется ближайший и единственный конкурент). После достижения в этом поиске узла  $c$  (снимок  $B$ ) обнаруживается, что  $F(c) = 10 > \text{Bound}$ . Поэтому поиск по этому пути (на время) прекращается, узлы  $c$  и  $b$  удаляются из памяти, значение  $F(c) = 10$  сохраняется в виде резервной копии в узле  $a$  и значение  $F(a)$  становится равным 10 (снимок  $B$ ). Теперь узел  $e$  становится наилучшим конкурентом ( $F(e) = 9 < 10 = F(a)$ ) и происходит поиск в его поддереве со значением  $\text{Bound} = 10 = F(a)$ . Этот поиск останавливается на узле  $f$ , поскольку  $F(f) = 11 > \text{Bound}$  (снимок  $\Gamma$ ). Узел  $e$  удаляется, и  $F(e)$  принимает значение 11 (снимок  $D$ ). Теперь поиск снова переключается на узел  $a$  со значением  $\text{Bound} = 11$ . После повторного формирования узла  $b$  обнаруживается, что  $f(b) = 8 < F(a)$ . Поэтому узел  $b$  наследует свое  $F$ -значение от узла  $a$  таким образом, что  $F(b)$  становится равным 10. Затем повторно формируется узел  $c$  и также наследует свое  $F$ -значение от  $b$ , поэтому  $F(c)$  становится равным 10. Значение  $\text{Bound} = 11$  превышает на снимке  $E$ , узлы  $d$ ,  $c$  и  $b$  удаляются и в узле  $a$  сохраняется резервная копия значения  $F(d) = 12$  (снимок  $Ж$ ). Теперь поиск переключается на узел  $e$  и беспрепятственно продолжается до цели  $t$ .

Ниже приведено более формальное определение алгоритма RBFS. В основе этого алгоритма лежит принцип обновления  $F$ -значений узлов. Поэтому удобный способ формулировки данного алгоритма состоит в определении следующей функции:

$\text{NewF}(N, F(N), \text{Bound})$

где  $N$  — узел, текущее  $F$ -значение которого равно  $F(N)$ . Функция выполняет поиск в пределах  $\text{Bound}$ , начиная с узла  $N$ , и вычисляет новое  $F$ -значение узла  $N$ ,  $\text{NewF}$ , полученное в результате этого поиска. Новое значение определяется в момент превышения предела. Но если цель удастся найти еще до этого, то функция прекращает свою работу, сообщая об успехе как о побочном результате своей работы. Функция  $\text{NewF}$  определена в листинге 12.5.

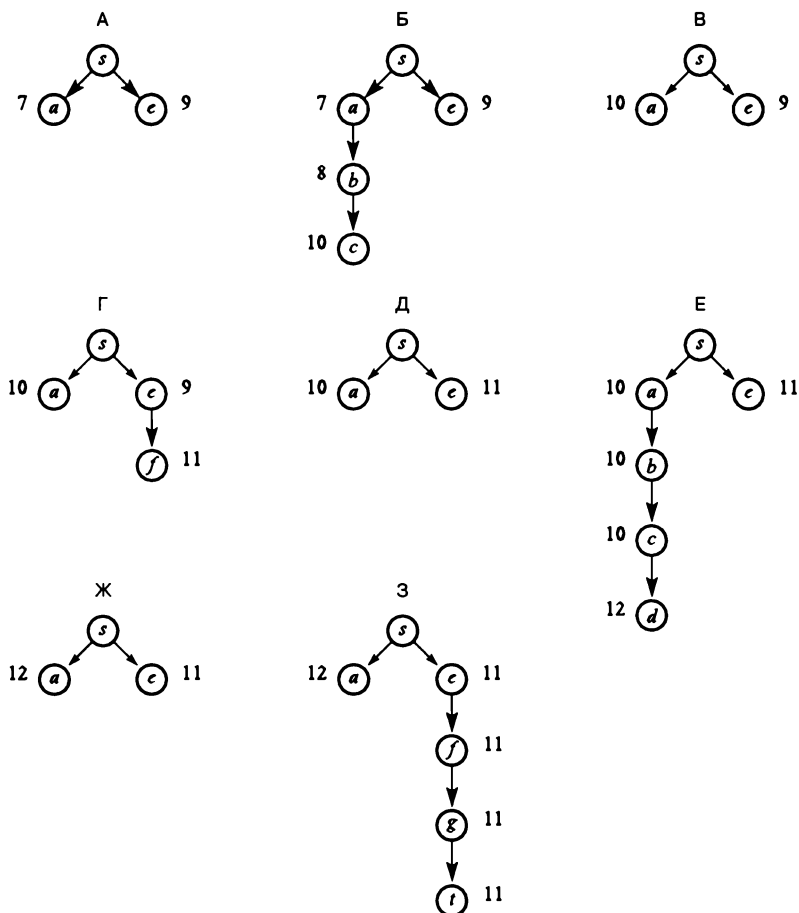


Рис. 12.7. Трассировка выполнения алгоритма RBFS в процессе решения задачи поиска маршрута, показанной на рис. 12.2; числа рядом с узлами представляют собой F-значения этих узлов (которые изменяются во время поиска)

#### Листинг 12.5. Функция обновления F-значений в процессе поиска по методу RBFS

```
function NewF(N, F(N), Bound)
begin
 if F(N) > Bound then NewF := F(N)
 else if goal(N) then успешное завершение процедуры поиска
 else if N не имеет дочерних узлов then NewF := infinity (тупиковый конец)
 else
 begin
 for каждого узла N_i , дочернего по отношению к N do
 if $f(N) < F(N_i)$ then $F(N_i) := \max(F(N), f(N_i))$
 else $F(N_i) := f(N_i)$;
 сортировать дочерние узлы N_i в порядке возрастания их F-значений;
 while $F(N_i) \leq \text{Bound}$ and $F(N_i) < \text{infinity}$ do
 begin
 Bound1 := min(Bound, F-значение наилучшего по сравнению с N_i
 однорангового узла);
```

```

 F(N1) := NewF(N1(F(N1), Bound1);
 переупорядочить узлы N1, N2, ... в соответствии
 с новым значением F(N1)
 end
 NewF := F(N1)
end
end

```

---

Реализация алгоритма RBFS на языке Prolog приведена в листинге 12.6. Эта программа аналогична программе A\*, показанной в листинге 12.1. Основой этой программы является процедура

```
rbfs(Path, SiblingNodes, Bound, NewBestFF, Solved, Solution)
```

которая реализует алгоритм RBFS. Параметры этой процедуры описаны ниже.

- Path. Найденный к этому моменту путь от начального узла поиска, представленный в виде списка узлов в обратном порядке.
- SiblingNodes. Дочерние узлы последнего узла в пути, пройденном до сих пор, т.е. голова списка Path.
- Bound. Верхний предел F-значений для развертывания поиска от узлов SiblingNodes.
- NewBestFF. Наилучшее F-значение после развертывания поиска непосредственно за пределы Bound.
- Solved. Индикатор успеха поиска на уровне ниже узлов SiblingNodes (Solved = yes, если цель найдена, Solved = no, если поиск только что вышел за пределы Bound, и Solved = never, если это направление является бесперспективным, или тупиковым).
- Solution. Путь решения, если Solved = yes, в противном случае — неконкретизированная переменная.

Представления узлов, кроме состояния в пространстве состояний, включают также стоимости пути, f- и F-значения, как показано ниже.

```
Node = (State, G/F/FF)
```

где G — стоимость пути от начального состояния до состояния State, F — статическое значение, f(State), а FF — текущее значение резервной копии F(State). Следует отметить, что переменная F в этой программе обозначает f-значение, а FF — F-значение. Процедура rbfs выполняет поиск на уровне ниже узлов SiblingNodes в пределах Bound и вычисляет значение NewBestFF в соответствии с функцией NewF, показанной в листинге 12.5.

#### Листинг 12.6. Программа поиска по заданному критерию, для которой потребность в пространстве линейно зависит от глубины поиска (алгоритм RBFS)

---

```

% Программа поиска по заданному критерию, для которой потребность в пространстве
% линейно зависит от глубины поиска, - алгоритм RBFS. В этой программе
% предполагается, что f-значение ни при каких условиях не превышает 99999

% bestfirst(Start, Solution): где Solution - путь от начального узла Start
% до целевого узла

bestfirst(Start, Solution) :-
 rbfs([], [(Start, 0/0/0)], 99999, _, yes, Solution).

% rbfs(Path, SiblingNodes, Bound, NewBestFF, Solved, Solution):
% Path - текущий путь, представленный в виде списка узлов в обратном порядке
% SiblingNodes - дочерние узлы узла в голове списка Path
% Bound - верхняя граница F-значения для поиска по узлам списка SiblingNodes

```

```

% NewBestFF - наилучшее f-значение после поиска непосредственно
% за пределами Bound
% Solved = yes, no или never
% Solution - путь решения, если Solved = yes
%
% Представление узлов: Node = (State, G/F/FF)
% где G - стоимость до наступления состояния State, F - статическое f-значение
% в состоянии State, FF - резервная копия f-значения в состоянии State

rbfs(Path, [(Node, G/F/FF) | Nodes], Bound, FF, no, _) :-
 FF > Bound, !.

rbfs(Path, [(Node, G/F/FF) | _], _, _, yes, [Node | Path]) :-
 F = FF, % Сообщать об успешном решении только один раз, после первого
 % достижения целевого узла; затем F=FF
 goal(Node).

rbfs(_, [], _, _, never, _) :- !. % Возможные продолжения отсутствуют,
 % тупиковый конец!

rbfs(Path, [(Node, G/F/FF) | Ns], Bound, NewFF, Solved, Sol) :-
 FF <= Bound, % Значение в пределах Bound - сформировать дочерние узлы
 findall(Child/Cost,
 (s(Node, Child, Cost), not member(Child, Path)),
 Children),
 inherit(F, FF, InheritedFF), % Дочерние узлы могут наследовать значение FF
 succlist(G, InheritedFF, Children, SuccNodes), % Переупорядочить эти узлы
 bestff(Ns, NextBestFF), % Ближайший конкурент по значению FF среди
 % одноранговых узлов
 min(Bound, NextBestFF, Bound2), !,
 rbfs([Node | Path], SuccNodes, Bound2, NewFF2, Solved2, Sol),
 continue(Path, [(Node,G/F/NewFF2)|Ns], Bound, NewFF, Solved2, Solved, Sol).

% continue(Path, Nodes, Bound, NewFF, ChildSolved, Solved, Solution)

continue(Path, [N | Ns], Bound, NewFF, never, Solved, Sol) :- !,
 rbfs(Path, Ns, Bound, NewFF, Solved, Sol). % Узел N - это тупиковый конец

continue(_, _, _, _, yes, yes, Sol).

continue(Path, [N | Ns], Bound, NewFF, no, Solved, Sol) :-
 insert(N, Ns, NewNs), !, % Соблюдать упорядочение одноранговых узлов
 % по значениям
 rbfs(Path, NewNs, Bound, NewFF, Solved, Sol).

succlist(_, _, [], []).

succlist(G0, InheritedFF, [Node/C | NCs], Nodes) :-
 G is G0 + C,
 h(Node, H),
 F is G + H,
 max(F, InheritedFF, FF),
 succlist(G0, InheritedFF, NCs, Nodes2),
 insert((Node, G/F/FF), Nodes2, Nodes).

inherit(F, FF, FF) :- % Дочерний узел наследует FF-значение родительского
 FF > F, !. % узла, если FF-значение родительского узла больше
 % F-значения родительского узла

inherit(F, FF, 0).

insert((N, G/F/FF), Nodes, [(N, G/F/FF) | Nodes]) :-
 bestff(Nodes, FF2),
 FF <= FF2, !.

```

```
insert(N, [N1 | Ns], [N1 | Ns1]) :-
 insert(N, Ns, Ns1).
```

```
bestff([(N, F/G/FF) | Ns], FF). % Первый узел - наилучшее FF-значение
```

```
bestff([], 99999). % Узлы отсутствуют - FF-значение равно
 % "бесконечно большому числу"
```

Еще раз кратко опишем наиболее важные свойства алгоритма RBFS. Во-первых, он характеризуется тем, что потребность в ресурсах пространства зависит от глубины поиска линейно. За это приходится платить увеличением затрат времени на повторное формирование ранее сформированных узлов. Но эти издержки значительно ниже по сравнению с алгоритмом IDA\*. Во-вторых, аналогично A\* и в отличие от IDA\*, в алгоритме RBFS предусмотрено разворачивание узлов с упорядочением по заданному критерию даже в случае немоной функции  $f$ .

## Упражнения

- 12.8. Рассмотрим задачу поиска маршрута, показанную на рис. 12.2. Сколько узлов формируется при решении этой задачи с помощью алгоритмов A\*, IDA\* и RBFS (с учетом также повторно формируемых узлов)?
- 12.9. Рассмотрим пространство состояний, показанное на рис. 12.8. Допустим, что  $a$  — начальный узел,  $l$  — целевой узел. Укажите порядок, в котором формировались узлы (включая случаи повторного формирования) с помощью алгоритма RBFS. Как изменялись резервные копии значений  $F(b)$  и  $F(c)$  в течение этого процесса?

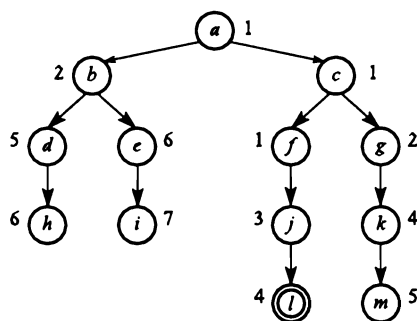


Рис. 12.8. Пространство состояний; числа рядом с узлами представляют собой  $f$ -значения узлов;  $l$  — целевой узел

- 12.10. Наследование  $F$ -значений в алгоритме RBFS позволяет сэкономить время (исключить ненужное повторное формирование узлов). Объясните, почему. Подсказка: рассмотрите процесс поиска по алгоритму RBFS в бинарном дереве, в котором  $f$ -значение каждого узла равно глубине этого узла в дереве. Проанализируйте, как происходит выполнение алгоритма RBFS с наследованием и без наследования  $F$ -значений в этом пространстве состояний вплоть до глубины 8.
- 12.11. Сравните программы A\*, IDA\* и RBFS для решения задач с различными условиями головоломки “игра в восемь”. Измерьте значения времени выполнения и определите количество узлов, формируемых во время поиска, включая повторно формируемые узлы (введите в программы счетчики узлов).

- *Эвристическая информация* может использоваться для оценки того, насколько далеко находится некоторый узел от ближайшего целевого узла в пространстве состояний. В этой главе рассматривалось применение числовых эвристических оценок.
- Принцип *эвристического поиска по заданному критерию* позволяет направлять процесс поиска таким образом, чтобы всегда разворачивался тот узел, который в настоящее время является наиболее перспективным согласно эвристическим оценкам. В данной главе приведена программа реализации широко известного алгоритма  $A^*$ , в котором используется этот принцип.
- Для того чтобы можно было воспользоваться алгоритмом  $A^*$  при решении конкретной задачи, необходимо определить пространство состояний, целевой предикат и эвристическую функцию. При решении сложных задач труднее всего найти приемлемую эвристическую функцию.
- *Теорема допустимости* позволяет определить, всегда ли алгоритм  $A^*$ , в котором используется конкретная эвристическая функция, находит оптимальное решение.
- Потребности алгоритма  $A^*$  в ресурсах времени и пространства обычно находятся в экспоненциальной зависимости от длины решения. В практических приложениях ресурсы пространства чаще всего являются более важными, чем ресурсы времени. Для экономии пространства за счет времени предназначены специальные методы поиска по заданному критерию.
- $IDA^*$  — это простой алгоритм поиска по заданному критерию, обеспечивающий экономию пространства, который основан на идее, аналогичной итеративному углублению. Издержки, связанные с повторным формированием узлов при использовании алгоритма  $IDA^*$ , являются приемлемыми в тех случаях, если многие узлы в пространстве состояний имеют одинаковые  $f$ -значения. А если узлы, как правило, не имеют одинаковых  $f$ -значений, издержки становятся неприемлемыми.
- $RBFS$  — это более сложный алгоритм поиска по заданному критерию, способствующий экономии пространства, который обычно требует повторного формирования меньшего количества узлов, чем  $IDA^*$ .
- Требования к пространству алгоритмов  $IDA^*$  и  $RBFS$  являются весьма умеренными. Они возрастают с глубиной поиска лишь линейно.
- В этой главе поиск по заданному критерию применялся в задаче решения головоломки “игра в восемь” и в задаче планирования заданий.
- В данной главе рассматривались следующие понятия:
  - эвристические оценки;
  - эвристический поиск;
  - поиск по заданному критерию;
  - алгоритмы  $A^*$ ,  $IDA^*$ ,  $RBFS$ ;
  - допустимость алгоритмов поиска, теоремы допустимости;
  - пространственная эффективность поиска по заданному критерию;
  - монотонность функции оценки.

## Дополнительные источники информации

Программа поиска по заданному критерию, приведенная в этой главе, является одним из вариантов многих аналогичных алгоритмов, из которых алгоритм  $A^*$  явля-

ется наиболее популярным. Описания алгоритма  $A^*$  можно найти в книгах, посвященных искусственному интеллекту, таких как [116], [133], [171]. В [44] впервые был описан поиск по заданному критерию, направляемый с помощью оценки расстояния до цели. Теорема допустимости была сформулирована в [63]. В литературе свойство  $h \leq h^*$  часто включается в определение  $A^*$ . Превосходное и строгое исследование многих вариантов алгоритмов поиска по заданному критерию и соответствующих математических результатов приведено в [118]. В [70] содержится отредактированный сборник интересных статей, касающихся различных сложных аспектов поиска. Одно из первых описаний метода  $IDA^*$  приведено в [78]. Алгоритм RBFS был предложен и проанализирован в [79]. В [133] обсуждаются многие другие идеи поиска в условиях ограниченных ресурсов пространства.

Головоломка “игра в восемь” использовалась в проблематике искусственного интеллекта как контрольная задача для изучения эвристических принципов многими исследователями, например в [44], [56] и [102].

Приведенная в этой главе задача планирования заданий и ее варианты сформулированы по условиям многочисленных приложений, в которых требовалось обеспечить планирование процесса обслуживания запросов к ресурсам. Приведенная в качестве примера в разделе 12.3 задача планирования заданий заимствована из [34].

Проблема поиска приемлемых эвристических функций является важной и сложной, поэтому исследование эвристических функций стало одной из центральных тем искусственного интеллекта. Но существуют также некоторые ограничения на то, до какой степени можно заниматься усовершенствованием эвристических функций. На первый взгляд может показаться, что для эффективного решения любой комбинаторной проблемы достаточно найти подходящую эвристическую функцию. Но для некоторых задач (включая многие задачи планирования) отсутствуют общие эвристические функции, которые могли бы во всех случаях гарантировать и эффективность, и допустимость решения. Многие теоретические результаты, которые относятся к проблеме анализа таких ограничений, собраны в [55].



# Декомпозиция задач и графы AND/OR

*В этой главе...*

|                                                  |     |
|--------------------------------------------------|-----|
| 13.1. Представление задач в виде графов AND/OR   | 277 |
| 13.2. Примеры представлений в виде графа AND/OR  | 281 |
| 13.3. Основные процедуры поиска в графе AND/OR   | 284 |
| 13.4. Поиск в графе AND/OR по заданному критерию | 289 |

Графы AND/OR могут служить удобным средством представления тех задач, которые возможно естественным образом разложить на ряд взаимно независимых подзадач. К примерам таких задач относится поиск маршрута, проектирование, символическое интегрирование, игры, доказательство теорем и т.д. В этой главе рассматриваются программы для поиска в графах AND/OR, включая поиск по заданному критерию в графе AND/OR, управляемый эвристическими средствами.

## 13.1. Представление задач в виде графов AND/OR

В главах 11 и 12 в основном рассматривались способы решения задач, представленных с помощью пространства состояний. В соответствии с этим решение задачи сводилось к поиску пути в графе пространства состояний. Но для некоторых категорий задач более естественно подходит другое представление, в виде графа AND/OR. В основе такого представления лежит декомпозиция задачи на подзадачи. Декомпозиция на подзадачи может применяться, если подзадачи являются взаимно независимыми и поэтому могут решаться независимо друг от друга.

Проиллюстрируем сказанное выше на примере. Рассмотрим задачу поиска маршрута между двумя указанными городами на дорожной карте, показанной на рис. 13.1. На данный момент значения длины маршрутов будут игнорироваться. Эту задачу, безусловно, можно сформулировать как поиск пути в пространстве состояний. Соответствующее пространство состояний может выглядеть полностью аналогично этой карте: узлы в пространстве состояний соответствуют городам, дуги — прямым соединениям между городами, а стоимости дуг — расстояниям между городами. Но может рассматриваться и другое представление этой задачи, основанное на ее естественной декомпозиции.

На карте, приведенной на рис. 13.1, имеется также река. Предположим, что эту реку можно пересечь только с помощью двух мостов, один из которых находится в городе  $f$ , а другой — в городе  $g$ . Очевидно, что необходимо включить в разрабатываемый маршрут один из мостов, поэтому маршрут должен пройти или через  $f$ , или через  $g$ . Таким образом, имеются два описанных ниже основных варианта.

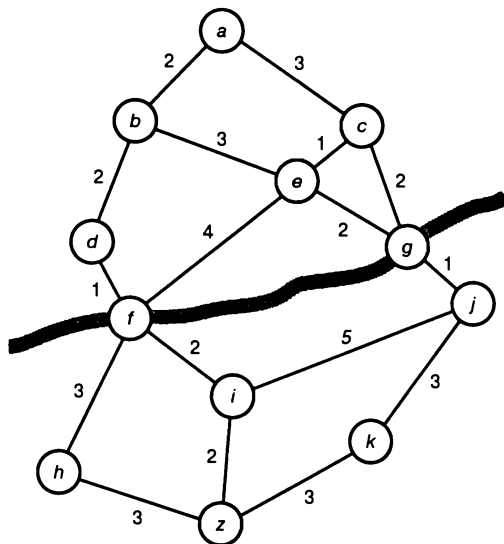


Рис. 13.1. Поиск маршрута между городами *a* и *z* по дорожной карте. Реку можно пересечь в пункте *f* или *g*. Представление этой задачи в виде графа AND/OR показано на рис. 13.2

Чтобы найти путь от *a* до *z*, необходимо выполнить одно из следующих действий.

1. Найти путь от *a* до *z* через *f*.
2. Найти путь от *a* до *z* через *g*.

Теперь каждый из этих двух вариантов задачи можно разложить на подзадачи, как описано ниже.

1. Чтобы найти путь от *a* до *z* через *f*, необходимо выполнить следующее:
  - 1.1. найти путь от *a* до *f*;
  - 1.2. найти путь от *f* до *z*.
2. Чтобы найти путь от *a* до *z* через *g*, необходимо выполнить следующее:
  - 2.1. найти путь от *a* до *g*;
  - 2.2. найти путь от *g* до *z*.

В конечном итоге имеются два основных варианта решения первоначальной задачи: проложить маршрут через *f* или проложить его через *g*. Кроме того, каждый из этих вариантов задачи можно разделить на две подзадачи (соответственно, 1.1 и 1.2 или 2.1 и 2.2). Здесь важно то, что (в обоих вариантах) каждая из подзадач может быть решена независимо от другой. Такую декомпозицию можно представить в виде графа AND/OR (рис. 13.2). Обратите внимание на две дуги, соединенные кривой линией, которые обозначают связь AND между подзадачами. Безусловно, что граф, показанный на рис. 13.2, представляет собой только верхнюю часть соответствующего дерева AND/OR. Дальнейшая декомпозиция подзадач может быть основана на рассмотрении других промежуточных городов.

Какими являются целевые узлы в подобном графе AND/OR? Целевые узлы соответствуют подзадачам, которые являются тривиальными, или “простейшими”. В данном примере такой подзадачей будет “поиск маршрута от *a* до *c*”, поскольку на дорожной карте имеется прямое соединение между городами *a* и *c*.

В этом примере введены некоторые важные понятия. Граф AND/OR представляет собой ориентированный граф, узлы которого соответствуют задачам, а дуги обозна-

чают отношения между задачами. Имеются также отношения между самими дугами. Этими отношениями являются AND и OR, в зависимости от того, требуется ли решить только одну из задач-преемников или сразу несколько (рис. 13.3). В принципе, из любого узла могут исходить и дуги, связанные отношением AND, и дуги, связанные отношением OR. Но мы предполагаем, что каждый узел имеет либо только преемников AND, либо только преемников OR. Дело в том, что каждый граф AND/OR можно преобразовать в стандартную форму, введя по мере необходимости вспомогательные узлы OR. Поэтому узел, из которого исходят только дуги AND, называется *узлом AND*, а узел, из которого исходят только дуги OR, называется *узлом OR*.

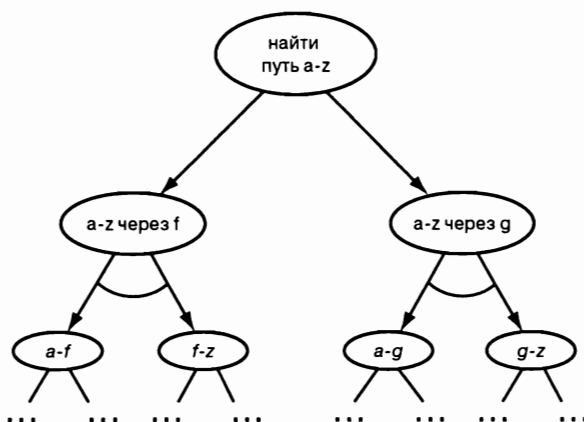


Рис. 13.2. Представление задачи поиска маршрута, приведенной на рис 13.1, в виде графа AND/OR. Узлы соответствуют задачам или подзадачам, а дуги, соединенные кривыми, показывают, что должны быть решены все (в данном случае обе) подзадачи

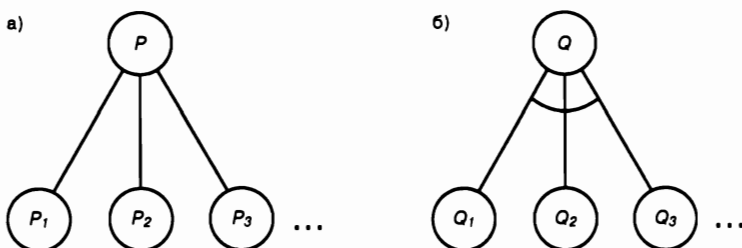


Рис. 13.3. Два типа узлов в графе AND/OR: а) чтобы решить задачу P, достаточно решить любую из подзадач P1, P2 или ...; б) чтобы решить задачу Q, необходимо решить все подзадачи, и Q1, и Q2, и ...

В представлении в виде пространства состояний решение задачи сводилось к поиску пути в пространстве состояний. Каково же решение в представлении в виде графа AND/OR? Безусловно, что любое решение должно включить все подзадачи любого узла AND. Поэтому теперь решение является не путем, а деревом. Такое дерево решения, T, определяется следующим образом.

- Первоначальная проблема, P, является корневым узлом дерева T.
- Если P — узел OR, то в дереве T находится один и только один из его преемников (по графу AND/OR) вместе с его собственным деревом решения.
- Если P — узел AND, то в дереве T находятся все его преемники (по графу AND/OR) наряду с их деревьями решения.

Это определение показано на рис. 13.4. На данном рисунке показаны также стоимости, назначенные дугам. Стоимости позволяют сформулировать критерий оптимизации. Можно, например, определить стоимость графа решения как сумму всех стоимостей дуг в этом графе. Поскольку нас обычно интересует минимальная стоимость, то предпочтительным является граф решения, показанный на рис. 13.4, в.

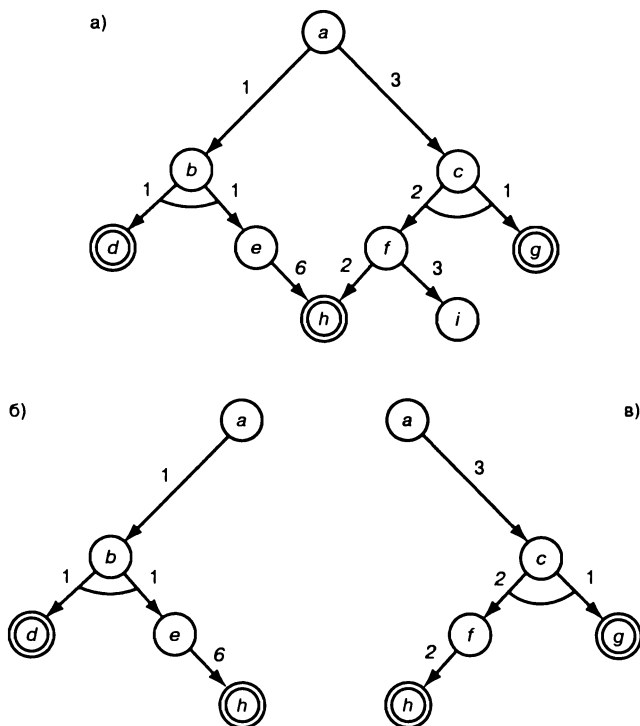


Рис. 13.4. Примеры графов решения: а) граф AND/OR, в котором  $d$ ,  $g$  и  $h$  являются целевыми узлами,  $a$  — задача, которая должна быть решена; б) и в) два дерева решения, стоимость которых составляет соответственно 9 и 8. Здесь стоимость дерева решения определена как сумма всех стоимостей дуг в дереве решения

Но мы не обязаны всегда рассматривать стоимости дуг в качестве критерия оптимизации. Иногда более естественно связать стоимости с узлами, а не с дугами или связать их и с дугами, и с узлами.

На основании сказанного выше можно сделать следующие выводы.

- Представление задачи в виде графа AND/OR основано на принципе декомпозиции задачи на подзадачи.
- Узлы в графе AND/OR соответствуют задачам, а связи между узлами указывают на отношения между задачами.
- Узел, из которого исходят связи OR, представляет собой узел OR. Для решения задачи, обозначенной узлом OR, достаточно решить одну из задач его узлов-преемников.
- Узел, из которого исходят связи AND, представляет собой узел AND. Для решения задачи, обозначенной узлом AND, необходимо решить все задачи его узлов-преемников.

- Для указанного графа AND/OR конкретная задача формулируется с помощью следующих двух понятий:
  - начальный узел;
  - целевое условие для распознавания целевых узлов.
- Целевые (или “оконечные”) узлы соответствуют тривиальным (или “простейшим”) задачам.
- Любое решение представлено в виде графа решения, который является подграфом графа AND/OR.
- Представление в виде пространства состояний может рассматриваться как частный случай представления в виде графа AND/OR, в котором все узлы являются узлами OR.
- Чтобы можно было воспользоваться преимуществами представления AND/OR, необходимо, чтобы узлы, связанные отношениями AND, представляли подзадачи, которые могут быть решены независимо друг от друга. Критерий независимости можно немного ослабить следующим образом: должно существовать упорядочение подзадач AND, такое, чтобы решения подзадач, встречающихся раньше при этом упорядочении, не уничтожались в результате решения последующих подзадач.
- Для обеспечения возможности сформулировать критерий оптимизации могут быть назначены стоимости дугам или узлам, или тем и другим.

## 13.2. Примеры представлений в виде графа AND/OR

### 13.2.1. Представление в виде графа AND/OR задачи поиска маршрута

Для задачи поиска кратчайшего маршрута (см. рис. 13.1) граф AND/OR, который включает функцию стоимости, можно определить следующим образом.

- Узлы OR имеют форму  $X-Z$ , а это означает, что нужно найти кратчайший маршрут от  $X$  до  $Z$ .
- Узлы AND имеют такую форму:  
 $X-Z \text{ via } Y$   
 а это означает — найти кратчайший маршрут от  $X$  до  $Z$ , при условии, что этот маршрут должен пройти через  $Y$ .
- Узел  $X-Z$  является целевым узлом (простейшая задача), если  $X$  и  $Z$  на карте соединены непосредственно.
- Стоимость каждого целевого узла  $X-Z$  представляет собой указанное дорожное расстояние между  $X$  и  $Z$ .
- Стоимости всех других (нетерминальных) узлов равны 0.

Стоимость графа решения представляет собой сумму стоимостей всех узлов в графе решения (в данном случае это сумма стоимостей всех терминальных узлов). Для задачи, показанной на рис. 13.1, начальным узлом является  $a-z$ . На рис. 13.5 показано дерево решения со стоимостью 9. Это дерево соответствует маршруту  $[a, b, d, f, i, z]$ , который можно реконструировать из дерева решения, посетив все листья в этом дереве в последовательности слева направо.

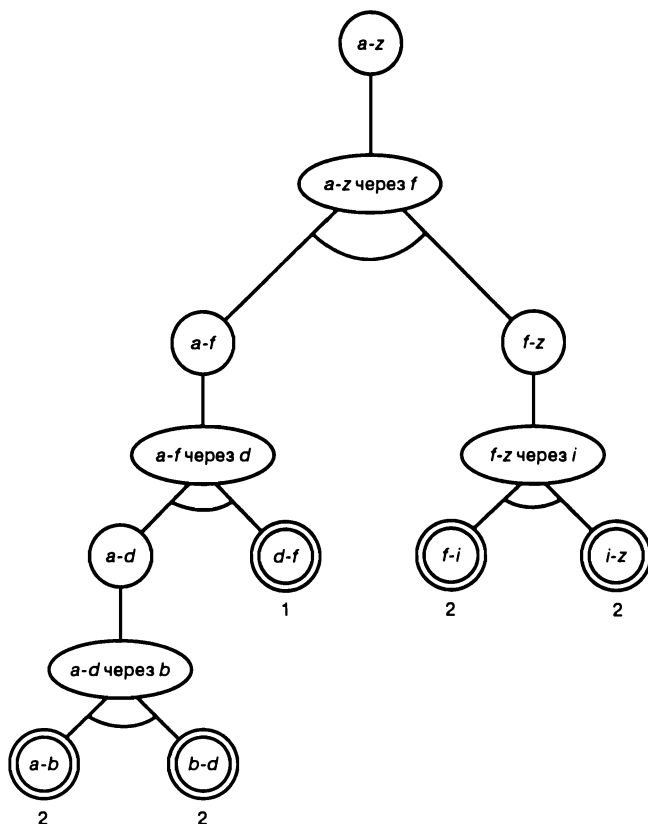


Рис. 13.5. Дерево решения с минимальной стоимостью для задачи поиска маршрута (см. рис. 13.1), оформленное в виде графа AND/OR

### 13.2.2. Задача с ханойской башней

Задача с ханойской башней (рис. 13.6) является еще одним классическим примером эффективного применения схемы декомпозиции AND/OR. Для упрощения рассмотрим описанную ниже простую версию этой задачи, содержащую только три диска.

Даны три оси, 1, 2 и 3, и три диска, а, б и с (из них а — самый маленький и с — самый большой). Первоначально все диски нанизаны на ось 1. Задача состоит в том, чтобы перенести их все на ось 3. Разрешено переносить одновременно только один диск и нельзя класть диск большего диаметра на диск меньшего диаметра.

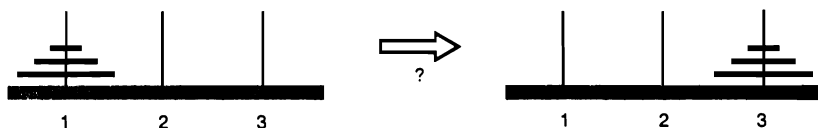


Рис. 13.6. Задача с ханойской башней

Такую задачу можно рассматривать как задачу достижения следующего множества целей.

1. Диск а на оси 3.
2. Диск b на оси 3.
3. Диск с на оси 3.

К сожалению, эти цели не являются независимыми. Например, диск а можно немедленно нанизать на ось 3, достигнув первой цели. Но это помешает достижению двух других целей (если мы не откажемся от непосредственного достижения первой цели). К счастью, имеется удобный способ упорядочения процесса достижения этих целей, такой, что решение можно легко найти на основании этого упорядочения. Требуемую последовательность достижения всех целей можно определить с помощью следующих рассуждений: цели 3 (диск с на оси 3) достичь сложнее всего, поскольку для перемещения диска с необходимо преодолеть больше всего ограничений. Хорошая идея, которая часто оказывается продуктивной в подобных ситуациях, состоит в том, что вначале следует попытаться достичь самой сложной цели. В основе этого принципа лежит наблюдение, что если другие цели не являются такими труднодостижимыми (не связаны с преодолением таких ограничений, как самая трудная), то можно надеяться, что их удастся достичь без необходимости отмены результатов достижения этой самой трудной цели.

Таким образом, стратегия решения задачи, которая вытекает из этого принципа, состоит в следующем:

вначале достичь цели "диск с на оси 3";  
затем достичь оставшихся целей.

Но первой из этих целей нельзя достичь немедленно: в начальной ситуации диск с невозможно переместить сразу же, поскольку на нем лежат еще два диска. Поэтому вначале подготовим этот ход и уточним нашу стратегию следующим образом.

1. Обеспечить перемещение с оси 1 на ось 3 диска с.
2. Перенести с оси 1 на ось 3 диск с.
3. Достижение оставшихся целей — диск а на оси 3 и диск b на оси 3.

Диск с можно перенести с оси 1 на ось 3, если оба оставшихся диска, а и b, нанизаны на ось 2. Поэтому наша первоначальная задача переноса с оси 1 на ось 3 дисков а, b и с сводится к трем перечисленным ниже подзадачам.

Чтобы переместить с оси 1 на ось 3 диски а, b и с, необходимо выполнить следующее.

1. Переместить с оси 1 на ось 2 диски а и b.
2. Переместить с оси 1 на ось 3 диск с.
3. Переместить с оси 2 на ось 3 диски а и b.

Задача 2 является тривиальной (решение состоит из одного хода). Две другие подзадачи можно решить независимо от задачи 2, поскольку диски а и b можно перенести независимо от положения диска с. Чтобы решить задачи 1 и 3, можно применить тот же принцип декомпозиции (на этот раз самой сложной является задача перемещения диска b). В соответствии с этим задача 1 сводится к трем перечисленным ниже тривиальным подзадачам.

Чтобы переместить с оси 1 на ось 2 диски а и b, необходимо выполнить следующее.

1. Переместить с оси 1 на ось 3 диск а.
2. Переместить с оси 1 на ось 2 диск b.
3. Переместить с оси 3 на ось 2 диск а.

### 13.2.3. Формулировка процесса игры в виде графа AND/OR

Такие игры, как шахматы и шашки, могут естественным образом рассматриваться как задачи, представленные в виде графов AND/OR. Подобные игры называются *играми с двумя участниками, с полной информацией*, и здесь предполагается, что в

них могут быть только два возможных исхода — победа и поражение. (Игры с тремя исходами — победа, поражение или ничья — также могут рассматриваться как имеющие только два исхода: победа и отсутствие победы.) По мере того как два игрока по очереди делают ходы, возникают позиции двух типов, в зависимости от того, кто должен ходить. Назовем двух игроков *свой* и *чужой*, поэтому два типа позиций состоят в следующем: *позиция своего хода* и *позиция чужого хода*. Предположим, что игра начинается с позиции своего хода  $P$ . Каждый вариант своего хода в этой позиции приводит к созданию одной из позиций чужого хода,  $Q_1, Q_2, \dots$  (рис. 13.7). Кроме того, каждый из вариантов чужого хода в позиции  $Q_1$  приводит к созданию одной из позиций  $R_{11}, R_{12}, \dots$ . В дереве AND/OR (рис. 13.7) узлы соответствуют позициям, а дуги — возможным ходам. Уровни своего хода чередуются с уровнями чужого хода. Чтобы выиграть в начальной позиции,  $P$ , необходимо найти ход, переводящий игру из позиции  $P$  в позицию  $Q_i$ , для некоторого  $i$ , таким образом, чтобы позиция  $Q_i$  была выигрышной. Поэтому позиция  $P$  является выигрышной, если выигрышной является позиция  $Q_1$ , или  $Q_2$ , или  $\dots$ . Итак, позиция  $P$  представляет собой узел OR. Для всех  $i$  позиция  $Q_i$  является позицией чужого хода, поэтому, если она должна быть выигрышной для своего игрока, то необходимо добиться, чтобы ее можно было выиграть после любого чужого хода. Поэтому позиция  $Q_i$  является выигрышной, если выигрышными являются все позиции, и  $R_{11}$ , и  $R_{12}$ , и  $\dots$ . В соответствии с этим все позиции чужого хода являются узлами AND. Целевые узлы представляют собой позиции, выигранные согласно правилам данной игры; например, в шахматах — это позиция с чужим королем, получившим мат. Те позиции, которые проиграны согласно правилам данной игры, соответствуют неразрешимым задачам. Чтобы решить задачу достижения выигрыша в игре, необходимо найти дерево решения, гарантирующее свою победу независимо от ответов противника. Таким образом, подобное дерево решения представляет собой полную стратегию победы в игре: на любое возможное продолжение, которое может быть выбрано противником, в этом дереве стратегии есть такой ход, который приводит к победе.

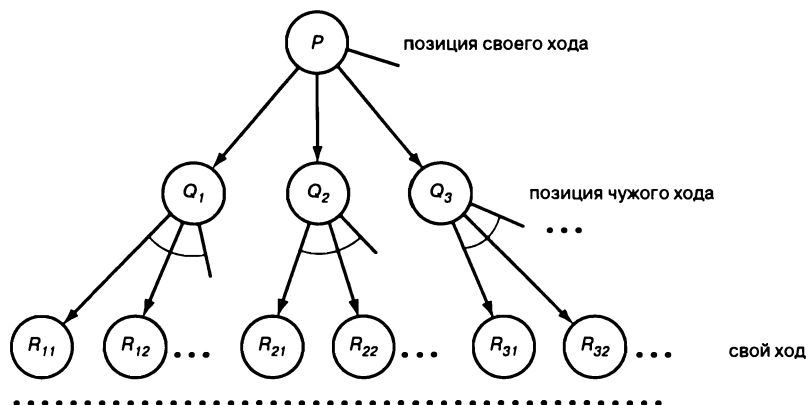


Рис. 13.7. Формулировка игры с двумя участниками в виде графа AND/OR; игроки обозначаются как свой и чужой

### 13.3. Основные процедуры поиска в графе AND/OR

В данном разделе рассматриваются только процедуры поиска хотя бы одного решения задачи, независимо от его стоимости. Поэтому для целей этого раздела можно проигнорировать стоимости связей или узлов в графе AND/OR.



Простейший способ организации поиска в графах AND/OR с помощью программы Prolog состоит в использовании собственного механизма поиска Prolog. Как оказалось, эта задача является тривиальной, поскольку по своему процедурному значению программа Prolog представляет собой ни что иное, как процедуру для поиска в графах AND/OR. Например, граф AND/OR, приведенный на рис. 13.4 (если не рассматривать стоимости дуг) может быть представлен с помощью следующих предложений:

```
a :- b. % Узел a - это узел OR с двумя преемниками, b и c
a :- c.
b :- d, e. % Узел b - это узел AND с двумя преемниками, d и e
e :- h.
c :- f, g.
f :- h, i.
d, g, h. % d, g и h - целевые узлы
```

Чтобы узнать, можно ли решить задачу а, достаточно просто задать системе следующий вопрос:

?- a.

После этого система Prolog фактически выполнит поиск в графе, приведенном на рис. 13.4, по методу поиска в глубину и ответит "yes", посетив ту часть графа поиска, которая соответствует дереву решения на рис. 13.4, б.

Преимуществом такого подхода к программированию поиска в графе AND/OR является его простота. Но этот подход имеет также перечисленные ниже недостатки.

- Он позволяет получить лишь ответ "yes" или "no", который не сопровождается также деревом решения. Дерево решения можно восстановить из трассировки программы, но такой подход становится громоздким и неэффективным, если необходимо, чтобы дерево решения было явно доступным в качестве одного из объектов в программе.
- Такую программу трудно дополнить, чтобы она позволяла также учитывать стоимости.
- Если граф AND/OR представляет собой граф общего типа, содержащий циклы, то система Prolog с ее стратегией поиска в глубину может войти в бесконечный рекурсивный цикл.

Эти недостатки будем исправлять постепенно. Вначале определим собственную процедуру поиска в глубину для графов AND/OR.

Прежде всего необходимо изменить представление графов AND/OR в программе. Для этого введем бинарное отношение, представленное в инфиксной системе обозначений с помощью оператора "--->". Например, информация о том, что узел а связан с его двумя преемниками типа OR, будет представлена с помощью следующего предложения:

```
a ---> or:[b,c].
```

Оба символа, "--->" и ":", являются инфиксными операторами, которые можно определить следующим образом:

```
:- op(600, xfx, --->).
:- op(500, xfx, :).
```

Поэтому полный граф AND/OR, показанный на рис. 13.4, можно представить с помощью таких предложений:

```
a ---> or:[b,c].
b ---> and:[d,e].
c ---> and:[f,g].
e ---> or:[h].
f ---> or:[h,i].
goal(d). goal(g). goal(h).
```

Процедуру поиска в глубину в графе AND/OR можно сформировать на основе приведенных ниже принципов.

Для поиска решения, представленного с помощью некоторого узла N, используются следующие правила.

1. Если N — целевой узел, то решение является тривиальным.
2. Если N имеет преемников OR, то найти решение с помощью одного из них (попытаться использовать их одного за другим до тех пор, пока не будет найден тот, который приведет к решению).
3. Если узел N имеет преемников AND, то найти решение для всех них (попытаться найти решения, перебирая преемников одного за другим, таким образом, чтобы решение было найдено для всех этих преемников).

Если приведенные выше правила не позволяют найти решение, то следует полагать, что задача не может быть решена.

Соответствующая программа может быть представлена следующим образом:

```
solve(Node) :-
 goal(Node).

solve(Node) :-
 Node ---> or:Nodes, % Узел Node - узел OR
 member(Node1, Nodes), % Выбрать преемника Node1 узла Node
 solve(Node1).

solve(Node) :-
 Node ---> and:Nodes, % Узел Node - узел AND
 solveall(Nodes). % Найти решения для всех преемников узла Node
 solveall([]).

solveall([Node | Nodes]) :-
 solve(Node),
 solveall(Nodes).
```

где member — обычное отношение проверки принадлежности к списку.

Тем не менее эта программа все еще имеет следующие недостатки.

- Не формирует дерево решения;
- Восприимчива к бесконечным циклам, в зависимости от свойств графа AND/OR (наличия в нем циклов).

Данную программу можно легко модифицировать таким образом, чтобы она формировала дерево решения. Для этого необходимо откорректировать отношение solve, предусмотрев в нем следующие два параметра:

```
solve(Node, SolutionTree)
```

Представим дерево решения следующим образом. Для этого могут рассматриваться три перечисленных ниже случая.

1. Если Node — целевой узел, то соответствующим деревом решения является сам узел Node.
2. Если Node — узел OR, то дерево решения имеет следующую форму:  
Node ---> Subtree  
где Subtree — дерево решения для одного из преемников узла Node.
3. Если Node — узел AND, то его дерево решения имеет форму  
Node ---> and:Subtrees  
где Subtrees — список деревьев решения всех преемников узла Node.

Например, в графе AND/OR (см. рис. 13.4) первое решение для верхнего узла a может быть представлено таким образом:

```
a ---> b ---> and:[d, e ---> h]
```

Эти три формы дерева решения соответствуют первым трем предложениям, касающимся рассматриваемого отношения `solve`. Поэтому первоначальную процедуру `solve` можно усовершенствовать, откорректировав каждое из трех предложений; это означает, что достаточно просто ввести в предложение `solve` дерево решения в качестве второго параметра. Результирующая программа показана в листинге 13.1. В этой программе имеется дополнительная процедура `show` для отображения деревьев решения. Например, дерево решения задачи (см. рис. 13.4) отображается процедурой `show` в следующей форме:

```
a ---> b ---> d
 e--->h
```

Программа в листинге 13.1 все еще способна переходить в бесконечные циклы. Один из простых способов предотвращения бесконечных циклов состоит в том, что необходимо следить за текущей глубиной поиска и запрещать программе поиск сверх некоторой указанной глубины. Это можно сделать, введя еще один параметр в отношение `solve`:

```
solve(Node, SolutionTree, MaxDepth)
```

Здесь, как и прежде, узел `Node` представляет задачу, которая должна быть решена, а `SolutionTree` — это решение, не превышающее по глубине `MaxDepth`. С другой стороны, `MaxDepth` — это допустимая глубина поиска в графе. В том случае, если `MaxDepth = 0`, дальнейшее разворачивание дерева не допускается; в противном случае, если `MaxDepth > 0`, то можно развернуть узел `Node` и попытаться достичь решения с помощью его преемников, с меньшим пределом глубины `MaxDepth - 1`. Такое условие можно легко включить в программу, приведенную в листинге 13.1. Например, второе предложение, касающееся процедуры `solve`, принимает следующий вид.

```
solve(Node, Node ---> Tree, MaxDepth) :-
 MaxDepth > 0,
 Node ---> or:Nodes, % Узел Node - это узел OR
 member(Node1, Nodes), % Выбрать преемника Node1 узла Node
 Depth1 is MaxDepth - 1, % Новый предел глубины
 solve(Node1, Tree, Depth1). % Найти решение для преемника,
 % установив меньший предел глубины
```

Эту процедуру поиска в глубину с ограничением по глубине можно также использовать в режиме итеративного углубления, моделируя тем самым поиск в ширину. Идея состоит в том, чтобы поиск в глубину выполнялся повторно, с увеличением каждый раз предела глубины до тех пор, пока не будет найдено решение. Это означает, что нужно попытаться решить проблему с использованием предела глубины 0, затем 1, 2 и т.д. Такая программа имеет следующий вид.

**Листинг 13.1. Программа поиска в глубину для графов AND/OR. Эта программа не предотвращает возникновения бесконечных циклов. Процедура `solve` находит дерево решения, а процедура `show` отображает такое дерево. При использовании процедуры `show` подразумевается, что для вывода имени каждого узла достаточно одного символа**

```
% Поиск в глубину для графов AND/OR
% solve(Node, SolutionTree):
% найти дерево решения для узла Node в графе AND/OR

:- op(500, xfx, :).
:- op(600, xfx, --->).

solve(Node, Node) :-
 goal(Node). % Деревом решения для целевого узла является
 % сам узел Node

solve(Node, Node ---> Tree) :-
 Node ---> or:Nodes, % Узел Node - это узел OR
```

```

member(Node1, Nodes), % Выбрать преемника Node1 узла Node
solve(Node1, Tree).

solve(Node, Node ---> and:Trees) :-
 Node ---> and:Nodes, % Узел Node - это узел AND
 solveall(Nodes, Trees). % Найти решения для всех преемников узла Node

% solveall([Node1,Node2, ...], [SolutionTree1,SolutionTree2, ...])

solveall([], []).

solveall([Node|Nodes], [Tree|Trees]) :-
 solve(Node, Tree),
 solveall(Nodes, Trees).

% Отображение дерева решения

show(Tree) :- % Отобразить дерево решения
 show(Tree,0), !. % с отступом 0

% show(Tree, H): отобразить дерево решения с отступом H

show(Node ---> Tree, H) :- !,
 write(Node), write(' ---> '),
 H1 is H + 7,
 show(Tree, H1).

show(and:[T], H) :- !, % Отобразить одно дерево AND
 show(T,H).

show(and:[T|Ts], H) :- !, % Отобразить список AND деревьев решения
 show(T, H),
 tab(H),
 show(and:Ts, H).

show(Node, H) :-
 write(Node), nl.

```

Как и при итеративном углублении в пространстве состояний (см. главу 11), недостаток такого моделирования поиска в ширину состоит в том, что программа повторно исследует верхнюю часть пространства состояний при каждом увеличении предела глубины. С другой стороны, важным преимуществом по сравнению с оригинальной процедурой поиска в ширину является экономия пространства.

## Упражнения

- 13.1. Доработайте программу поиска в графе AND/OR по принципу поиска в глубину с ограничением по глубине в соответствии с процедурой, описанной в данном разделе.
- 13.2. Определите на языке Prolog пространство состояний AND/OR для задачи с ханойской башней и используйте это определение с процедурами поиска, представленными в данном разделе.
- 13.3. Рассмотрите какую-либо простую игру с двумя участниками и с полной информацией, в которой не применяется жребий, и определите ее представление в виде графа AND/OR. Воспользуйтесь программой поиска в глубину в графах AND/OR, чтобы найти стратегии выигрыша в форме деревьев AND/OR.

## 13.4. Поиск в графе AND/OR по заданному критерию

### 13.4.1. Эвристические оценки и алгоритм поиска

В простых процедурах поиска, приведенных в предыдущем разделе, предусматривалось проведение в графах AND/OR систематического и исчерпывающего поиска, без каких-либо эвристических средств управления поиском. При решении сложных задач такие процедуры являются слишком неэффективными из-за комбинаторной сложности пространства поиска. Поэтому возникает необходимость в использовании эвристических средств управления поиском, которые предназначены для уменьшения этой сложности за счет исключения бесполезных вариантов. Эвристические средства управления поиском, представленные в этом разделе, основаны на числовых эвристических оценках сложности задач, представленных в виде графа AND/OR. Разрабатываемая здесь программа представляет собой реализацию алгоритма, известного под названием АО\*. Она может рассматриваться как обобщение программы поиска по заданному критерию А\*, которая была предназначена для представления пространства состояний, описанного в главе 12.

Вначале введем критерий оптимизации, основанный на стоимостях дуг в графе AND/OR. Прежде всего дополним применяемое представление графов AND/OR, чтобы в нем учитывались стоимости дуг. Например, граф AND/OR (см. рис. 13.4) может быть представлен с помощью следующих предложений:

```
a ---> or: [b/1, c/3].
b ---> and: [d/1, e/1].
c ---> and: [f/2, g/1].
e ---> or: [h/6].
f ---> or: [h/2, i/3].
goal(d). goal(g). goal(h).
```

Стоимость дерева решения должна быть определена как сумма всех стоимостей дуг в дереве. Задача оптимизации состоит в том, что нужно найти дерево решения с минимальной стоимостью. В качестве примера снова рассмотрим рис. 13.4.

Стоимость узла в графе AND/OR удобно определить как стоимость оптимального дерева решения для этого узла. После такого определения стоимость узла будет соответствовать сложности достижения данного узла.

Теперь предположим, что можно оценить стоимости узлов (не зная их деревьев решения) в графе AND/OR с помощью некоторой эвристической функции  $h$ . Такие оценки будут использоваться для управления поиском. Рассматриваемая эвристическая программа поиска должна начинать поиск с начального узла и постепенно наращивать дерево поиска, развертывая уже посещенные узлы. В этом процессе дерево будет расти даже в тех случаях, если сам граф AND/OR не представляет собой дерево; в подобных случаях граф развертывается в дерево в результате дублирования частей этого графа.

В данном процессе поиска в любой момент поиска для очередного развертывания выбирается “наиболее перспективное” возможное дерево решения. Но как при этом используется функция  $h$  для оценки того, насколько перспективным является возможное дерево решения? Или, иначе говоря, как узнать, насколько перспективным является узел (корень возможного дерева решения)?

Для узла  $N$  в дереве поиска обозначим как  $H(N)$  оценку его сложности. Для концевых узла  $N$  в текущем дереве поиска  $H(N)$  равно  $h(N)$ . С другой стороны, для внутреннего узла дерева поиска не следует использовать функцию  $h$  непосредственно, поскольку уже имеется некоторая дополнительная информация об этом узле; это означает, что уже известны его преемники. Поэтому, как показано на рис. 13.8, для внутреннего узла  $N$  типа OR сложность этого узла можно приближенно представить следующим образом:

$$H(N) = \min_i (\text{cost}(N, N_i) + H(N_i))$$

где  $\text{cost}(N, N_i)$  представляет собой стоимость дуги от  $N$  до  $N_i$ . Применение правила минимизации в этой формуле оправдано следующим фактом: чтобы найти дерево решения, которое включает узел  $N$  (вместо этого выражения будет также применяться краткая формулировка "чтобы решить узел  $N$ "), достаточно найти дерево решения для одного из его преемников.

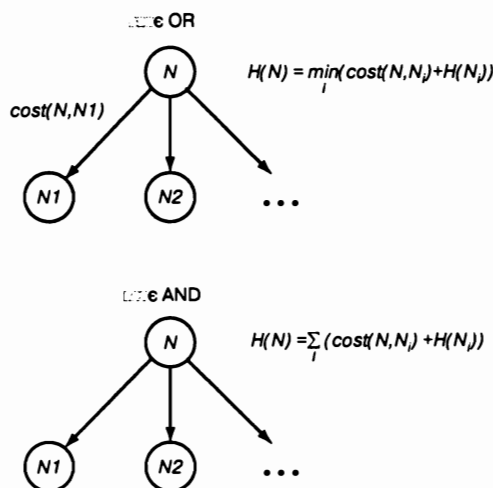


Рис. 13.8. Оценка сложности,  $H$ , задач, представленных узлами в графе AND/OR

Сложность узла  $N$  типа AND можно приближенно представить следующим образом:

$$H(N) = \sum_i (\text{cost}(N, N_i) + H(N_i))$$

В дальнейшем  $H$ -значение внутреннего узла будем также называть *резервной копией* его оценки.

В рассматриваемой программе поиска целесообразно использовать вместо  $H$ -значений другой критерий,  $F$ , определенный в терминах  $H$  следующим образом. Допустим, что узел  $M$  — предшественник узла  $N$  в дереве поиска, а стоимость дуги от  $M$  до  $N$  выражается как  $\text{cost}(M, N)$ ; в таком случае можно определить следующее:

$$F(N) = \text{cost}(M, N) + H(N)$$

В соответствии с этим, если  $M$  — родительский узел узла  $N$ , а  $N_1, N_2, \dots$  — дочерние узлы узла  $N$ , то имеет место следующее:

$$F(N) = \text{cost}(M, N) + \min_i F(N_i), \text{ если } N - \text{узел OR}$$

$$F(N) = \text{cost}(M, N) + \sum_i F(N_i), \text{ если } N - \text{узел AND}$$

Начальный узел поиска  $S$  не имеет предшественника, но предположим, что он имеет (воображаемую) входящую дугу со стоимостью 0. Итак, если значение  $h$  для всех целевых узлов в графе AND/OR равно 0 и найдено оптимальное дерево решения, то  $F(S)$  — это стоимость такого дерева решения (иными словами, сумма всех стоимостей его дуг).

На любом этапе поиска один из возможных вариантов поддеревья решения представляет любой преемник узла OR. В процессе поиска должно всегда приниматься решение о продолжении исследования графа с того преемника, для которого  $F$ -значение является минимальным. Снова вернемся к рис. 13.4 и проследим за подобным процессом поиска на примере поиска в графе AND/OR. Первоначально дерево поиска состоит только

из начального узла  $a$ , после чего дерево растет до тех пор, пока не будет найдено дерево решения. На рис. 13.9 показаны некоторые снимки, полученные в процессе роста этого дерева поиска. Для упрощения будем предполагать, что  $h = 0$  для всех узлов. Числа, стоящие рядом с узлами на рис. 13.9, представляют собой  $F$ -значения этих узлов (безусловно, они изменяются в ходе поиска по мере накопления дополнительной информации). Ниже приведены некоторые пояснения к рис. 13.9.

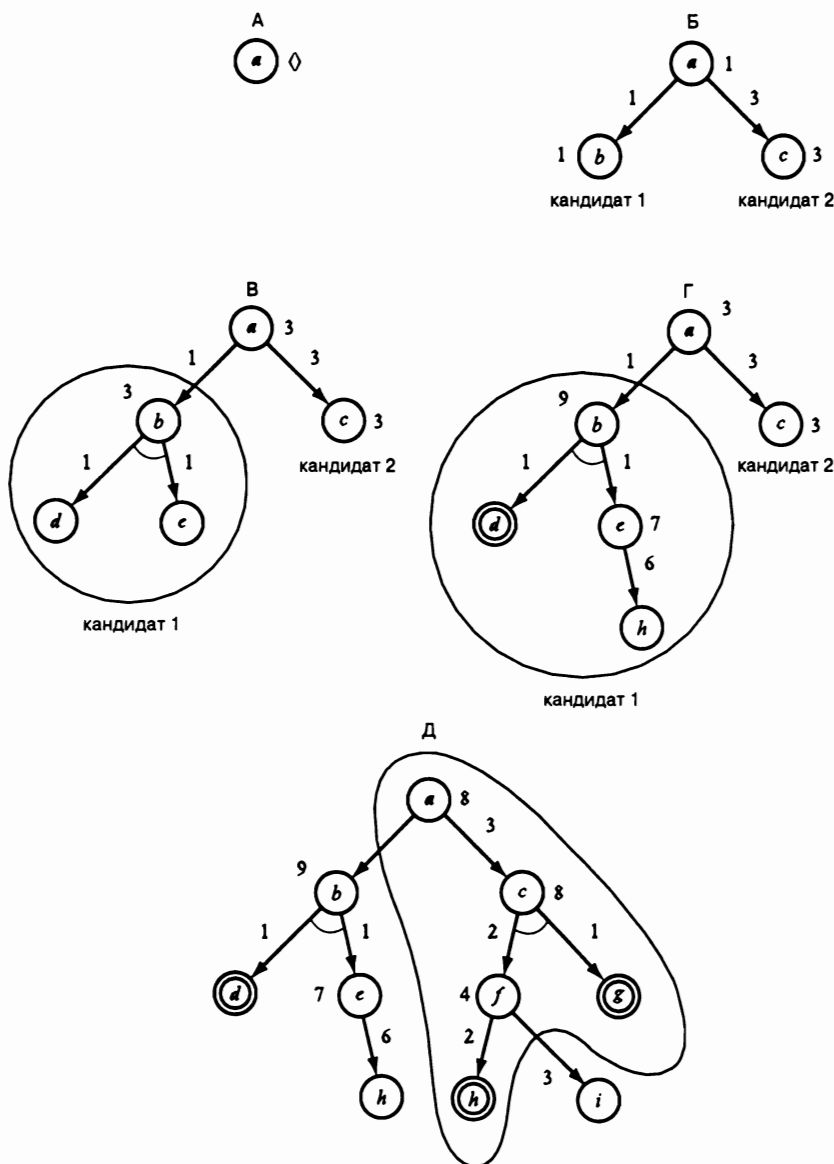


Рис. 13.9. Трассировка поиска в графе AND/OR по заданному критерию (с использованием  $h = 0$ ) в процессе решения задачи, приведенной на рис. 13.4

В результате развертывания первоначального дерева поиска (снимок А) формируется дерево, показанное на снимке Б. Узел а является узлом OR, поэтому теперь имеются два возможных варианта поддеревья решения — б и с. Поскольку  $F(b) = 1 < 3 = F(c)$ , то для развертывания выбирается вариант б. Возникает вопрос, насколько далеко может быть развернут вариант б. Развертывание может продолжаться до тех пор, пока не возникнет одна из следующих ситуаций.

1. F-значение узла б станет больше, чем у его конкурента, узла с.
2. Станет очевидно, что дерево решения найдено.

Итак, возможное поддерево решения б начинает расти в пределах верхней границы для  $F(b)$ , которая составляет  $F(b) \leq 3 = F(c)$ . Вначале формируются приемники узла б, узлы d и e (снимок В), и F-значение узла б возрастает до 3.

Поскольку это значение не превышает верхнего предела, возможное дерево решения, корнем которого является узел б, продолжает расширяться. Обнаруживается, что d — целевой узел, а затем развертывается узел e, что приводит к получению снимка Г. В этот момент  $F(b) = 9 > 3$ , и это вызывает прекращение развертывания варианта б. Таким образом, в данном процессе исключается возможность определить, что h также является целевым узлом и что дерево решения уже сформировано. Вместо этого активность теперь переключается на конкурирующий вариант с. Предельное значение  $F(c)$  для развертывания этого варианта теперь устанавливается равным 9, поскольку в данный момент  $F(b) = 9$ . В этих пределах возможное дерево решения с корнем в узле с развертывается до тех пор, пока не будет достигнута ситуация, показанная на снимке Д. Теперь в этом процессе обнаруживается, что найдено дерево решения (которое включает целевые узлы h и g) и весь процесс поиска заканчивается. Обратите внимание на то, что в результате данного процесса получено сообщение о дереве решения с наименьшей стоимостью из двух возможных, т.е. о дереве решения, показанном на рис. 13.4, в.

## 13.4.2. Программа поиска

Программа, в которой реализованы идеи, представленные в предыдущем разделе, показана в листинге 13.2. Прежде чем перейти к изложению некоторых дополнительных сведений об этой программе, рассмотрим, какое представление дерева поиска в ней используется.

Как показано на рис. 13.10, может быть несколько случаев. Дерево поиска может принимать различные формы в результате сочетания следующих факторов, от которых зависят *размер дерева* и *состояние решения*, как описано ниже.

### ■ Размер дерева:

- дерево может представлять собой либо дерево с одним узлом (листом)
- либо иметь корень и (непустые) поддерева.

### ■ Состояние решения:

- уже было обнаружено, что дерево содержит решение (дерево представляет собой дерево решения),
- или оно все еще представляет собой *возможное* дерево решения.

Основной функтор, применяемый для представления дерева, обозначает сочетание всех этих возможностей. Он может представлять собой одно из следующих:

```
leaf solvedleaf tree solvedtree
```

Кроме того, используемое представление состоит из некоторой или всей следующей информации.

- Корневой узел дерева.
- F-значение дерева.
- Стоимость С дуги в графе AND/OR, указывающей на это дерево.



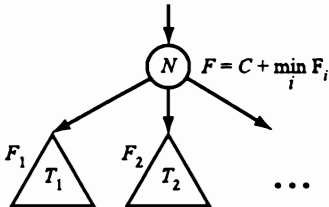
- Список поддеревьев.
- Отношение между поддеревьями (AND или OR).

Случай 1. Лист дерева поиска



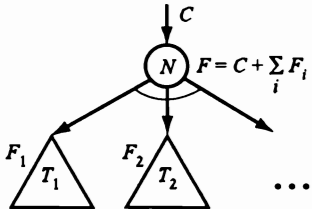
**leaf( N, F, C)**

Случай 2. Дерево поиска с поддеревьями OR



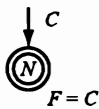
**tree( N, F, C, or:[T1,T2,...])**

Случай 3. Дерево поиска с поддеревьями AND



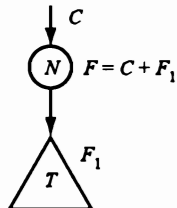
**tree( N, F, C, and:[T1,T2,...])**

Случай 4. Лист дерева решения



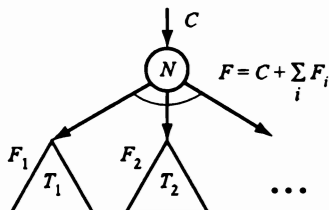
**solvedleaf( N, F)**

Случай 5. Дерево решения с корнем в узле OR



**solvedtree( N, F, T)**

Случай 6. Дерево решения с корнем в узле AND



**solvedtree( N, F, and:[T1,T2,...])**

Рис. 13.10. Представление дерева поиска

Список поддеревьев всегда упорядочен в соответствии с возрастающими  $F$ -значениями. Одно из поддеревьев уже может быть решено. Такие поддеревья находятся в конце списка. Теперь перейдем к описанию программы, приведенной в листинге 13.2. Она имеет следующее отношение верхнего уровня:

```
andor(Node, SolutionTree)
```

где  $Node$  — начальный узел поиска. Эта программа формирует дерево решения (если оно существует), такое, что можно надеяться, что это решение — оптимальное. Действительно ли такое решение имеет минимальную стоимость, зависит от эвристической функции  $h$ , используемой в данном алгоритме. Существует теорема, в которой рассматривается такая зависимость от  $h$ . Эта теорема аналогична теореме допустимости, касающейся использования пространства состояний при поиске по заданному критерию, который рассматривался в главе 12 (алгоритм  $A^*$ ). Предположим, что  $COST(N)$  обозначает стоимость дерева решения узла  $N$  с минимальной стоимостью. Если для каждого узла  $N$  в графе AND/OR эвристическая оценка  $h(N) \leq COST(N)$ , то отношение `andor` гарантирует нахождение оптимального решения. А если функция  $h$  не соответствует этому условию, то найденное решение может оказаться неоптимальным. Тривиальной эвристической функцией, которая удовлетворяет данному условию допустимости, является  $h = 0$  для всех узлов. Безусловно, что недостаток такой функции состоит в отсутствии эвристического потенциала.

### Листинг 13.2. Программа поиска в графе AND/OR по заданному критерию

```
/* Поиск в графе AND/OR по заданному критерию
```

Данная программа вырабатывает только одно решение. Гарантируется, что это решение имеет наименьшую стоимость, если используемая эвристическая функция вырабатывает значения, не превышающие фактических стоимостей деревьев решения

Дерево поиска представляет собой одно из следующих отношений:

```
tree(Node, F, C, SubTrees) дерево возможных решений
```

```
leaf(Node, F, C) лист дерева поиска
```

```
solvedtree(Node, F, SubTrees) дерево решения
```

```
solvedleaf(Node, F) лист дерева решения
```

$C$  — это стоимость дуги, направленной к узлу  $Node$

$F = C + H$ , где  $H$  — эвристическая оценка оптимального поддерева решения с корнем в узле  $Node$

Список поддеревьев  $SubTrees$  всегда упорядочен таким образом:

- (1) все поддеревья с решениями находятся в конце списка;
- (2) другие поддеревья (для которых еще не найдены решения) упорядочены по возрастающим  $F$ -значениям

```
*/
:- op(500, xfx, :).
:- op(600, xfx, --->).

andor(Node, SolutionTree) :-
 expand(leaf(Node, 0, 0), 9999, SolutionTree, yes). % Предполагается, что
 % любое F-значение не превышает 9999

% Процедура expand(Tree, Bound, NewTree, Solved)
% Развертывает дерево Tree в пределах Bound, формируя дерево NewTree,
% для которого "состояние решения" определяется переменной Solved
```

```

% Случай 1. Превышен предел Bound
expand(Tree, Bound, Tree, no) :-
 f(Tree, F), F > Bound, !. % Превышен предел Bound

% Во всех остальных случаях F <= Bound

% Случай 2. Обнаружена цель
expand(leaf(Node, F, C), _, solvedleaf(Node, F), yes) :-
 goal(Node), !.

% Случай 3. Развертывание лист-узла
expand(leaf(Node, F, C), Bound, NewTree, Solved) :-
 (expandnode(Node, C, Tree1), !,
 expand(Tree1, Bound, NewTree, Solved);
 Solved = never, !). % Преемники отсутствуют, тупиковый конец

% Случай 4. Развертывание дерева
expand(tree(Node, F, C, SubTrees), Bound, NewTree, Solved) :-
 Bound1 is Bound - C,
 expandlist(SubTrees, Bound1, NewSubs, Solved1),
 continue(Solved1, Node, C, NewSubs, Bound, NewTree, Solved).

% expandlist(Trees, Bound, NewTrees, Solved)
% Развертывает список деревьев Trees в пределах Bound, формируя список
% NewTrees, для которого "состояние решения" определяется переменной Solved
expandlist(Trees, Bound, NewTrees, Solved) :-
 selecttree(Trees, Tree, OtherTrees, Bound, Bound1),
 expand(Tree, Bound1, NewTree, Solved1),
 combine(OtherTrees, NewTree, Solved1, NewTrees, Solved).

% Процедура continue принимает решение в отношении дальнейших действий
% после развертывания списка деревьев
continue(yes, Node, C, SubTrees, _, solvedtree(Node,F,SubTrees), yes) :-
 backup(SubTrees, H), F is C + H, !.

continue(never, _, _, _, _, never) :- !.

continue(no, Node, C, SubTrees, Bound, NewTree, Solved) :-
 backup(SubTrees, H), F is C + H, !,
 expand(tree(Node, F, C, SubTrees), Bound, NewTree, Solved).

% Процедура combine объединяет результаты развертывания дерева и списка деревьев
combine(or:_, Tree, yes, Tree, yes) :- !. % Найдено решение для списка OR

combine(or:Trees, Tree, no, or:NewTrees, no) :-
 insert(Tree, Trees, NewTrees), !. % Решение для списка OR еще не найдено

combine(or:[], _, never, _, never) :- !. % Узлов, для которых возможно
 % развертывание, больше нет

combine(or:Trees, _, never, or:Trees, no) :- !. % Еще есть узлы, для которых
 % возможно развертывание

combine(and:Trees, Tree, yes, and:[Tree|Trees], yes) :-
 allsolved(Trees), !. % Найдено решение для списка AND

combine(and:_, _, never, _, never) :- !. % Решение для списка AND
 % найти невозможно

```

```

combine(and:Trees, Tree, YesNo, and:NewTrees, no) :-
 insert(Tree, Trees, NewTrees), !. % Решение для списка AND еще не найдено

% Процедура expandnode формирует дерево, состоящее из узла и его приемников
expandnode(Node, C, tree(Node, F, C, Op:SubTrees)) :-
 Node ---> Op:Successors,
 evaluate(Successors, SubTrees),
 backup(Op:SubTrees, H), F is C + H.

evaluate([], []).

evaluate([Node/C|NodesCosts], Trees) :-
 h(Node, H), F is C + H,
 evaluate(NodesCosts, Trees1),
 insert(leaf(Node, F, C), Trees1, Trees).

% Процедура allsolved проверяет, для всех ли деревьев в списке деревьев
% найдено решение
allsolved([]).

allsolved([Tree|Trees]) :-
 solved(Tree),
 allsolved(Trees).

solved(solvedtree(_,_,_)).

solved(solvedleaf(_,_)).

f(Tree, F) :-
 arg(2, Tree, F), !. % Извлечь F-значение для дерева.
 % F - это второй параметр в отношении Tree

% insert(Tree, Trees, NewTrees): вставляет дерево Tree в список деревьев Trees,
% формируя список NewTrees
insert(T, [], [T]) :- !.

insert(T, [T1|Ts], [T,T1|Ts]) :-
 solved(T1), !.

insert(T, [T1|Ts], [T1|Ts1]) :-
 solved(T),
 insert(T, Ts, Ts1), !.

insert(T, [T1|Ts], [T,T1|Ts]) :-
 f(T, F), f(T1, F1), F <= F1, !.

insert(T, [T1|Ts], [T1|Ts1]) :-
 insert(T, Ts, Ts1).

% Процедура backup находит все резервные копии F-значений для списка
% деревьев AND/OR
backup(or:[Tree|_], F) :- % Первое дерево в списке OR является наилучшим
 f(Tree, F), !.

backup(and:[], 0) :- !.

backup(and:[Tree1|Trees], F) :-
 f(Tree1, F1),
 backup(and:Trees, F2),
 F is F1 + F2, !.

backup(Tree, F) :-

```

```
f(Tree, F).

% Отношение selecttree(Trees, BestTree, OtherTrees, Bound, Bound1):
% OtherTrees - это список Trees деревьев AND/OR без его лучшего члена, BestTree;
% Bound - это предел разворачивания для деревьев из списка Trees,
% Bound1 - предел разворачивания для BestTree

selecttree(Op:[Tree], Tree, Op:[], Bound, Bound1) :- !. % Единственное дерево,
 % для которого возможно разворачивание

selecttree(Op:[Tree|Trees], Tree, Op:Trees, Bound, Bound1) :-
 backup(Op:Trees, F),
 (Op = or, !, min(Bound, F, Bound1);
 Op = and, Bound1 is Bound - F).

min(A, B, A) :- A < B, !.

min(A, B, B).
```

---

Ключевым отношением в программе, приведенной в листинге 13.2, является следующее:

```
expand(Tree, Bound, Tree1, Solved)
```

где Tree и Bound — “входные” параметры, а Tree1 и Solved — “выходные”. Значения этих параметров описаны ниже.

- Tree. Дерево поиска, которое должно быть развернуто.
- Bound. Предел для F-значения, в котором разрешено разворачивание дерева Tree.
- Solved. Индикатор, значение которого указывает на один из следующих трех случаев.
  - Solved = yes. Дерево Tree может быть развернуто в заданных пределах таким образом, чтобы оно представляло собой дерево решения Tree1.
  - Solved = no. Дерево Tree может быть развернуто в дерево Tree1 таким образом, что F-значение дерева Tree1 превышает предел Bound и не найдено поддерево решения до того, как F-значение превысило предел Bound.
  - Solved = never. Дерево Tree не содержит решения.

В зависимости от описанных выше случаев, Tree1 представляет собой дерево решения, полученное в результате разворачивания дерева Tree непосредственно за пределы Bound, или неконкретизированную переменную (в случае Solved = never). Процедура

```
expandlist(Trees, Bound, Trees1, Solved)
```

аналогична процедуре expand. Как и в процедуре expand, Bound представляет собой предел разворачивания дерева, а Solved — индикатор того, что произошло во время разворачивания (“yes”, “no” или “never”). Но первым параметром этой процедуры является список деревьев (список AND или список OR), как показано ниже.

```
Trees = or:[T1, T2, ...] или Trees = and:[T1, T2, ...]
```

Процедура expandlist выбирает в списке Trees наиболее перспективное дерево T (в соответствии с F-значением). Благодаря используемому упорядочению поддеревьев это — всегда первое дерево в списке Trees. Наиболее перспективное поддерево разворачивается с новым пределом Bound1. Значение Bound1 зависит от Bound, а также от других деревьев в списке Trees. Если Trees представляет собой список OR, то Bound1 меньше Bound и равен F-значению дерева в списке Trees со следующим по порядку критерием оптимальности (F-значением). Если Trees — список AND, то Bound1 равен значению Bound за вычетом суммы F-значений остальных де-

ревьев в списке `Trees`. Состав списка `Trees1` зависит от ситуации, обозначенной индикатором `Solved`. В том случае, если `Solved = no`, список `Trees1` представляет собой список `Trees` с наиболее перспективным деревом из списка `Trees`, развернутым в пределах `Bound1`. В случае `Solved = yes`, список `Trees1` является решением списка `Trees` (найденным в пределах `Bound`). Если `Solved = never`, список `Trees1` представляет собой неконкретизированную переменную.

Процедура `continue`, которая вызывается после развертывания списка деревьев, определяет, какие действия должны выполняться в дальнейшем, в зависимости от результатов выполнения процедуры `expandlist`. Она формирует дерево решения, обновляет дерево поиска и продолжает его развертывание или же сигнализирует о ситуации “never” (в том случае, если было обнаружено, что список деревьев не содержит решения).

Еще одна процедура,

```
combine(OtherTrees, NewTree, Solved1, NewTrees, Solved)
```

объединяет в себе средства представления нескольких объектов, которые обрабатываются процедурой `expandlist`. Здесь `NewTree` — дерево, развернутое в списке деревьев процедуры `expandlist`, `OtherTrees` — остальные, неизменившиеся деревья в списке деревьев, а `Solved1` — индикатор “состояния решения” для `NewTree`. В процедуре `combine` обрабатывается несколько ситуаций, в зависимости от значения `Solved1` и от того, является ли список деревьев списком AND или списком OR. Например, в предложении

```
combine(or_, Tree, yes, Tree, yes).
```

указано, что в случае, если список деревьев представляет собой список OR, только что развернутое дерево было решено и его дерево решения — `Tree`, то найдено решение для всего списка и этим решением является само дерево `Tree`. Назначение других случаев можно проще всего понять из кода самой процедуры `combine`.

Для отображения дерева решения может быть определена процедура, аналогичная процедуре `show` (см. листинг 13.1). Разработку такой процедуры оставляем в качестве упражнения для читателя.

### 13.4.3. Пример отношений с определением задачи — поиск маршрута

Теперь сформулируем задачу поиска маршрута как поиска пути в графе AND/OR таким образом, чтобы эту формулировку можно было непосредственно использовать в процедуре `andor`, приведенной в листинге 13.2. Предположим, что дорожная карта представлена с помощью следующего отношения:

```
s(City1, City2, D)
```

Оно означает, что существует прямое соединение между городами `City1` и `City2`, имеющее длину `D`. Кроме того, предположим, что задано отношение

```
key(City1 - City2, City3)
```

Это отношение означает: чтобы найти маршрут из города `City1` в город `City2`, необходимо рассмотреть маршрут, который проходит через `City3` (`City3` — это *ключевой пункт* между `City1` и `City2`). Например, на карте, приведенной на рис. 13.1, ключевыми пунктами в маршрутах между `a` и `z` являются `f` и `g`, как показано ниже.

```
key(a-z, f). key(a-z, g).
```

В программе должны быть реализованы приведенные ниже принципы поиска маршрута. Чтобы найти маршрут между городами `X` и `Z`, необходимо выполнить следующее.

1. Если имеются ключевые пункты `Y1`, `Y2`, ..., между `X` и `Z`, то необходимо найти либо
  - маршрут от `X` до `Z` через `Y1`, либо

- маршрут от A до Z через Y2, либо

- ...

2. Если нет ключевых пунктов между X и Z, то нужно найти некоторый соседний город Y города X, такой, что есть маршрут из Y в Z.

Таким образом, нам предстоит решать задачи двух типов, которые могут быть представлены, как показано ниже.

1. X-Z.

Найти маршрут от X до Z.

2. X-Z via Y.

Найти маршрут от X до Z через Y.

В данном случае "via" — инфиксный оператор с приоритетом выше, чем у оператора "-", и ниже, чем у оператора "--->". Теперь соответствующий граф AND/OR может быть неявно определен с помощью следующего фрагмента программы:

```
:- op(560, xfx, via).
% Правило развертывания для задачи X-Z, если есть ключевые пункты
% между X и Z; стоимости всех дуг равны 0
X-Z ---> or:ProblemList
:- bagof((X-Z via Y)/0, key(X-Z, Y), ProblemList), !.
% Правило развертывания для задачи X-Z, если нет ключевых пунктов
X-Z ---> or:ProblemList
:- bagof((Y-Z)/D, s(X, Y, D), ProblemList).
% Свести задачу "via" к двум взаимосвязанным подзадачам AND
X-Z via Y ---> and:[(X-Y)/0, (Y-Z)/0].
goal(X-X). % Задача поиска маршрута из X в X является тривиальной
```

Функцию h можно определить, например, как расстояние между городами по прямой.

## Упражнения

- 13.4. Допустим, что некоторый граф AND/OR определен следующим образом:

```
a ---> or:[b/1,c/2].
b ---> and:[d/1,e/1].
c ---> and:[e/1,f/1].
d ---> or:[h/6].
e ---> or:[h/2].
f ---> or:[h/4,i/2].
goal(h). goal(i).
```

Нарисуйте все деревья решения и рассчитайте их стоимости. Предположим, что поиск в этом графе осуществляется с помощью алгоритма АО\*, где начальным узлом является a, а все эвристические h-значения узлов b, d, e, h и i равны 0. Задайте интервалы для значений h(c) и h(f), которые позволяют найти оптимальное решение с помощью алгоритма АО\*.

- 13.5. Напишите процедуру

```
show2(SolutionTree)
```

для отображения дерева решения, найденного программой andor (см. листинг 13.2). Допустим, что формат отображения является аналогичным применяемому в процедуре show (см. листинг 13.1), чтобы show2 можно было написать как модификацию show с использованием другого представления дерева. Еще одной полезной модификацией может оказаться замена цели write( Node) в процедуре show определяемой пользователем процедурой writenode( Node, H)

которая выводит узел Node в некоторой подходящей форме и конкретизирует переменную H значением количества символов, которое потребуется для выво-

да узла Node в этой форме. Таким образом, N используется для вывода обозначений поддеревьев с подходящим отступом.

## Резюме

- Графы AND/OR применяются в качестве формального способа представления задач. Они естественным образом подходят для тех задач, которые могут быть разложены на независимые подзадачи. Примерами таких задач могут служить задачи поиска выигрыша в играх.
- Узлы в графе AND/OR относятся к двум типам: AND и OR.
- Каждая конкретная задача определяется начальным узлом и целевым состоянием. Решение задачи представляется с помощью дерева решения.
- На графе AND/OR можно показать стоимости дуг и узлов для моделирования задач оптимизации.
- Решение любой задачи, представленной с помощью графа AND/OR, сводится к поиску в графе. Стратегия поиска в глубину предусматривает систематическое обследование графа и может быть легко реализована в программе. Но такая программа может характеризоваться низкой эффективностью из-за комбинаторного роста количества вариантов.
- Для обозначения сложности задач могут быть введены эвристические оценки, а для управления поиском может применяться эвристический принцип поиска по заданному критерию. Реализация такой стратегии является более трудной.
- В данной главе приведены программы Prolog для поиска в глубину, поиска в глубину с итеративным углублением и поиска по заданному критерию в графах AND/OR.
- В данной главе введены следующие понятия:
  - графы AND/OR;
  - дуги AND, дуги OR;
  - узлы AND, узлы OR;
  - граф решения, дерево решения;
  - стоимость дуги, стоимость узла;
  - эвристические оценки в графах AND/OR, резервные копии оценок;
  - поиск в глубину в графах AND/OR;
  - итеративное углубление в графах AND/OR;
  - поиск по заданному критерию в графах AND/OR.

## Дополнительные источники информации

Графы AND/OR и относящиеся к ним алгоритмы поиска входят в состав тех классических средств решения задач и нахождения выигрыша в играх, которые относятся к проблематике искусственного интеллекта. Одним из первых примеров применения таких средств является программа символического интегрирования [151]. На принципе поиска в графе AND/OR основана работа самой системы Prolog. Общее описание графов AND/OR и алгоритма поиска в графе AND/OR по заданному критерию можно найти в книгах по искусственному интеллекту [115], [116]. Приведенная в этой главе программа поиска в графе AND/OR по заданному критерию является одним из вариантов алгоритма, известного под названием AO\*. Формальные свойства алгоритма AO\* (включая его допустимость) исследовались несколькими авторами; в [118] приведен полный отчет о результатах этих исследований.



# Логическое программирование в ограничениях

*В этой главе...*

|                                                                                      |     |
|--------------------------------------------------------------------------------------|-----|
| 14.1. Удовлетворение ограничений и логическое программирование                       | 301 |
| 14.2. Применение метода CLP для обработки действительных чисел — CLP(R)              | 306 |
| 14.3. Планирование с помощью метода CLP                                              | 310 |
| 14.4. Программа моделирования в ограничениях                                         | 317 |
| 14.5. Применение метода CLP для поддержки конечных областей<br>определения — CLP(FD) | 321 |

Программирование в ограничениях — это удобный подход к формулировке и решению задач, которые могут быть естественным образом представлены в терминах ограничений, заданных в множестве переменных. Решение подобных задач заключается в поиске таких комбинаций значений переменных, которые соответствуют ограничениям. Этот процесс называется *удовлетворением ограничений*. Логическое программирование в ограничениях (Constraint Logic Programming — CLP) представляет собой сочетание подхода к решению задач в ограничениях с логическим программированием. При использовании метода CLP средства удовлетворения ограничений встраиваются в логический язык программирования, такой как Prolog. В этой главе приведены основные сведения о методе CLP и рассматриваются некоторые примеры приложений, включая планирование и моделирование.

## 14.1. Удовлетворение ограничений и логическое программирование

### 14.1.1. Удовлетворение ограничений

Проблема удовлетворения ограничений формулируется, как описано ниже.

Дано:

- 1) множество переменных;
- 2) области определения, из которых могут выбираться значения переменных;
- 3) ограничения, которым должны удовлетворять переменные.

Найти:

такие значения, присваиваемые переменным, которые удовлетворяют всем заданным ограничениям.

Часто существует несколько вариантов присваивания, удовлетворяющих ограничениям. В задачах оптимизации может быть определен критерий выбора вариантов присваивания, которые удовлетворяют ограничениям.

Как оказалось, подход, предусматривающий поиск значений переменных, удовлетворяющих ограничениям, и особенно его сочетание с логическим программированием, представляет собой инструментальное средство, которое может весьма успешно применяться для решения широкого круга задач. К типичным примерам таких задач относятся задачи планирования, снабжения и управления ресурсами на производстве, на транспорте и в складском хозяйстве. Для решения этих задач необходимо распределять ресурсы по процессам, например: автобусы по маршрутам; солдат по поездам; экипажи по самолетам; бригады по поездам; врачей и медсестер по дежурствам и сменам и т.д.

Рассмотрим типичный пример из области планирования. Предположим, что имеются четыре задания,  $a$ ,  $b$ ,  $c$  и  $d$ , продолжительности которых составляют соответственно 2, 3, 5 и 4 часа. Между этими заданиями установлены ограничения предшествования: задание  $a$  должно предшествовать заданиям  $b$  и  $c$ , а задание  $b$  должно предшествовать заданию  $d$  (рис. 14.1). Задача состоит в том, чтобы найти значения времени начала выполнения соответствующих задач  $T_a$ ,  $T_b$ ,  $T_c$  и  $T_d$  таким образом, чтобы время завершения  $T_f$  выполнения всего расписания было минимальным. Допустим, что самым ранним временем запуска является 0.

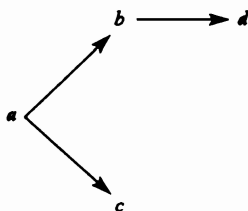


Рис. 14.1. Ограничения предшествования между заданиями  $a$ ,  $b$ ,  $c$ ,  $d$

Соответствующую задачу удовлетворения ограничений можно формально определить, как описано ниже.

Переменные:  $T_a$ ,  $T_b$ ,  $T_c$ ,  $T_d$ ,  $T_f$ .

Области определения: все переменные — неотрицательные действительные числа.

Ограничения:

$T_a + 2 \leq T_b$ . Задача  $a$ , на выполнение которой требуется 2 часа, предшествует  $b$ ;

$T_a + 2 \leq T_c$ . Задача  $a$  предшествует задаче  $c$ ;

$T_b + 3 \leq T_d$ . Задача  $b$  предшествует задаче  $d$ ;

$T_c + 5 \leq T_f$ . Задача  $c$  завершается к моменту времени  $T_f$ ;

$T_d + 4 \leq T_f$ . Задача  $d$  завершается к моменту времени  $T_f$ .

Критерий: минимизация значения  $T_f$ .

Эта задача удовлетворения ограничений имеет множество решений, причем все они позволяют обеспечить минимальное время завершения. Это множество решений можно определить следующим образом:

$T_a = 0$

$T_b = 0$

$2 \leq T_c \leq 4$

$T_d = 5$

$T_f = 9$

Определены все значения времени начала, за исключением задания  $c$ , выполнение которого может начаться в любое время в интервале от 2 до 4.

## 14.1.2. Решение задачи удовлетворения ограничений

Условия задач удовлетворения ограничений часто изображаются в виде графов, называемых *сетями ограничений*. Узлы в таком графе соответствуют переменным, а дуги — ограничениям. Для каждого бинарного ограничения  $p(X, Y)$  между переменными  $X$  и  $Y$  в этом ориентированном графе имеются две дуги,  $(X, Y)$  и  $(Y, X)$ . Для поиска решения задачи удовлетворения ограничений могут использоваться различные алгоритмы обеспечения совместимости. Эти алгоритмы лучше всего рассматривать как действующие в сетях ограничений. Они проверяют *совместимость* областей определения переменных с ограничениями. Основные принципы функционирования таких алгоритмов будут описаны ниже. Следует отметить, что в данной главе рассматриваются методы обеспечения совместимости, применяемые к бинарным ограничениям, но в общем ограничения могут связывать между собой любое количество переменных (иметь любую арность).

Рассмотрим переменные  $X$  и  $Y$ , которые имеют области определения  $D_x$  и  $D_y$ . Предположим, что между переменными  $X$  и  $Y$  задано бинарное ограничение  $p(X, Y)$ . Дуга  $(X, Y)$  называется *совместимой с определяемым ограничением*, или просто *совместимой*, если для каждого значения  $X$  в области определения  $D_x$  существует некоторое значение для  $Y$  в области определения  $D_y$ , удовлетворяющее ограничению  $p(X, Y)$ . Если  $(X, Y)$  не является совместимой, то все значения в области  $D_x$ , для которых отсутствует соответствующее значение в области  $D_y$ , могут быть удалены из  $D_x$ . В результате  $(X, Y)$  становится совместимой.

Например, рассмотрим переменные  $X$  и  $Y$ , областями определения которых являются множества всех целых чисел от 0 до 10 включительно, что можно записать следующим образом:

$$D_x = 0..10, D_y = 0..10$$

Допустим, что задано ограничение  $p(X, Y)$  следующим образом:  $X + 4 \leq Y$ . В таком случае дуга  $(X, Y)$  перестает быть совместимой. Например, для  $X = 7$  в области  $D_y$  нет значения  $Y$ , удовлетворяющего условию  $p(7, Y)$ . Для того чтобы дуга  $(X, Y)$  стала совместимой, область определения  $D_x$  необходимо сократить до  $D_x = 0..6$ . Аналогичным образом, дугу  $(Y, X)$  можно сделать совместимой, сократив  $D_y$  таким образом:  $D_y = 4..10$ . Но в результате такого сокращения областей  $D_x$  и  $D_y$  мы не теряем ни одного решения этой задачи удовлетворения ограничений, поскольку отброшенные значения, безусловно, не должны были войти в состав какого-либо решения.

После сокращения области  $D_x$  могут стать несовместимыми некоторые другие дуги. Например, для дуги, заданной как  $(Z, X)$ , могут существовать такие значения  $Z$ , для которых после сокращения  $D_x$  больше не будет ни одного соответствующего значения в области  $D_x$ . После этого, в свою очередь, можно сделать дугу  $(Z, X)$  совместимой, уменьшив соответствующим образом область  $D_z$ . Итак, эффект подобного действия может в течение определенного времени распространяться по всей сети, возможно, циклически, до тех пор, пока все дуги не станут совместимыми или некоторая область определения не станет пустой. В последнем случае, безусловно, ограничения не могут быть удовлетворены. А в случае, если все дуги являются совместимыми, могут возникать еще две ситуации.

1. Каждая область определения включает единственное значение; это означает, что данная задача удовлетворения ограничений имеет единственное решение.
2. Все области определения непусты, и по меньшей мере одна область определения содержит несколько значений.

Во второй ситуации, которая относится к тому случаю, когда все дуги являются совместимыми, безусловно, нет никакой гарантии, что все возможные сочетания значений из областей определения являются решениями задачи удовлетворения ограничений. Может даже оказаться, что фактически ни одно сочетание значений не удовлетворяет всем ограничениям. Поэтому для поиска решения необходимо выпол-

нить некоторый комбинаторный поиск по сокращенным областям определения. Одна возможность состоит в том, чтобы выбрать какую-то из многозначных областей определения и попытаться поочередно присвоить ее значения соответствующей переменной. Присваивание конкретного значения переменной равносильно сокращению области определения переменной, и такая операция может снова вызвать появление несовместимых дуг. Таким образом, алгоритм обеспечения совместимости может быть применен снова для дальнейшего сокращения областей определения переменных и т.д. Если области определения являются конечными, то такой способ действий может в конечном итоге привести к получению либо какой-либо пустой области, либо всех однозначных областей определения. Такой поиск может осуществляться поразному, а не обязательно с помощью выбора одного значения из некоторой области. Другой способ может предусматривать выбор области, состоящей из нескольких значений и разделения ее на два подмножества приблизительно одинаковых размеров. После этого алгоритм выбора отдельного значения применяется к обоим подмножествам.

В качестве иллюстрации рассмотрим, как может действовать этот алгоритм в приведенном выше примере составления расписания. Предположим, что области определения всех переменных представляют собой целые числа от 0 до 10. На рис. 14.2 показана сеть ограничений, а в табл. 14.1 приведена трассировка выполнения алгоритма удовлетворения ограничений. Первоначально, в шаге "Start", все области определения равны 0..10. В каждом шаге выполнения одна из дуг в сети становится совместимой. В шаге 1 рассматривается дуга (Tb, Ta), которая сокращает область Tb до 2..10. Затем рассматривается дуга (Td, Tb), которая сокращает область Td до 5..10, и т.д. После выполнения шага 8 все дуги становятся совместимыми и все сокращенные области являются многозначными. Поскольку мы заинтересованы в получении минимального времени завершения, теперь можно попытаться присвоить значение Tf = 9. После этого снова выполняется алгоритм обеспечения совместимости дуг, в результате чего все области определения сокращаются до однозначной, кроме области определения Tc, которая становится равной 2..4.

Таблица 14.1. Трассировка выполнения алгоритма обеспечения совместимости дуг

| Шаг   | Дуга     | Ta    | Tb    | Tc    | Td    | Tf    |
|-------|----------|-------|-------|-------|-------|-------|
| Start |          | 0..10 | 0..10 | 0..10 | 0..10 | 0..10 |
| 1     | (Tb, Ta) |       | 2..10 |       |       |       |
| 2     | (Td, Tb) |       |       |       | 5..10 |       |
| 3     | (Tf, Td) |       |       |       |       | 9..10 |
| 4     | (Td, Tf) |       |       |       | 5..6  |       |
| 5     | (Tb, Td) |       | 2..3  |       |       |       |
| 6     | (Ta, Tb) | 0..1  |       |       |       |       |
| 7     | (Tc, Ta) |       |       | 2..10 |       |       |
| 8     | (Tc, Tf) |       |       | 2..5  |       |       |

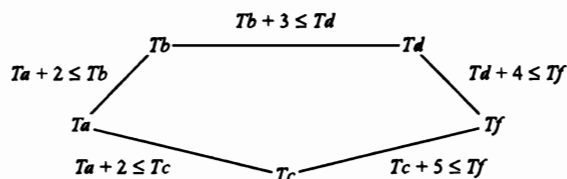


Рис. 14.2. Сеть ограничений для задачи составления расписания

Обратите внимание на то, как в методе обеспечения совместимости используются ограничения для сокращения областей определения переменных после получения новой информации. Поступление новой информации активизирует соответствующие ограничения, что приводит к сокращению областей определения рассматриваемых переменных. Подобное выполнение алгоритма может рассматриваться как управляемое данными. Ограничения являются активными в том смысле, что не ожидают явного вызова программистом, но активизируются автоматически при появлении соответствующей информации. Такой принцип вычисления, управляемого данными, дополнительно рассматривается в главе 23, в разделе “Программирование, управляемое шаблонами”.

## Упражнения

- 14.1. Попытайтесь выполнить алгоритм обеспечения совместимости дуг, выбрав другую последовательность дуг. Что при этом происходит?
- 14.2. Выполните алгоритм обеспечения совместимости дуг в заключительном состоянии трассировки, показанном в табл. 14.1, после присваивания переменной  $Tf$  значения 9.

### 14.1.3. Расширение Prolog для использования в качестве языка логического программирования в ограничениях

Рассмотрим взаимосвязь между языком Prolog и задачей удовлетворения ограничений. Базовый Prolog сам может рассматриваться как довольно специфический язык удовлетворения ограничений, в котором все ограничения имеют весьма жесткую форму. Они представляют собой ограничения равенства между термами. Эти ограничения равенства проверяются средствами согласования термов языка Prolog. Хотя ограничения, установленные между параметрами предикатов, также задаются в терминах других предикатов, эти вызовы предикатов в конечном итоге сводятся к согласованию. Prolog может быть расширен до “настоящего” языка CLP путем введения других типов ограничений, кроме согласования. Безусловно, должен быть также усовершенствован интерпретатор Prolog таким образом, чтобы он мог обрабатывать указанные ограничения других типов. Система CLP, способная обрабатывать арифметические ограничения равенства и неравенства, позволяет непосредственно решать задачи составления расписаний, подобные приведенным выше.

Программа с ограничениями интерпретируется примерно таким образом. Во время выполнения списка целей сопровождается множество текущих ограничений  $CurrConstr$ . Первоначально это множество является пустым. Цели в списке целей выполняются одна за другой в обычном порядке. Стандартные цели Prolog обрабатываются как обычно. При обработке цели с ограничениями  $Constr$  множества ограничений  $Constr$  и  $CurrConstr$  сливаются, в результате чего создается множество  $NewConstr$ . Затем процедура решения задач в ограничениях, предназначенная для работы с областью определения данного типа, пытается удовлетворить ограничение  $NewConstr$ . При этом возможны два основных результата: а) обнаруживается, что ограничения  $NewConstr$  удовлетворить невозможно, что соответствует недостижению цели и вызывает перебор с возвратами; б) не обнаруживается такой ситуации, что ограничения  $NewConstr$  удовлетворить невозможно, и эти ограничения максимально упрощаются процедурой решения задач в ограничениях. Например, два ограничения,  $X \leq 3$  и  $X \leq 2$ , упрощаются таким образом, что вместо них вводится одно ограничение —  $X \leq 2$ . Степень упрощения зависит от текущего состояния информации о переменных, а также от возможностей конкретной процедуры решения задач в ограничениях. Остальные цели в списке выполняются с множеством текущих ограничений, обновленным таким образом.

Системы CLP различаются по типам областей определения и типам ограничений, которые они способны обрабатывать. Семейства методов CLP упоминаются под именами в форме  $CLP(X)$ , где  $X$  обозначает область определения. Например, в методах  $CLP(R)$  областями определения переменных являются действительные числа, а в качестве ограничений применяются операции проверки на равенство и неравенство, а также операции сравнения действительных чисел. К системам  $CLP(X)$ , используемым в других областях определения, относятся следующие:  $CLP(Z)$  — целые числа,  $CLP(Q)$  — рациональные числа,  $CLP(B)$  — логические области определения и  $CLP(FD)$  — задаваемые пользователем конечные области определения. Доступные области определения и типы ограничений в фактических реализациях в значительной степени зависят от существующих методов решения конкретных типов ограничений. Например, в системах  $CLP(R)$  обычно доступны линейные равенства и неравенства, поскольку существуют эффективные методы обработки ограничений этих типов. С другой стороны, нелинейные ограничения имеют очень узкую область применения.

В последней части этой главы подробно рассматриваются системы  $CLP(R)$ ,  $CLP(Q)$  и  $CLP(FD)$ , в которых используются синтаксические соглашения для CLP в версии SICStus Prolog (см. раздел “Дополнительные источники информации” в конце главы).

## 14.2. Применение метода CLP для обработки действительных чисел — $CLP(R)$

Рассмотрим следующий запрос:

```
?- 1 + X = 5.
```

В языке Prolog такое согласование оканчивается неудачей, поэтому ответом системы Prolog является “no”. Но если пользователь имел в виду, что  $X$  — число, а знаком “+” обозначена операция арифметического сложения, то ответ  $X = 4$  был бы более приемлемым. Использование вместо знака “=” встроенного предиката `is` не позволяет полностью добиться такой интерпретации, а система  $CLP(R)$  позволяет. В соответствии с применяемыми синтаксическими соглашениями (версии SICStus Prolog) этот запрос  $CLP(R)$  может быть записан следующим образом:

```
?- { 1 + X = 5 }. % Числовое ограничение
X = 4
```

Это ограничение обрабатывается специализированной процедурой решения задач в ограничениях, которая способна выполнять операции с действительными числами и обычно может находить решения систем уравнений, заданных в виде равенств или неравенств определенных типов. В соответствии с применяемыми синтаксическими соглашениями множество ограничений вставляется в предложение Prolog в виде цели, заключенной в фигурные скобки. Отдельные ограничения разделяются запятыми и точками с запятой. Как и в языке Prolog, запятая означает конъюнкцию, а точка с запятой — дизъюнкцию. Поэтому конъюнкция ограничений  $C1$ ,  $C2$  и  $C3$  может быть записана так:

```
{ C1, C2, C3 }
```

Каждое ограничение задается в следующей форме:

```
Expr1 Operator Expr2
```

И `Expr1`, и `Expr2` представляют собой обычные арифметические выражения. Они могут также, в зависимости от конкретной системы  $CLP(R)$ , включать вызовы некоторых стандартных функций, таких как  $\sin(X)$ . В качестве `Operator` может быть задан один из следующих операторов, в зависимости от типа ограничения.

- `=`. Проверка на равенство.
- `=\=`. Проверка на неравенство.
- `<`, `<=`, `>`, `>=`. Арифметическое сравнение.

Теперь рассмотрим некоторые простые примеры использования этих ограничений и, в частности, определим, насколько более гибкими они являются по сравнению с обычными встроенными средствами языка Prolog.

В языке Prolog для преобразования значений температуры из градусов Цельсия в градусы Фаренгейта может применяться встроенный предикат `is`, например, следующим образом:

```
convert(Centigrade, Fahrenheit) :-
 Centigrade is (Fahrenheit - 32)*5/9.
```

С помощью этой программы можно легко преобразовать значение температуры 35 градусов Цельсия в градусы Фаренгейта, но обратное преобразование невозможно, поскольку предполагается, что при использовании встроенного предиката `is` все, что находится справа от него, должно быть конкретизировано. Для того чтобы обеспечить работу этой процедуры в обоих направлениях, необходимо проверить, являются ли ее параметры конкретизированы значением числа, а затем использовать формулу преобразования, подготовленную соответствующим образом для каждого случая. Но все эти операции могут быть реализованы гораздо более изящно в системе CLP(R), где одна и та же формула, интерпретируемая как числовое ограничение, действует в обоих направлениях, как показано ниже.

```
convert(Centigrade, Fahrenheit) :-
 { Centigrade = (Fahrenheit - 32)*5/9 }.
?- convert(35, F).
F = 95
?- convert(C, 95).
C = 35
```

Такая программа CLP(R) работает, даже если не конкретизирован ни один из двух параметров:

```
?- convert(C, F).
{ F = 32.0 + 1.8*C }
```

Поскольку вычисление в этом случае невозможно, ответом является формула, которая означает следующее: решение — это множество всех значений `F` и `C`, которые удовлетворяют этой формуле. Обратите внимание на то, что эта формула, выработанная системой CLP, представляет собой упрощение ограничения в рассматриваемой программе `convert`.

Типичная процедура решения задач в ограничениях способна решать системы линейных уравнений, заданных с помощью операторов проверки на равенство и неравенство, а также операторов арифметического сравнения. Ниже приведены подобные примеры.

```
?- { 3*X - 2*Y = 6, 2*Y = X }.
X = 3.0
Y = 1.5
?- { Z <= X-2, Z <= 6-X, Z+1 = 2 }.
Z = 1.0
{X >= 3.0}
{X <= 5.0}
```

Процедура поиска решения системы CLP(R) включает также средства линейной оптимизации, которые позволяют находить предельное значение заданного линейного выражения в области, которая удовлетворяет указанным линейным ограничениям. Для этого применяются следующие встроенные предикаты CLP(R):

```
minimize(Expr)
maximize(Expr)
```

В этих двух предикатах `Expr` — это линейное выражение в терминах переменных, которые появляются в линейных ограничениях. Указанные предикаты находят значения переменных, удовлетворяющих этим ограничениям, и соответственно минимизируют или максимизируют значение выражения, как показано ниже.

```

?- { X =< 5}, maximize(X).
X = 5.0
?- { X =< 5, 2 =< X}, minimize(2*X + 3).
X = 2.0
?- { X >=2, Y >=2, Y =< X+1, 2*Y =< 8-X, Z = 2*X + 3*Y}, maximize(Z).
X = 4.0
Y = 2.0
Z = 14.0
?- { X =< 5}, minimize(X).
no

```

В последнем примере значение  $X$  не имеет нижней границы, поэтому цель минимизации не достигается.

Следующие предикаты CLP(R) находят *супремум* (наименьшую верхнюю грань) или *инфимум* (наибольшую нижнюю грань) любого выражения:

```

sup(Expr, MaxVal)
inf(Expr, MinVal)

```

где Expr — это линейное выражение в терминах переменных с линейными ограничениями, а MaxVal и MinVal — максимальное и минимальное значения, которые принимает это выражение в той области, в которой удовлетворяются ограничения. В отличие от предикатов maximize/1 и minimize/1, переменные в выражении Expr не конкретизируются крайними значениями, как показано ниже.

```

?- { 2 =< X, X =< 5}, inf(X, Min), sup(X, Max).
Max = 5.0
Min = 2.0
{X >= 2.0}
{X =< 5.0}
?- {X >=2, Y >=2, Y =< X+1, 2*Y =< 8-X, Z = 2*X + 3*Y},
sup(Z,Max), inf(Z,Min), maximize(Z).
X = 4.0
Y = 2.0
Z = 14.0
Max = 14.0
Min = 10.0

```

Приведенный в начале этой главы простой пример составления расписания с заданиями a, b, c и d может быть представлен в системе CLP(R) в следующем непосредственном виде:

```

?- {Ta + 2=< Tb, % Задание a предшествует заданию b
Ta + 2 =< Tc, % Задание a предшествует заданию c
Tb + 3 =< Td, % Задание b предшествует заданию d
Tc + 5 =< Tf, % Задание c завершается ко времени завершения Tf
Td + 4 = < Tf }, % Задание d завершается ко времени завершения Tf
minimize(Tf).
Ta = 0.0
Tb = 2.0
Td = 5.0
Tf = 9.0
{Tc =< 4.0}
{Tc >= 2.0}

```

Следующий пример еще более наглядно показывает, насколько гибкими являются ограничения по сравнению со стандартными арифметическими средствами Prolog. Рассмотрим применение предиката fib( N, F) для вычисления N-го числа Фибоначчи F следующим образом: F(0)=1, F(1)=1, F(2)=2, F(3)=3, F(4)=5 и т.д. В общем для  $N > 1$ ,  $F(N) = F(N-1) + F(N-2)$ . Ниже приведено определение процедуры fib/2 на языке Prolog.

```

fib(N, F) :-
 N=0, F=1
 ;
 N=1, F=1
 ;

```



```

N>1,
N1 is N-1, fib(N1,F1),
N2 is N-2, fib(N2,F2),
F is F1 + F2.

```

Предполагается, что эта программа должна использоваться таким образом:

```

?- fib(6,F).
F=13

```

Но рассмотрим следующий вопрос, в котором предпринята попытка решить обратную задачу:

```

?- fib(N, 13).

```

Он приводит к ошибке, поскольку цель  $N > 1$  выполняется с неконкретизированным значением  $N$ . Но эта же программа, записанная с помощью определений CLP(R), может использоваться более гибко, как показано ниже.

```

fib(N, F) :-
{ N = 0, F = 1 }
;
{ N = 1, F=1 }
;
{ N > 1, F = F1+F2, N1 = N-1, N2 = N-2 },
fib(N1, F1),
fib(N2, F2).

```

Эта программа может быть вызвана на выполнение в противоположном направлении. Например, она позволяет следующим образом определить по числу Фибоначчи  $F$  его индекс  $N$ :

```

?- fib(N, 13).
N = 6

```

Но данная программа все еще сталкивается с затруднениями после получения вопроса, для которого нельзя найти удовлетворительного решения:

```

?- fib(N, 4).

```

Эта программа постоянно предпринимает попытки найти такие два числа Фибоначчи  $F1$  и  $F2$ , что  $F1 + F2 = 4$ . Она продолжает вырабатывать все более крупные значения  $F1$  и  $F2$  в расчете на то, что в конечном итоге их сумма будет равна 4. Программе неизвестно, что после того как сумма чисел Фибоначчи превысит 4, она будет лишь возрастать и поэтому никогда не станет равной 4. Наконец, этот бессмысленный поиск оканчивается переполнением стека. Решение, которое показывает, как исправить эту ситуацию, введя некоторые очевидные дополнительные ограничения в процедуру `fib`, является очень поучительным. Легко понять, что для всех  $N$  имеет место  $F(N) \geq N$ . Поэтому переменные  $N1$ ,  $F1$ ,  $N2$  и  $F2$  в данной программе должны всегда удовлетворять следующим ограничениям:  $F1 \geq N1$ ,  $F2 \geq N2$ . Эти дополнительные ограничения можно добавить к ограничениям в третьем дизъюнкте в теле предложения `fib`. В результате эта программа принимает вид

```

fib(N, F) :-
{ N > 1, F = F1+F2, N1 = N-1, N2 = N-2,
F1 >= N1, F2 >= N2 }, % Дополнительные ограничения
fib(N1, F1),
fib(N2, F2).

```

Указанные дополнительные ограничения позволяют программе определить, что приведенный выше вопрос должен закончиться неудачей:

```

?- fib(N, 4).
no

```

При рекурсивных вызовах процедуры `fib` фактически разворачивается выражение для  $F$  в условии  $F = 4$ :

```

4 = F = F1 + F2 =
F1' + F2' + F2 =
F1'' + F2'' + F2' + F2

```

После каждого развертывания этого выражения к предыдущим ограничениям добавляются новые ограничения. Ко времени получения данного выражения суммы с четырьмя термами процедура решения задач в ограничениях обнаруживает, что накопленные ограничения содержат противоречие, которое никогда не удастся разрешить.

Рассмотрим кратко системы CLP(Q), которые являются ближайшими аналогами систем CLP(R). Различие между ними состоит в том, что в системах CLP(R) действительные числа аппроксимируются числами с плавающей точкой, а область определения Q являются рациональные числа, т.е. дроби, состоящие из двух целых чисел. Они могут использоваться в качестве другого способа аппроксимации действительных чисел. Область определения Q может иметь преимущество над областью определения R (представленной с помощью чисел с плавающей точкой) в том, что решения некоторых арифметических ограничений могут быть определены в виде дробей точно, а числа с плавающей точкой позволяют получить лишь приближенные значения. Соответствующий пример приведен ниже.

```
?- { X=2*Y, Y=1-X }.
```

Процедура решения CLP(Q) дает следующий ответ:  $X = 2/3, Y = 1/3$ .

Процедура решения CLP(R) сообщает приблизительно такой ответ:  $X = 0.6666666666, Y = 0.3333333333$ .

## Упражнение

**14.3.** Покажите, как при выполнении запроса “?-fib( N, 4).”, приведенного выше в качестве примера, процедура решения задач в ограничениях позволяет обнаружить, что накопленные ограничения не могут быть удовлетворены.

## 14.3. Планирование с помощью метода CLP

Задачи составления расписаний, которые рассматриваются в этом разделе, состоят из перечисленных ниже элементов.

- Множество задач  $T_1, \dots, T_n$ .
- Продолжительности  $D_1, \dots, D_n$  задач.
- Ограничения предшествования, заданные как отношения, следующим образом:  
`prec( Ti, Tj )`  
Такое ограничение указывает, что выполнение задания  $T_i$  должно закончиться до того времени, как может начаться выполнение задания  $T_j$ .
- Множество процессоров  $m$ , которые могут применяться для выполнения заданий.
- Ограничения на ресурсы: какие задания могут выполняться теми или иными процессорами (какие процессоры являются подходящими для выполнения конкретного задания).

Задача состоит в том, чтобы составить расписание, время завершения которого является минимальным. В расписании назначается процессор для каждого задания и задается время начала выполнения каждого задания. Безусловно, расписание должно удовлетворять всем ограничениям предшествования и распределения ресурсов: каждое задание должно быть выполнено подходящим процессором, причем ни один процессор не может выполнять два задания одновременно. В соответствующей формулировке CLP задачи составления расписания применяются следующие переменные: значение времени начала,  $S_1, \dots, S_n$ , и имена процессоров, назначенных для каждого задания,  $P_1, \dots, P_n$ .

Простым частным случаем этой задачи составления расписания является отсутствие ограничений на ресурсы. В таком случае предполагается, что ресурсы не ограничены, поэтому в любое время всегда имеется свободный процессор для выполнения

любого задания. Таким образом, достаточно добиться удовлетворения ограничений, соответствующих отношениям предшествования между заданиями. Как уже было показано во вступительном разделе данной главы, подобная задача может быть сформулирована очень просто. Предположим, что имеется следующее ограничение предшествования, касающееся заданий  $a$  и  $b$ , —  $\text{prec}(a, b)$ . Допустим, что продолжительность задания  $a$  равна  $Da$ , а значения времени начала выполнения заданий  $a$  и  $b$  равны  $Sa$  и  $Sb$ . Чтобы было удовлетворено ограничение предшествования, значения  $Sa$  и  $Sb$  должны соответствовать следующему ограничению:

```
{ Sa + Da =< Sb }
```

Кроме того, требуется обеспечить, чтобы ни одно задание  $T_i$  не могло начаться до времени 0, а все задания должны быть выполнены ко времени завершения расписания  $\text{FinTime}$  следующим образом:

```
{ Si >= 0, Si + Di =< FinTime }
```

Для определения конкретной задачи составления расписания введем следующие предикаты:

```
tasks([Task1/Duration1, Task2/Duration2, ...])
```

Приведенный ниже предикат задает список всех имен заданий и значений их продолжительности.

```
prec(Task1, Task2)
```

Этот предикат определяет отношение предшествования между заданиями  $\text{Task1}$  и  $\text{Task2}$ .

```
resource(Task, [Proc1, Proc2, ...])
```

Данный предикат указывает, что задание  $\text{Task}$  может быть выполнено любым из процессоров  $\text{Proc1}$ ,  $\text{Proc2}$ , ... . Обратите внимание на то, что это — спецификация задачи составления расписания, аналогичная используемой в главе 12 (раздел 12.3), где для составления наилучшего расписания применялся эвристический поиск по заданному критерию. Но фактически применяемая здесь формулировка является немного более общей. В главе 12 ограничения на ресурсы заключались лишь в том, что лимитировалось количество процессоров, но все эти процессоры считались пригодными для выполнения любого задания.

Вначале рассмотрим простой случай без ограничений на ресурсы. Один из способов решения задачи планирования такого рода см. в разделе 14.1. Программа, приведенная в листинге 14.1, является более общей реализацией той же идеи. В этой программе подразумевается использование некоторого определения конкретной задачи планирования с использованием представления, описанного выше. Основным предикатом является

```
schedule(Schedule, FinTime)
```

где  $\text{Schedule}$  — наилучшее расписание для задачи, которая определена с помощью предикатов  $\text{tasks}$  и  $\text{prec}$ , а  $\text{FinTime}$  — время завершения расписания. Расписание имеет следующее представление:

```
Schedule = [Task1/Start1/Duration1, Task2 /Start2/Duration2, ...]
```

#### Листинг 14.1. Планирование с учетом ограничений предшествования и без учета ограничений на ресурсы

```
% Планирование с помощью метода CLP с неограниченными ресурсами
```

```
schedule(Schedule, FinTime) :-
 tasks(TasksDurs),
 precedence_constr(TasksDurs, Schedule, FinTime), % Сформировать отношения
 % предшествования
 minimize(FinTime).
```

```
precedence_constr([], [], FinTime).
```

```

precedence_constr([T/D | TDs], [T/Start/D | Rest], FinTime) :-
 { Start >= 0, % Выполнение должно начаться не раньше времени 0
 Start + D <= FinTime, % Временем завершения должно быть FinTime
 precedence_constr(TDs, Rest, FinTime),
 prec_constr(T/Start/D, Rest).

prec_constr(_, []).

prec_constr(T/S/D, [T1/S1/D1 | Rest]) :-
 (prec(T, T1), !, { S+D <= S1 }
 ;
 prec(T1, T), !, { S1+D1 <= S }
 ;
 true),
 prec_constr(T/S/D, Rest).

% Список заданий, подлежащих планированию
tasks([t1/5, t2/7, t3/10, t4/2, t5/9]).

% Ограничения предшествования
prec(t1, t2). prec(t1, t4). prec(t2, t3). prec(t4, t5).

```

Процедура `schedule`, по сути, выполняет описанные ниже действия.

1. Формирует ограничения неравенства между значениями времени начала в расписании, соответствующие отношениям предшествования между заданиями.
2. Минимизирует время завершения с учетом сформированных ограничений неравенства.

Поскольку все ограничения являются линейными неравенствами, данная задача сводится к задаче линейной оптимизации, для решения которой в системе CLP(R) предусмотрены встроенные средства.

**Предикат**

```
precedence_constr(TasksDurations, Schedule, FinTime)
```

формирует ограничения между значениями времени начала заданий в расписании `Schedule` и значением времени завершения расписания `FinTime`.

**Предикат**

```
prec_constr(Task/Start/Duration, RestOfSchedule)
```

формирует ограничения между значением времени начала `Start` задания `Task` и значениями времени начала заданий в списке `RestOfSchedule` таким образом, чтобы эти ограничения на значения времени начала соответствовали ограничениям предшествования между заданиями.

В программе, приведенной в листинге 14.1, содержится также определение простой задачи составления расписания с пятью заданиями. Эта программа-планировщик вызывается на выполнение с помощью следующего вопроса:

```

?- schedule(Schedule, FinTime).
FinTime = 22,
Schedule = [t1/0/5,t2/5/7,t3/12/10,t4/4/2,t5/5/9],
{S5 <= 13}
{S4 >= 5}
{S4-S5 <= -2}

```

Для заданий `t4` и `t5` предусмотрена определенная степень свободы в отношении времени их начала. При всех значениях времени начала `S4` и `S5` для заданий `t4` и `t5` в пределах указанных интервалов достигается оптимальное время завершения расписания. Три другие задания (`t1`, `t2` и `t3`) находятся на критическом пути, и время их начала не может сдвигаться.

Теперь рассмотрим задачи планирования более сложного типа — с ограничениями на ресурсы. Например, в задаче планирования, приведенной в главе 12 (см. раздел 12.3, рис. 12.6), имеются всего три процессора. Хотя любой из процессоров может выполнять какое угодно задание, это ограничение означает, что одновременно могут обрабатываться не больше трех заданий.

На этот раз приходится иметь дело с ограничениями следующих двух типов.

1. Отношения предшествования между заданиями.
2. Ограничения на ресурсы.

Ограничения предшествования могут обрабатываться таким же образом, как и в программе, приведенной в листинге 14.1. Теперь рассмотрим ограничения на ресурсы. Для удовлетворения ограничений на ресурсы необходимо решить задачу назначения некоторого процессора для каждого задания. Такую задачу можно решить, введя для каждого задания  $T_i$  еще одну переменную,  $P_i$ , возможными значениями которой являются имена процессоров. В соответствии с этим необходимо немного расширить представление расписания:

```
Schedule = [Task1/Proc1/Start1/Dur1, Task2/Proc2/Start2/Dur2, ...]
```

где Proc1 — процессор, назначенный заданию Task1, Start1 — время начала задания Task1 и Dur1 — его продолжительность. Теперь с использованием такого представления расписания можно разработать программу, приведенную в листинге 14.2, как расширение программы из листинга 14.1. Основным предикатом снова является `schedule( BestSchedule, BestTime)`

Этот предикат позволяет составить расписание с минимальным временем завершения BestTime. Ограничения неравенства, налагаемые на значения времени начала с учетом отношений предшествования между заданиями, снова формируются с помощью следующего предиката:

```
precedence_constr(TasksDurations, Schedule, FinTime)
```

#### Листинг 14.2. Программа составления расписания CLP(R) для задач с ограничениями предшествования и ограничениями на ресурсы

```
% Планирование с помощью метода CLP с ограниченными ресурсами

schedule(BestSchedule, BestTime) :-
 tasks(TasksDurs),
 precedence_constr(TasksDurs, Schedule, FinTime), % Задать неравенства,
 % обозначающие отношения предшествования
 initialise_bound, % Инициализировать предельное
 % значение времени завершения
 assign_processors(Schedule, FinTime), % Назначить процессоры для заданий
 minimize(FinTime),
 update_bound(Schedule, FinTime),
 fail % Выполнить перебор с возвратами, чтобы найти другие расписания
 ;
 bestsofar(BestSchedule, BestTime). % Наилучшее к этому времени расписание

% precedence_constr(TasksDurs, Schedule, FinTime):
% На основании исходных данных, которые представляют собой задания и значения
% их продолжительности, сформировать структуру Schedule, состоящую из
% переменных, которые представляют время начала выполнения заданий.
% Значения этих переменных и времени завершения FinTime определяются
% с учетом ограничений предшествования, сформулированных в виде неравенств

precedence_constr([], [], FinTime).

precedence_constr([T/D | TDs], [T/Proc/Start/D | Rest], FinTime) :-
 { Start >= 0, % Выполнение должно начаться не раньше времени 0
 Start + D <= FinTime }, % Временем завершения должно быть FinTime
 precedence_constr(TDs, Rest, FinTime),
```

```

prec_constr(T/Proc/Start/D, Rest).

prec_constr(_, []).

prec_constr(T/P/S/D, [T1/P1/S1/D1 | Rest]) :-
 (prec(T, T1), !, { S+D =< S1}
 ;
 prec(T1, T), !, { S1+D1 =< S}
 ;
 true),
 prec_constr(T/P/S/D, Rest).

% assign_processors(Schedule, FinTime):
% назначить процессоры заданиям в расписании Schedule

assign_processors([], FinTime).

assign_processors([T/P/S/D | Rest], FinTime) :-
 assign_processors(Rest, FinTime),
 resource(T, Processors), % Задание T может быть выполнено любым
 % процессором из списка Processors
 member(P, Processors), % Выбрать один из подходящих процессоров
 resource_constr(T/P/S/D, Rest), % Учесть ограничения на ресурсы
 bestsofar(_, BestTimeSoFar),
 { FinTime < BestTimeSoFar }. % Новое расписание лучше любого
 % найденного к этому времени

% resource_constr(ScheduledTask, TaskList):
% сформировать ограничения так, чтобы гарантировалось отсутствие
% противоречивых назначений ресурсов в списках ScheduledTask и TaskList

resource_constr(_, []).

resource_constr(Task, [Task1 | Rest]) :-
 no_conflict(Task, Task1),
 resource_constr(Task, Rest).

no_conflict(T/P/S/D, T1/P1/S1/D1) :-
 P \== P1, ! % Разные процессоры
 ;
 prec(T, T1), ! % Ограничения уже учтены
 ;
 prec(T1, T), ! % Ограничения уже учтены
 ;
 { S+D =< S1 % Тот же процессор; перекрытие по времени отсутствует
 ;
 S1+D1 =< S }.

initialise_bound :-
 retract(bestsofar(_, _)), fail
 ;
 assert(bestsofar(dummy_schedule, 9999)). % Предполагается, что время
 % завершения ни при каких условиях не превышает 9999

% update_bound(Schedule, FinTime):
% обновить определение наилучшего расписания и значение времени

update_bound(Schedule, FinTime) :-
 retract(bestsofar(_, _)), !,
 assert(bestsofar(Schedule, FinTime)).

% Список заданий, которые должны быть включены в расписание

tasks([t1/4,t2/2,t3/2, t4/20, t5/20, t6/11, t7/11]).

```

## % Ограничения предшествования

```
prec(t1, t4). prec(t1, t5). prec(t2, t4). prec(t2, t5).
prec(t2, t6). prec(t3, t5). prec(t3, t6). prec(t3, t7).
```

```
% resource(Task, Processors):
```

```
% любой процессор из списка Processors, пригодный для выполнения задания Task
```

```
resource(_, [1,2,3]). % Три процессора, причем любой из них пригоден
 % для выполнения любого задания
```

---

Данная программа почти аналогична приведенной в листинге 14.1. Единственное небольшое различие между ними связано с разными представлениями расписания.

Теперь рассмотрим ограничения на ресурсы. Задачу составления расписания с учетом этих ограничений нельзя решить столь же эффективно, как и с учетом ограничений предшествования. Чтобы удовлетворить ограничения на ресурсы, необходимо найти оптимальное распределение процессоров по заданиям. Для этого требуется выполнить поиск среди возможных вариантов назначения, а общего способа выполнения такого поиска за время, измеряемое некоторым полиномом, не существует. В программе, приведенной в листинге 14.2, такой поиск выполняется с помощью *метода ветвей и границ*, который можно кратко описать следующим образом. С помощью недетерминированного способа один за другим формируются варианты расписания (этот способ состоит в том, что формируется расписание и вызывается цель fail). Наилучшее расписание из всех составленных до сих пор вносится в базу данных как факт. После составления каждого нового расписания в базе данных обновляется расписание, наилучшее к этому времени (которое обозначается как *bestsofar*). При составлении нового расписания наилучшее к этому моменту время завершения используется в качестве верхней границы для времени завершения нового расписания. Как только обнаруживается, что новое частично сформированное расписание не позволяет достичь сокращения времени завершения по сравнению с наилучшим к данному моменту, происходит отказ от дальнейшей работы над этим расписанием.

Эта идея реализована в программе, приведенной в листинге 14.2, следующим образом. Процедура `assign_processors` недетерминированно назначает заданиям один за другим подходящие процессоры. Назначение процессора заданию приводит к появлению дополнительных ограничений на значения времени начала, поскольку необходимо обеспечить, чтобы не перекрывалось время между заданиями, назначенными одному и тому же процессору. Поэтому после каждого назначения процессора заданию частично составленное расписание улучшается. После каждого такого улучшения частично составленного расписания оно проверяется для определения того, есть ли какая-либо возможность улучшить расписание, которое к данному времени является наилучшим. Для того чтобы текущее частично составленное расписание предоставляло хотя бы малейшую такую возможность, его значение `FinTime` должно быть меньше по сравнению с наилучшим временем, достигнутым до сих пор. В программе это условие учитывается с помощью следующего ограничения:

```
{ FinTime < BestTimeSoFar }
```

Если это ограничение несовместимо с другими текущими ограничениями, то данное частично составленное расписание не дает ни малейшей возможности для улучшения. А поскольку для окончательного составления этого расписания необходимо удовлетворить еще больше ограничений на ресурсы, то фактически время его завершения может в конечном итоге стать еще хуже. Поэтому, если ограничение `FinTime < BestTimeSoFar` удовлетворить невозможно, то работа над текущим частично составленным расписанием прекращается; в противном случае процессору назначается другое задание и т.д. После составления каждого полного расписания можно гарантировать, что время его завершения меньше по сравнению с наилучшим временем завершения, найденным до сих пор. Поэтому обновляется расписание, наилучшее к

данному моменту. Наконец, если характеристики, наилучшие к данному моменту, нельзя улучшить, поиск останавливается. Безусловно, этот алгоритм вырабатывает только одно из многих возможных наилучших расписаний.

Следует отметить, что этот процесс является комбинаторно сложным из-за экспоненциального количества возможных вариантов назначения процессоров заданиям. Но благодаря тому, что характеристики текущего частично составленного расписания контролируются с помощью значения *BestTimeSoFar*, удается отказаться от целых групп некачественных расписаний еще до их полного составления. Количество времени вычисления, которое экономится благодаря этому, зависит от того, насколько удачно выбрана эта верхняя граница. Если верхняя граница не оставляет возможностей для маневра, то некачественные расписания распознаются и отбрасываются на ранних этапах их создания, что позволяет сэкономить много времени. Поэтому, чем быстрее удастся найти какое-то качественное расписание, тем скорее начинает применяться жестко заданная верхняя граница и тем большая часть пространства поиска исключается из рассмотрения.

В листинге 14.2 содержится также спецификация задачи составления расписания (см. рис. 12.6), которая подготовлена в соответствии с принятыми соглашениями по оформлению условий задачи. Вопрос, с помощью которого может быть составлено расписание, соответствующее условиям этой задачи, приведен ниже.

```
?- schedule(Schedule, FinTime).
FinTime = 24
Schedule = [t1/3/0/4,12/2/0/2,13/1/0/2, t4/3/4/20,
t5/2/4/20, 16/1/2/11,17/1/13/11]
```

Задача *t1* выполняется процессором 3, начиная со времени 0, задача *t2* выполняется процессором 2, начиная со времени 0 и т.д. В этой задаче планирования представляет интерес еще один нюанс. Все три доступных процессора являются эквивалентными, поэтому перестановка списков назначений процессоров заданиям не приводит к каким-либо изменениям. Таким образом, нет смысла выполнять поиск по всем подобным перестановкам, так как они должны давать одинаковые результаты. Чтобы избежать таких бесполезных перестановок, можно, например, закрепить процессор 1 за заданием *t7* и ограничить выбор процессора для задания *t6* только процессорами 1 и 2. Такое решение можно легко реализовать, изменив предикат *resource*. Тем не менее, хотя в целом это — неплохая идея, как оказалось, ее нет смысла реализовывать в данном конкретном примере. Несмотря на то что при этом количество возможных вариантов назначения могло бы сократиться в 6 раз, экономия времени является незначительной, что на первый взгляд кажется необъяснимым. Но причина этого состоит в том, что после обнаружения оптимального расписания устанавливается жесткая верхняя граница выбора времени завершения, поэтому в дальнейшем отказ от других возможных назначений процессора происходит очень быстро.

## Упражнения

- 14.4. Проведите эксперименты с программой, приведенной в листинге 14.1. Попробуйте применить разные спецификации ресурсов, предназначенные для исключения бесполезных перестановок, и проведите измерения продолжительности времени выполнения программы. В чем состоят достигнутые улучшения?
- 14.5. В программе, приведенной в листинге 14.2, верхняя граница значений времени завершения (*bestsofar*) инициализируется большим числовым значением, которое намного превосходит все возможные значения времени завершения. Это позволяет гарантировать, что оптимальное расписание будет находиться в пределах установленной границы и будет обнаружено программой. Такой подход, хотя и безопасный, является неэффективным, поскольку подобная верхняя граница, оставляющая большую свободу для маневра, не позволяет достаточно качественно ограничивать поиск. Рассмотрите другие подходы к инициализации и корректировке верхней границы, например, при которых работа



начинается с очень низкой границы и увеличивается по мере необходимости (если в пределах этой границы не существует ни одного расписания). Сравните значения продолжительности работы программы при использовании разных подходов. Проведите измерения времени выполнения для того случая, когда граница сразу же устанавливается равной действительно минимальному времени завершения.

## 14.4. Программа моделирования в ограничениях

Иногда с помощью системы CLP(R) удается найти весьма изящные решения задач числового моделирования. Такие средства являются особенно подходящими, если моделируемая система может рассматриваться как состоящая из большого количества компонентов и соединений между этими компонентами. К таким системам относятся, например, электрические схемы, характерными компонентами которых являются резисторы и конденсаторы.

С компонентами связаны параметры и переменные, имеющие реальное значение, такие как электрические сопротивления, напряжения и токи. Подобная организация модели хорошо сочетается со стилем программирования в ограничениях. Законы физики налагают ограничения на переменные, связанные с компонентами. Соединения между компонентами налагают дополнительные ограничения. Поэтому для решения задач числового моделирования с помощью системы CLP(R) для семейства систем, таких как электрические сети, необходимо определить законы, распространяющиеся на компоненты тех типов, которые используются в данной проблемной области, а также законы, регламентирующие соединения компонентов. Эти законы задаются как ограничения на переменные. Затем для моделирования конкретной системы из такого семейства необходимо определить конкретные компоненты и соединения в системе. В результате этого интерпретатор CLP налагает ограничения на всю систему и осуществляет моделирование по принципу удовлетворения ограничений. Безусловно, такой подход является эффективным, если типы ограничений, относящиеся к моделируемой проблемной области, могут эффективно обрабатываться данной конкретной системой CLP.

В данном разделе этот подход применяется для моделирования электрических схем, состоящих из резисторов, диодов и источников питания. Отношения между напряжениями и токами в таких схемах являются кусочно-линейными. Учитывая то, что применяемая система CLP(R) эффективно обрабатывает линейные равенства и неравенства, она является подходящим инструментальным средством для моделирования подобных схем.

На рис. 14.3 показаны рассматриваемые компоненты и соединения, а также соответствующие ограничения, налагаемые этими компонентами и соединениями. Такие элементы могут быть определены в программе CLP(R) следующим образом. Резистор имеет некоторое сопротивление  $R$  и две клеммы,  $T1$  и  $T2$ . Переменными, связанными с каждой клеммой, являются электрический потенциал  $V$  и ток  $I$  (направленный в резистор). Поэтому клемма  $T$  представляет собой пару  $(V, I)$ . Определяемое законами физики поведение резистора можно описать с помощью следующего предиката:

```
resistor((V1,I1), (V2,I2), R) :-
 { I1 = -I2, V1-V2 = I1*R }.
```

Поведение источника питания можно определить аналогичным образом, как показано в программе, приведенной в листинге 14.3. В этом листинге дано также определение диода. А что касается соединений, то лучше всего определить общий случай, в котором соединено любое количество клемм, следующим образом:

```
conn([Terminal1, Terminal2, ...])
```

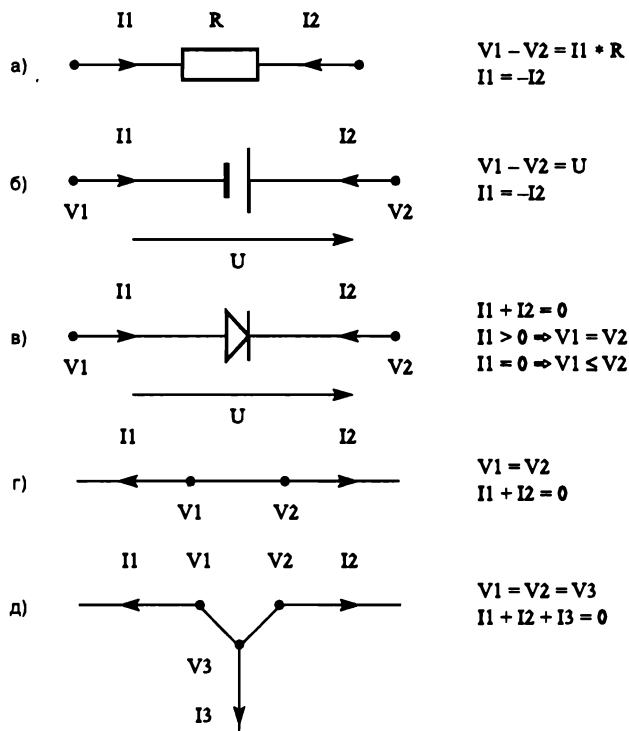


Рис. 14.3. Компоненты и соединения для электрических схем и соответствующие ограничения: а) резистор; б) источник питания; в) диод; г) соединение между двумя клеммами; д) соединение между тремя клеммами

#### Листинг 14.3. Ограничения для некоторых электрических компонентов и соединений

```
% Модель электрической схемы, заданная средствами CLP(R)

% resistor(T1, T2, R):
% R - резистор; T1, T2 - его клеммы

resistor((V1,I1), (V2,I2), R) :-
 { I1 = -I2, V1-V2 = I1*R }.

% diode(T1, T2):
% T1, T2 - клеммы диода.
% Диод пропускает ток в направлении от T1 к T2

diode((V1,I1), (V2,I2)) :-
 { I1 + I2 = 0 },
 { I1 > 0, V1 = V2
 ;
 I1 = 0, V1 =< V2 }.

battery((V1,I1), (V2,I2), Voltage) :-
 { I1 + I2 = 0, Voltage = V1 - V2 }.

% conn([T1,T2,...]):
% Соединение клемм T1, T2, ...
% Поскольку потенциалы на всех клеммах соединения одинаковы,
```

% сумма токов равна нулю

```
conn(Terminals) :-
 conn(Terminals, 0).
```

```
conn([(V,I)], Sum) :-
 { Sum + I = 0 }.
```

```
conn([(V1,I1), (V2,I2) | Rest], Sum) :-
 { V1 = V2, Sum1 = Sum + I1 },
 conn([(V2, I2) | Rest], Sum1).
```

---

Потенциалы на всех клеммах соединения должны быть одинаковыми, а сумма токов через все клеммы должна быть равна нулю.

Теперь можно легко составлять моделируемые схемы. Некоторые примеры схем приведены на рис. 14.4. На этом рисунке даны также определения таких схем, которые могут использоваться в программе моделирования, выполняемой в системе CLP(R). Рассмотрим схему на рис. 14.4, а. Следующий пример показывает, что данная программа моделирования может до некоторой степени использоваться также для проектирования, а не только для моделирования. В частности, было решено в определении предиката `circuit_a` на рис. 14.4, а сделать клемму T21 одним из параметров этого предиката. Это дает возможность “считывать” значения потенциала и тока в данной точке схемы. Потенциал на клемме T2 имеет постоянное значение 0, источник питания имеет напряжение 10 В, но резисторы остались незадаанными (они также являются параметрами предиката `circuit_a`).

Рассмотрим вопрос о том, какими должны быть резисторы, чтобы напряжение на клемме T21 было равно 6 В, а ток — 1 А.

```
?- circuit_a(R1, R2, (6,1)).
R1 = 4.0
R2 = 6.0
```

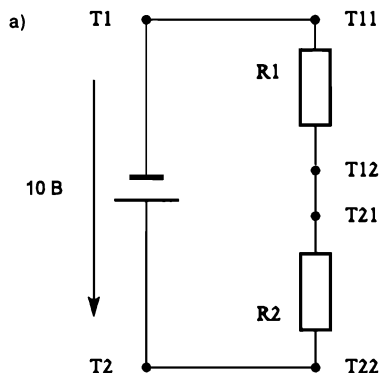
Теперь рассмотрим более сложную схему (см. рис. 14.4, б). В этом случае можно задать вопрос о том, какими будут электрические потенциалы и ток на “среднем” резисторе R5, если источник питания имеет напряжение 10 В.

```
?- circuit_b(10, _, _, _, T51, T52).
T51 = (7.340425531914894, 0.0425531914893617)
T52 = (5.212765957446809, -0.0425531914893617)
```

Итак, потенциалы на клеммах резистора R5 равны соответственно 7,340 В и 5,213 В, а ток равен 0,04255 А.

## Упражнение

- 14.6. Проведите эксперименты с программами, приведенными на рис. 14.4. Определите другие схемы. Например, дополните схему на рис. 14.4, б, установив диод последовательно с резистором R5. Как это повлияет на потенциал клеммы T51? Попробуйте также переставить диод в противоположном направлении.

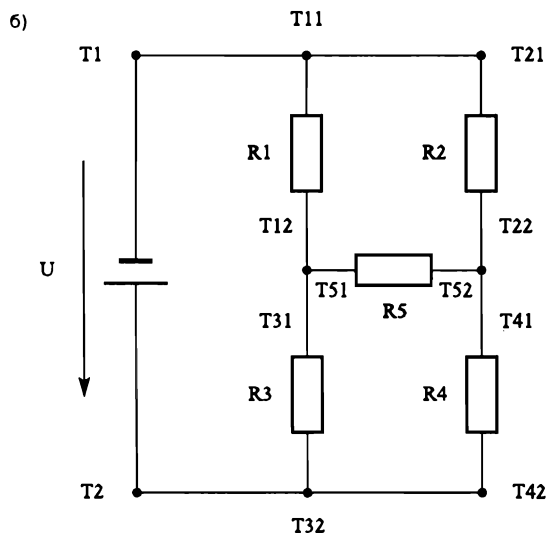


```

circuit_a(R1, R2, T21) :-
 T2 = (0, _),
 battery(T1, T2, 10),
 resistor(T11, T12, R1),
 resistor(T21, T22, R2),
 conn([T1, T11]),
 conn([T12, T21]),
 conn([T2, T22]).

```

% Потенциал клеммы T2 равен 0  
 % Источник питания на 10 В



```

circuit_b(U, T11, T21, T31, T41, T51, T52) :-
 T2 = (0, _),
 battery(T1, T2, U),
 resistor(T11, T12, 5),
 resistor(T21, T22, 10),
 resistor(T31, T32, 15),
 resistor(T41, T42, 10),
 resistor(T51, T52, 50),
 conn([T1, T11, T21]),
 conn([T12, T31, T51]),
 conn([T22, T41, T52]),
 conn([T2, T32, T42]).

```

% Потенциал клеммы T2 равен 0  
 % R1 = 5  
 % R2 = 10  
 % R3 = 15  
 % R4 = 10  
 % R5 = 50

Рис. 14.4. Две электрические схемы

## 14.5. Применение метода CLP для поддержки конечных областей определения — CLP(FD)

Система CLP, предназначенная для работы с конечными областями определения, реализована на основе некоторых специальных механизмов. Их работа будет показана в настоящем разделе на типичных примерах с использованием синтаксиса и некоторых предикатов из библиотеки CLP(FD) версии SICStus Prolog. В данном разделе рассматривается лишь небольшое подмножество этих предикатов, которое требуется для решения приведенных здесь примеров.

В качестве областей определения переменных будут использоваться множества целых чисел. Для указания области определения переменной применяется предикат `in/2`, что записывается следующим образом:

```
X in Set
```

В данном случае `Set` может быть одним из следующих.

- `{Integer1, Integer2, ...}`. Множество заданных целых чисел.
- `Term1..Term2`. Множество целых чисел между `Term1` и `Term2`.
- `Set1 \/ Set2`. Объединение множеств `Set1` и `Set2`.
- `Set1 /\ Set2`. Пересечение множеств `Set1` и `Set2`.
- `\Set1`. Дополнение множества `Set1`.

Арифметические ограничения имеют следующую форму:

```
Exp1 Relation Exp2
```

где `Exp1` и `Exp2` — арифметические выражения, а `Relation` может представлять собой одно из следующих.

- `#=`. Равно.
- `#\=`. Не равно.
- `#<`. Меньше.
- `#>`. Больше.
- `#=<`. Меньше или равно.
- `#>=`. Больше или равно.

В качестве примера рассмотрим следующие вопросы:

```
?- X in 1..5, Y in 0..4,
X #< Y, Z #= X+Y+1.
X in 1..3
Y in 2..4
Z in 3..7
?- X in 1..5, X in \{4}.
X in (1..3) \/ {5}
```

Предикат `indomain(X)` назначает с помощью перебора с возвратами возможные значения переменной `X`, например, следующим образом:

```
?- X in 1..3, indomain(X).
X = 1;
X = 2;
X = 3
```

Ниже показаны некоторые полезные предикаты, которые определены на списках переменных.

- `domain( L, Min, Max)`. Этот предикат означает, что все переменные в списке `L` имеют области определения `Min..Max`.

- `all_different( L)`. Этот предикат означает, что все переменные в списке `L` должны иметь разные значения.
- `labeling( Options, L)`. Этот предикат вырабатывает возможные конкретные значения переменных в списке `L`. Параметр `Options` представляет собой список опций, касающихся того порядка, в каком осуществляется *разметка* ("labelling"; отсюда и имя предиката) переменных в списке `L`. Если параметр `Options = []`, то по умолчанию переменные размечаются слева направо; при этом возможные значения берутся из списка один за другим, от меньшего к большему. Эта простейшая стратегия разметки является подходящей для всех примеров настоящей главы, хотя она не всегда оказывается наиболее эффективной. Ниже приведены некоторые примеры.

```
?- domain([X,Y], 1, 2), labeling([], [X,Y]).
X=1, Y=1;
X=1, Y=2;
X=2, Y=1;
X=2, Y=2
?- L = [X,Y], domain(L, 1, 2), all_different(L), labeling([], L).
L = [1,2], X = 1, Y = 2;
L = [2,1], X = 2, Y = 1
```

С помощью этих примитивов можно легко решать числовые ребусы. Рассмотрим следующую задачу:

```
DONALD
+ GERALD

ROBERT
```

Каждая буква в приведенном выше ребусе должна быть заменена отличной от других десятичной цифрой таким образом, чтобы эти цифры составляли правильное арифметическое выражение. В листинге 14.4 показана программа `CLP(FD)` для решения этой задачи. Ниже приведен запрос к этой программе.

```
?- solve(N1, N2, N3).
N1 = [5,2,6,4,8,5]
N2 = [1,9,7,4,8,5]
N3 = [7,2,3,9,7,0]
```

#### Листинг 14.4. Решение числового ребуса в системе `CLP(FD)`

```
% Программа решения числового ребуса DONALD+GERALD=ROBERT средствами CLP(FD)

solve([D,O,N,A,L,D], [G,E,R,A,L,D], [R,O,B,E,R,T]) :-
 Vars = [D,O,N,A,L,G,E,R,B,T], % Переменные, необходимые для решения задачи
 domain(Vars, 0, 9), % Все эти переменные обозначают десятичные цифры
 all_different(Vars), % Все эти переменные - разные
 100000*D + 10000*O + 1000*N + 100*A + 10*L + D +
 100000*G + 10000*E + 1000*R + 100*A + 10*L + D #=
 100000*R + 10000*O + 1000*B + 100*E + 10*R + T,
 labeling([], Vars).
```

В листинге 14.5 приведена программа `CLP(FD)` для решения знакомой задачи с восемью ферзями. В качестве указания по составлению подобных программ следует отметить, что обе программы (и для решения числовых ребусов, и для решения задачи с восемью ферзями) имеют следующую основную структуру: вначале задаются области определения переменных, затем налагаются определенные ограничения и в конечном итоге предикат разметки находит конкретные решения. Это — обычная структура программ `CLP(FD)`.

## Листинг 14.5. Программа CLP(FD) для задачи с восемью ферзями

% Программа решения задачи с восемью ферзями средствами CLP(FD)

```
solution(Ys) :-
 Ys = [_,'_','_','_','_','_','_','_'],
 domain(Ys, 1, 8),
 all_different(Ys),
 safe(Ys),
 labeling([], Ys).

safe([]).

safe([Y | Ys]) :-
 no_attack(Y, Ys, 1),
 safe(Ys).

% no_attack(Y, Ys, D):
% ферзь, который определен переменной Y, не нападает ни на одного из ферзей
% в списке Ys;
% D - расстояние по горизонтали между первым ферзем и остальными ферзями

no_attack(Y, [], _).

no_attack(Y1, [Y2 | Ys], D) :-
 D #\= Y1-Y2,
 D #\= Y2-Y1,
 D1 is D+1,
 no_attack(Y1, Ys, D1).
```

Наконец, рассмотрим процедуру решения с помощью системы CLP(FD) задач оптимизации, таких как минимизация времени завершения при составлении расписаний. Для решения задач оптимизации удобно применять следующие встроенные предикаты CLP(FD):

minimize( Goal, X) и maximize( Goal, X)

Эти предикаты находят такое решение Goal, которое минимизирует (максимизирует) значение X. Как правило, Goal — это цель предиката indomain или предиката разметки, например, как показано ниже.

```
?- X in 1..20, V #= X*(20-X), maximize(indomain(X), V).
X = 10, V = 100
```

Простую задачу планирования (см. рис. 14.1) можно решить с помощью следующего запроса:

```
?- StartTimes = [Ta,Tb,Tc,Td,Tf], % Tf - время завершения
domain(StartTimes, 0, 20),
Ta #>= 0,
Ta + 2#<=Tb,
Ta + 2#<=Tc,
Tb + 3#<=Td,
Tc + 5#<=Tf,
Td + 4#<= Tf,
minimize(labeling([],StartTimes), Tf).
StartTimes = [0,2,2,5,9]
```

В данном случае вырабатывается только одно оптимальное решение.

## Упражнения

- 14.7. Измерьте время, необходимое программе, приведенной в листинге 14.4, для решения рассматриваемой задачи. Затем замените цель разметки следующим образом:

```
labeling([ff], Vars)
```

Опция разметки “ff” (сокращение от “first fail”) определяет принцип работы “до первого неудачного завершения”. Это означает, что в первую очередь значение присваивается переменной, которая в настоящее время имеет наименьшую область определения. А поскольку область определения мала, то, как правило, такая переменная может с наибольшей вероятностью стать причиной неудачи. Подобная стратегия разметки предназначена для обнаружения несовместимости как можно быстрее, чтобы был исключен бесполезный поиск среди несовместимых вариантов. Измерьте время выполнения модифицированной программы.

- 14.8. Обобщите программу CLP(FD) с восемью ферзями до программы с  $N$  ферзями. Для больших значений  $N$  хорошая стратегия разметки для  $N$  ферзей состоит в движении от “среднего”, когда поиск начинается в середине области определения, а затем продолжается среди значений, отстоящих все дальше и дальше от середины. Реализуйте эту стратегию разметки и сравните с помощью экспериментов ее эффективность с последовательной разметкой (как в листинге 14.5).

## Резюме

- *Задачи удовлетворения ограничений* формулируются в терминах переменных, областей определения переменных и ограничений между переменными.
- Задачи удовлетворения ограничений часто представляются в виде *сетей ограничений*.
- *Алгоритмы обеспечения совместимости* применяются к сетям ограничений и сокращают области определения переменных.
- *Логическое программирование в ограничениях* (Constraint Logic Programming — CLP) представляет собой сочетание подхода, связанного с удовлетворением ограничений, и логического программирования.
- *Системы CLP* различаются по типам областей определения и ограничений. Системы логического программирования в ограничениях классифицируются по типам ограничений следующим образом: CLP(R) — действительные числа; CLP(Q) — рациональные числа; CLP(Z) — целые числа; CLP(FD) — конечные области определения; CLP(B) — логические значения.
- Потенциал системы CLP в основном зависит от потенциала специализированных процедур решения задач в ограничениях, которые могут находить решения для систем ограничений конкретных типов и, возможно, осуществлять оптимизацию по заданному критерию в рамках этих ограничений.
- Одним из аспектов программирования CLP является *определение ограничений*, которые являются настолько жесткими, насколько это возможно. Чем более жесткими являются ограничения, тем в большей степени они сокращают пространства поиска и тем самым способствуют повышению эффективности. Повышению эффективности может способствовать даже добавление избыточных ограничений.
- К типичным *областям практического применения* систем CLP относятся планирование, снабжение и управление ресурсами.
- В этой главе приведены *программы CLP* для планирования, моделирования электрических схем, решения числовых ребусов и задачи с восемью ферзями.



- В данной главе рассматривались следующие понятия:
  - задачи удовлетворения ограничений;
  - удовлетворение ограничений;
  - сети ограничений;
  - алгоритмы обеспечения совместимости дуг;
  - логическое программирование в ограничениях (Constraint Logic Programming — CLP);
  - CLP(R), CLP(Q), CLP(FD);
  - метод ветвей и границ.

## Источники дополнительной информации

В [94] дано превосходное введение в методы удовлетворения ограничений и в программирование CLP. В [161] приведено получившее широкую известность описание различных методов программирования в системах CLP. В [67] и [91] приведен обзор методов решения задач в ограничениях. Современные результаты исследований в этой области публикуются в специализированном журнале *Constraints* (который выпускается издательством Kluwer Academic), а также в журналах *Journal of Logic Programming* и *Artificial Intelligence Journal*. Пример программы расчета чисел Фибоначчи, приведенный в этой главе, аналогичен примеру, приведенному в [35].

Для определения ограничений в данной главе используется такой же синтаксис, как и в версии SICStus Prolog [150].

# Представление знаний и экспертные системы

*В этой главе...*

|                                                                             |     |
|-----------------------------------------------------------------------------|-----|
| 15.1. Назначение и структура экспертной системы                             | 326 |
| 15.2. Представление знаний с помощью правил вывода                          | 328 |
| 15.3. Прямой и обратный логический вывод в системах, основанных на правилах | 331 |
| 15.4. Формирование объяснений                                               | 336 |
| 15.5. Учет неопределенности                                                 | 337 |
| 15.6. Байесовские сети доверия                                              | 340 |
| 15.7. Семантические сети и фреймы                                           | 348 |

Экспертная система — это программа, которая действует подобно эксперту в некоторой проблемной области. Она должна быть способной объяснять свои решения и сообщать, на основании каких рассуждений получены соответствующие выводы. От экспертной системы часто ожидают, что она сможет функционировать в условиях недоверенной и неполной информации. В этой главе рассматриваются основные принципы представления знаний и создания экспертных систем.

## 15.1. Назначение и структура экспертной системы

Экспертная система — это программа, которая действует как эксперт в некоторой, обычно узкой прикладной области. К типичным ее приложениям относятся такие задачи, как медицинская диагностика, поиск причин неисправностей оборудования или интерпретация результатов измерений. Экспертные системы должны быть способными решать задачи, для которых требуются специальные знания в определенной области. Они должны обладать этими знаниями, представленными в определенной форме. Поэтому такие системы называют также *системами, основанными на знаниях*. Но не каждую систему, основанную на знаниях, можно рассматривать как экспертную. Мы придерживаемся той точки зрения, что экспертная система должна быть способна в определенной степени объяснять свое поведение и свои решения пользователю, как это делают люди-эксперты. Такие функции объяснения особенно необходимы в областях, характеризующихся значительной неопределенностью (как медицинские диагнозы), поскольку они позволяют укрепить доверие пользователя к рекомендациям системы и дают возможность обнаружить возможную ошибку в ее рассуждениях. Поэтому экспертные системы должны обладать способностью дружественного взаимодействия с пользователем, благодаря которому ход рассуждений системы становится прозрачным для пользователя.

Дополнительным свойством, которое также часто требуется от экспертной системы, является способность функционировать в условиях неопределенной и неполной информации. Информация о задаче, требующей решения, может быть неполной или ненадежной, а отношения в проблемной области могут быть определены приближенно. Например, иногда невозможно утверждать с полной уверенностью, что у пациента наблюдаются некоторые симптомы и что известная часть результатов измерений является абсолютно правильной; иногда прием некоторых лекарств может стать причиной определенных расстройств, но обычно этого не происходит. Все эти обстоятельства требуют, чтобы система проводила свои рассуждения в условиях неопределенности.

Для формирования полноценной экспертной системы необходимо, как правило, реализовать в ней следующие функции.

- Функции решения задач, позволяющие использовать специальные знания в проблемной области (при этом может потребоваться обеспечить работу в условиях неопределенности).
- Функции взаимодействия с пользователем, которые, в частности, позволяют объяснить намерения и выводы системы в процессе решения задачи и по завершении этого процесса.

Каждая из этих функций может оказаться очень сложной, а способ их реализации может зависеть от проблемной области и практических требований. К тому же разработка и реализация проекта такой системы часто требует решения разнообразных и сложных проблем. К ним относится выбор способа представления знаний и соответствующих средств проведения рассуждений. В данной главе рассматриваются наиболее важные понятия в этой области, которые могут стать основой для дальнейшего усовершенствования. В главе 16 показано, как реализовать полнофункциональный командный интерпретатор экспертной системы, основанной на правилах. При этом вся сложность состоит в том, как поддерживать бесперебойное взаимодействие с пользователем в процессе проведения рассуждений.

Разработку экспертной системы удобно разделить на следующие три главных модуля (рис. 15.1).

1. База знаний.
2. Машина логического вывода.
3. Пользовательский интерфейс.

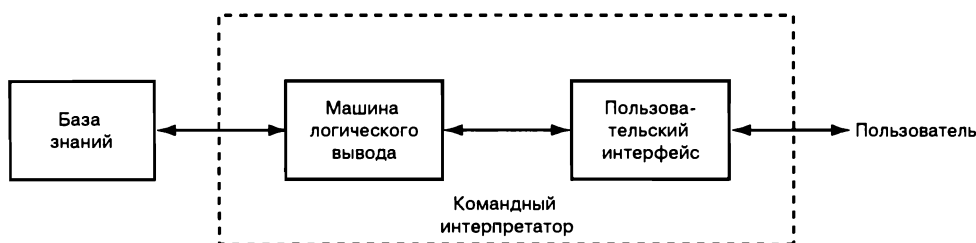


Рис. 15.1. Структура типичной экспертной системы

База знаний содержит сведения, которые относятся к рассматриваемой прикладной области, в том числе такие информационные компоненты, как простые факты об этой области, правила или ограничения, которые описывают отношения, или феномены в этой области и возможно также методы, эвристики и идеи для решения задач в данной области. Машина логического вывода обладает способностью активно использовать знания, представленные в базе знаний. Пользовательский интерфейс обеспечивает бесперебойное взаимодействие пользователя и системы, а также дает возможность пользователю получить представление о том, как выполняется процесс решения задачи, осуществляемый машиной логического вывода. Машину логическо-

го вывода и пользовательский интерфейс удобно рассматривать как один модуль, который обычно называют *командным интерпретатором экспертной системы*, или для краткости просто *командным интерпретатором*.

В описанной выше схеме экспертной системы предусматривается разделение знаний и алгоритмов, с помощью которых используются эти знания. Такое разделение удобно по следующим причинам. С одной стороны, очевидно, что состав базы знаний зависит от приложения. С другой стороны, командный интерпретатор, по крайней мере в принципе, не зависит от проблемной области. Поэтому, если должны быть разработаны экспертные системы для нескольких приложений, целесообразно сначала создать командный интерпретатор универсального назначения, а затем подключить к нему новую базу знаний по мере разработки каждого нового приложения. Безусловно, при таком подходе все базы знаний должны соответствовать одним и тем же формальным требованиям, которые совместимы с командным интерпретатором. Но практический опыт создания сложных экспертных систем показывает, что подход, в котором используются один командный интерпретатор и много разных баз знаний, удается реализовать без особых затруднений, только если прикладные области действительно очень похожи друг на друга. Тем не менее, даже если и потребуются вносить изменения в командный интерпретатор при переходе от одной проблемной области к другой, то по крайней мере могут быть сохранены основные принципы формирования экспертных систем.

В данной главе описаны некоторые основные методы функционирования экспертных систем. В частности, здесь рассматриваются способы представления знаний с помощью правил вывода (правил "if-then"), главные механизмы логического вывода в экспертных системах на основе правил (такие как прямой или обратный логический вывод), усовершенствование представления на основе правил с учетом неопределенности, байесовские сети доверия, семантические сети и средства представления знаний с помощью фреймов.

## 15.2. Представление знаний с помощью правил вывода

В принципе для использования в экспертной системе можно взять любую непротиворечивую формальную систему, позволяющую представлять знания о некоторой проблемной области. Но по своей популярности язык *правил вывода*, называемых также *порождающими правилами* (или *продукциями*), намного превосходит другие формальные системы представления знаний. В целом такие правила представляют собой условные выражения, но они могут иметь разные интерпретации, например, как показано ниже.

- if (если) предварительное условие P then (то) заключение C;
- if ситуация S then действие A;
- if соблюдаются условия C1 и C2 then условие C не соблюдается.

Обычно правила вывода оказываются наиболее естественной формой представления знаний, а также обладают перечисленными ниже дополнительными положительными свойствами.

- **Модульность.** Каждое правило определяет небольшой, относительно независимый фрагмент знаний.
- **Наращиваемость.** Новые правила могут вводиться в базу знаний относительно независимо от других правил.
- **Модифицируемость (как следствие модульности).** Существующие правила можно изменять относительно независимо от других правил.
- **Прозрачность.** Использование правил обеспечивает прозрачность системы.

Последнее свойство является одним из важных отличительных свойств экспертных систем. Под *прозрачностью системы* подразумевается ее способность объяснять свои решения и заключения. Правила вывода позволяют легко найти ответ на описанные ниже основные типы вопросов пользователя.

1. Вопросы, в ответ на которые необходимо предоставить объяснение последовательности рассуждений: “Как было получено это заключение?”
2. Вопросы, в ответ на которые необходимо предоставить объяснение предпосылок конкретного действия: “Для чего потребовалась эта информация?”

Механизмы получения ответов на такие вопросы, основанные на использовании правил вывода, рассматриваются ниже.

Правила вывода часто определяют логические отношения между понятиями в проблемной области. Чисто логические отношения можно охарактеризовать как принадлежащие к *категорическим знаниям*. Под словом “категорический” подразумевается, что эти знания всегда рассматриваются как абсолютно истинные. Но в некоторых областях, таких как медицинская диагностика, в основном преобладают *неполные или вероятностные знания*. Они являются “неполными” в том смысле, что установленные опытным путем закономерности обычно являются истинными только до определенной степени (они чаще всего оправдываются, но не всегда). В таких случаях правила вывода можно изменить, введя оценку вероятности в их логическую интерпретацию, например, следующим образом:

if условие A then следует заключение B с достоверностью F

Программы, приведенные в листингах 15.1–15.3, позволяют понять, какие разнообразные способы представления знаний могут быть реализованы с помощью правил вывода. В них показаны примеры правил из трех различных систем, основанных на знаниях: система MYCIN для медицинских консультаций, система AL/X для диагностики неисправности оборудования и система AL3 для решения задач в шахматах.

#### Листинг 15.1. Одно из правил вывода из системы MYCIN для медицинских консультаций [148].

Параметр 0,7 указывает, до какой степени можно доверять этому правилу

if

1. инфекция относится к типу первичной бактериемии, и
2. культура локализуется на одном из стерильных участков, и
3. есть основания полагать, что путь проникновения инфекции в организм – желудочно-кишечный тракт

then

эти факты могут (с достоверностью 0,7) свидетельствовать о том, что рассматриваемый микроорганизм является бактериальным.

#### Листинг 15.2. Два правила из демонстрационной базы знаний AL/X для диагностики неисправностей [130]. Здесь n и s – критерии “необходимости” и “достаточности”. Критерий s оценивает, до какой степени условная часть правила является достаточной для вывода части с заключением. Критерий n оценивает, до какой степени условная часть является необходимой для того, чтобы заключение было истинным

if

давление в клапане на участке V-01 достигло уровня срабатывания перепускного клапана

then

перепускной клапан на участке V-01 сработал [N = 0.005, S = 400]

if

давление на участке V-01 не достигло уровня срабатывания перепускного клапана, и перепускной клапан на участке V-01 сработал

then

перепускной клапан на участке V-01 сработал преждевременно (из-за неконтролируемого изменения уставки давления) [N = 0.001, S = 2000]

```

if
1. есть гипотеза H, что план P ведет к выигрышу, и
2. есть две гипотезы,
 H1, что план R1 опровергает план P, и
 H2, что план R2 опровергает план P, и
3. есть факты, что H1 является ложной и H2 является ложной
then
1. сформировать гипотезу, H3, что объединенный план "R1 or R2"
 опровергает план P, и
2. сформировать факт, что H3 влечет за собой not (H)

```

Вообще говоря, чтобы разработать практически применимую экспертную систему для некоторой выбранной проблемной области, необходимо проконсультироваться со специалистами в этой области и много узнать о ней самому. Получение определенного представления о проблемной области от экспертов и из литературы, а также преобразование этой информации в выбранное формальное представление знаний называется *выявлением знаний*. Это, как правило, сложное занятие, описанию которого мы не можем уделить здесь внимания. Но нам для изучения примеров, приведенных в данной главе, в качестве материала требуются определенная проблемная область и небольшая база знаний. Рассмотрим учебную базу знаний, показанную на рис. 15.2. В ней рассматривается задача определения причин появления утечки воды в квартире. Проблема может возникнуть в ванной или в кухне. В любом случае эта утечка также может вызвать проблемы в гостиной (появление воды на полу). Эта база знаний не только в целом является слишком простой, но и допускает наличие лишь одной неисправности; это означает, что проблема может возникнуть только в ванной или в кухне, но не в обоих этих помещениях одновременно. Рассматриваемая база знаний представлена на рис. 15.2 как сеть логического вывода. Узлы в этой сети соответствуют высказываниям, а связи — правилам в базе знаний. Кривые линии, которые соединяют некоторые из связей, обозначают конъюнктивную зависимость между соответствующими высказываниями. Таким образом, правило, касающееся проблемы в кухне, в этой сети формулируется следующим образом:

```

if hall_wet and bathroom_dry then problem_in_kitchen % Если в гостиной вода,
% а в ванной сухо, то неисправность - в кухне

```

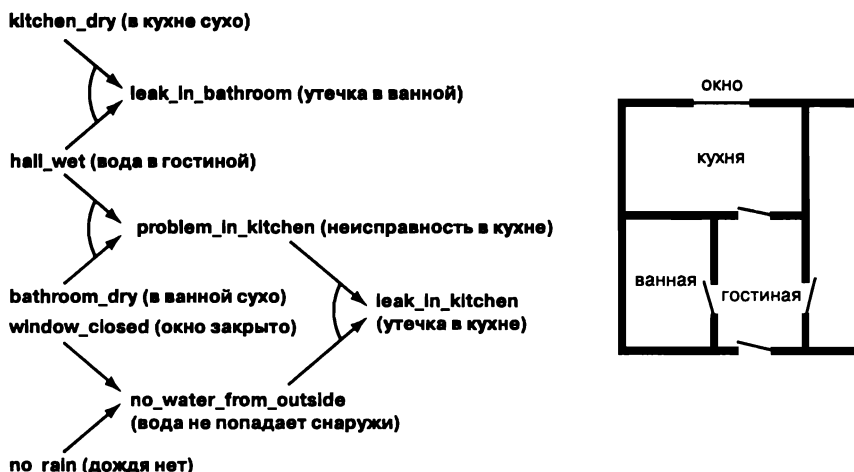


Рис. 15.2. Учебная база знаний для диагностики утечек в водопроводной сети рассматриваемой квартиры

Представление сети, показанное на рис. 15.2, фактически является графом AND/OR, который был описан в главе 13. Это указывает на то, что представление задач в виде графов AND/OR может применяться в контексте экспертных систем, основанных на правилах.

## 15.3. Прямой и обратный логический вывод в системах, основанных на правилах

После представления знаний в определенной форме необходимо предусмотреть использование некоторой процедуры формирования рассуждений для вывода заключений из данной базы знаний. Существуют два основных способа формирования рассуждений на основе правил вывода.

- Обратный логический вывод.
- Прямой логический вывод.

Обе эти процедуры очень легко реализуются на языке Prolog. При этом фактически могут использоваться процедуры поиска в графах AND/OR (см. главу 13). К тому же некоторые более развитые методы поиска в графах AND/OR являются даже более сложными по сравнению с теми, которые обычно применяются в экспертных системах. С другой стороны, в экспертных системах основное внимание уделяется используемому стилю поиска с точки зрения его семантики, применительно к способу рассуждений человека, а это означает, что желательно формировать последовательность рассуждений таким образом, который специалисты в данной проблемной области считают наиболее естественным. Это важно, если в процессе формирования рассуждений осуществляется взаимодействие с пользователем и необходимо, чтобы этот процесс был прозрачным для пользователя. В данном разделе кратко рассматриваются основные процедуры формирования рассуждений на языке Prolog, которые могут применяться в контексте экспертных систем, хотя они по своему назначению аналогичны поиску в графах AND/OR.

### 15.3.1. Обратный логический вывод

На примере базы знаний (см. рис. 15.2) формирование рассуждений в стиле обратного логического вывода может осуществляться следующим образом. Работа по поиску причин появления воды на полу начинается с выдвижения гипотезы (например, `leak_in_kitchen` — утечка в кухне), после этого цепочка рассуждений в сети логического вывода формируется в обратном направлении. Для подтверждения этой гипотезы необходимо, чтобы утверждения `problem_in_kitchen` (неисправность в кухне) и `no_water_from_outside` (вода не поступает снаружи) были истинными. Первое из них подтверждается после проверки того, что на полу в гостиной есть вода, а в ванной — нет. Последнее утверждение можно проверить, убедившись в том, что окно закрыто.

Такой стиль формирования рассуждений называется *обратным логическим выводом*, поскольку в ходе него цепочка правил прослеживается в обратном направлении, от гипотезы (`leak_in_kitchen`) к наблюдаемым фактам (`hall_wet` — вода в гостиной — и т.д.). Такую процедуру можно запрограммировать на языке Prolog очень легко, поскольку она фактически соответствует собственному, встроенному стилю формирования рассуждений языка Prolog. Проще всего можно определить правила базы знаний как правила Prolog следующим образом:

```
leak_in_bathroom :- % Утечка в ванной
hall_wet, % Вода в гостиной
kitchen_dry. % В кухне сухо

problem_in_kitchen :- % Неисправность в кухне
```

```

hall_wet, % Вода в гостиной
bathroom_dry. % В ванной сухо

no_water_from_outside :- % Вода не поступает снаружи
 window_closed % Окно закрыто
;
no_rain. % Дождя нет

leak_in_kitchen :- % Утечка в кухне
 problem_in_kitchen, % Неисправность в кухне
 no_water_from_outside. % Вода не поступает снаружи

```

Наблюдаемые фактические сведения могут быть заданы как факты Prolog следующим образом:

```

hall_wet. % Вода в гостиной
bathroom_dry. % В ванной сухо
window_closed. % Окно закрыто

```

После этого проверку гипотез можно осуществлять, задавая вопросы примерно таким образом:

```

? - leak_in_kitchen. % Утечка в кухне
yes

```

Но использование для представления правил собственного синтаксиса языка Prolog, как в приведенных выше примерах, имеет определенные недостатки, которые описаны ниже.

1. Этот синтаксис может оказаться не самым подходящим для пользователя, незнакомого с языком Prolog; например, может потребоваться, чтобы специалист в проблемной области имел возможность читать правила, задавать новые правила и изменять существующие.
2. При такой организации работы база знаний становится синтаксически не отличимой от остальной программы; может потребоваться проведение более явного различия между базой знаний и остальной частью программы.

Проще всего приспособить синтаксис экспертных правил к нашему вкусу, используя операторную запись языка Prolog. Например, можно выбрать способ использования логических конструкций “if”, “then”, “and” и “or” как операторов, объявленных соответствующим образом, как показано ниже.

```

:- op(800, fx, if).
:- op(700, xfx, then).
:- op(300, xfy, or).
:- op(200, xfy, and).

```

Этого достаточно для того, чтобы можно было записать правила, приведенные в качестве примера на рис. 15.2, следующим образом:

```

if
 hall_wet and kitchen_dry
then
 leak_in_bathroom.

if
 hall_wet and bathroom_dry
then
 problem_in_kitchen.

if
 window_closed or no_rain
then
 no_water_from_outside.

...

```

Предположим, что наблюдаемые факты будут вводиться в качестве заданий для экспертной системы с помощью процедуры fact:



```
fact(hall_wet).
fact(bathroom_dry).
fact(window_closed).
```

Безусловно, теперь потребуется новый интерпретатор для правил, заданных с помощью нового синтаксиса. Такой интерпретатор можно определить как следующую процедуру:

```
is_true(P)
```

Здесь высказывание *P* можно либо найти в процедуре *fact*, либо вывести с помощью правил. Новый интерпретатор правил приведен в листинге 15.4. Следует отметить, что в нем все еще осуществляется обратный логический вывод в форме поиска в глубину. Но теперь этот интерпретатор можно вызвать на выполнение с помощью такого вопроса:

```
?- is_true(leak_in_kitchen).
yes
```

#### Листинг 15.4. Интерпретатор обратного логического вывода для правил вывода

---

```
% Простой интерпретатор обратного логического вывода для правил вывода
```

```
:- op(800, fx, if).
:- op(700, xfx, then).
:- op(300, xfy, or).
:- op(200, xfy, and).
```

```
is_true(P) :-
 fact(P).
```

```
is_true(P) :-
 if Condition then P, % Подходящее правило,
 is_true(Condition). % условие которого является истинным
```

```
is_true(P1 and P2) :-
 is_true(P1),
 is_true(P2).
```

```
is_true(P1 or P2) :-
 is_true(P1)
 ;
 is_true(P2).
```

---

Основным практическим недостатком простых процедур логического вывода, рассматриваемых в этом разделе, является то, что пользователь должен задавать всю необходимую информацию в виде фактов заранее, до начала процесса формирования рассуждений. Это означает, что пользователь может передать системе слишком много или слишком мало информации. Поэтому было бы лучше, чтобы информация предоставлялась пользователем интерактивно, в виде диалога, когда это потребуется. Такие средства ведения диалога будут реализованы в виде программы в главе 16.

### 15.3.2. Прямой логический вывод

При обратном логическом выводе работа начинается с гипотез (например, о том, что утечка — в кухне) и происходит в обратном направлении согласно правилам в базе знаний, в целях получения легко подтверждаемых фактов (допустим, таких, что в гостиной вода). Но иногда более естественным способом является формирование рассуждений в противоположном направлении, от части “if” к части “then”. Прямой логический вывод начинается не с гипотез, а с некоторых подтвержденных фактов. Обнаружив, что в гостиной вода, а в ванной сухо, можно сделать вывод, что неисправность в кухне; кроме того, заметив, что окно кухни закрыто, можно сделать

заключение, что вода не поступает снаружи; это ведет к окончательному выводу, что утечка в кухне.

Реализации в программе Prolog простой процедуры прямого логического вывода все еще остается простой, хотя и не столь тривиальной, как и реализация процедуры обратного логического вывода. В листинге 15.5 показан интерпретатор прямого логического вывода, в котором предполагается, что правила, как и прежде, представлены в следующей форме:

```
if Condition then Conclusion
```

В качестве условия (Condition) здесь может быть задано любое выражение AND/OR. Для простоты во всей данной главе предполагается, что правила не содержат переменных. Интерпретатор начинает свою работу с того, что известно (задано в отношении fact), выводит все заключения (Conclusion), которые следуют из этого отношения, и добавляет (с помощью предиката assert) эти заключения к отношению fact. Поэтому база знаний, рассматриваемая в качестве примера, обрабатывается этим интерпретатором следующим образом:

```
?- forward.
Derived: problem_in_kitchen
Derived: no_water_from_outside
Derived: leak_in_kitchen
No more facts
```

### Листинг 15.5. Интерпретатор правил прямого логического вывода

% Простой интерпретатор прямого логического вывода на языке Prolog

```
forward :-
 new_derived_fact(P), % Новый факт
 !,
 write('Derived: '), write(P), nl,
 assert(fact(P)),
 forward % Продолжить работу
 ;
 write('No more facts'). % Процесс порождения правил завершен

new_derived_fact(Concl) :-
 if Cond then Concl, % Некоторое правило
 not fact(Concl), % Заключение правила - еще не факт
 composed_fact(Cond). % Условие истинно?

composed_fact(Cond) :-
 fact(Cond). % Простой факт

composed_fact(Cond1 and Cond2) :-
 composed_fact(Cond1),
 composed_fact(Cond2). % Оба конъюнкта истинны

composed_fact(Cond1 or Cond2) :-
 composed_fact(Cond1)
 ;
 composed_fact(Cond2).
```

## 15.3.3. Сравнение прямого и обратного логического вывода

Как показано на рис. 15.2, правила вывода образуют цепочки, по которым переход осуществляется слева направо. Элементы в левой части этих цепочек представляют собой входную информацию, а в правой части — производную информацию:

входная информация → ... → производная информация

Эти два вида информации могут иметь разные названия, в зависимости от контекста, в котором они используются. Входная информация может называться *данными* (например, данные измерений), или *фактами*, или *проявлениями*. Производная информация может называться *доказываемыми гипотезами*, или *причинами проявлений*, или *диагнозами*, или *объяснениями*, которые позволяют трактовать имеющиеся факты. Поэтому цепочки этапов вывода соединяют информацию различных типов следующим образом:

данные → ... → цели

свидетельства → ... → гипотезы

факты, наблюдения → ... → объяснения, диагнозы

проявления → ... → диагнозы, причины

И прямой, и обратный логический вывод требуют поиска, но они отличаются друг от друга по направлению поиска. При обратном логическом выводе поиск происходит от целей к данным, от диагнозов к фактам и т.д. В отличие от этого, при прямом логическом выводе поиск осуществляется от данных к целям, от фактов к объяснениям или диагнозам и т.д. Поскольку обратный логический вывод начинается с целей, принято считать, что он *управляется целями*. Аналогичным образом, поскольку прямой логический вывод начинается с данных, говорят, что он *управляется данными*.

Возникает резонный вопрос о том, что лучше, — прямой или обратный логический вывод. Этот вопрос аналогичен выбору между прямым и обратным поиском в пространстве состояний (см. главу 11). Как и в том случае, ответ на данный вопрос также зависит от конкретной задачи. Если необходимо проверить, является ли истинной некоторая определенная гипотеза, то более естественно формировать логический вывод в обратном направлении, начиная от рассматриваемой гипотезы. С другой стороны, если имеется много конкурирующих гипотез и нет причин начинать с одной, а не с другой, то может оказаться, что лучше сформировать логический вывод в прямом направлении. В частности, прямой логический вывод является более естественным в задачах текущего контроля, в которых непрерывно поступают новые данные и система должна определить, не возникла ли аномальная ситуация; любое изменение во входных данных может распространяться в системе в форме прямого логического вывода для проверки того, не указывает ли данное изменение на некоторое нарушение в контролируемом процессе или на снижение уровня производительности. В выборе между прямым и обратным логическим выводом может также помочь анализ даже самой формы сети правил. Если узлов данных (на левом фланге сети) мало, а целевых узлов (на правом фланге) много, то прямой логический вывод, по-видимому, является более приемлемым, а если количество целевых узлов гораздо меньше по сравнению с узлами данных, то верно обратное.

Реальные экспертные задачи обычно гораздо сложнее по сравнению с описанной и требуют сочетания логического вывода, осуществляемого в обоих направлениях. Например, в медицине некоторые начальные результаты наблюдений над пациентом обычно вызывают формирование рассуждений врача в прямом направлении для выработки некоторых начальных диагностических гипотез. Эти первоначальные гипотезы необходимо подтвердить или опровергнуть с помощью дополнительных фактов, а определение состава требуемых фактов осуществляется в форме обратного логического вывода. В примере, приведенном на рис. 15.2, тот факт, что в гостиной есть вода, может вызвать выполнение этапов логического вывода, показанных на рис. 15.3.

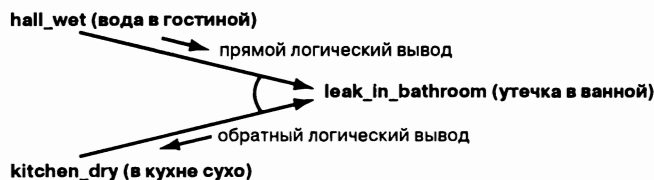


Рис. 15.3. Примеры этапов логического вывода

- 15.1. Напишите программу, в которой объединяется прямой и обратный логический вывод, в стиле, описанном в этом разделе.

## 15.4. Формирование объяснений

Для формирования объяснений в системах на основе правил предусмотрены некоторые стандартные способы. Два широко применяемых типа объяснений получили название *объяснения последовательности рассуждений* и *объяснения предпосылок*, при которых система отвечает на вопросы “как” (“How?”) и “для чего” (“Why?”). Вначале рассмотрим объяснение последовательности рассуждений. После того как система вырабатывает некоторый ответ, пользователь может задать вопрос о том, как был получен этот ответ. Типичное объяснение заключается в том, что пользователю предоставляется трассировка процесса формирования данного ответа. Предположим, что система только что обнаружила, что утечка в кухне и пользователь задает вопрос “How?”. Объяснение может формироваться в соответствии с приведенным ниже примером.

Поскольку:

- 1) неисправность находится в кухне, а вывод об этом сделан на основании того, что в гостиной есть вода, а в ванной сухо; и
- 2) вода не поступает снаружи, и это следует из того факта, что окно закрыто.

Такое объяснение по сути представляет собой дерево доказательства того, что окончательный вывод следует из правил и фактов в базе знаний. Допустим, что знак “<=” определен как инфиксный оператор. В таком случае можно представить дерево доказательства высказывания  $P$  в одной из описанных ниже форм, в зависимости от конкретной ситуации.

1. Если  $P$  — факт, то дерево доказательства —  $P$ .
2. Если факт  $P$  получен с использованием правила

if Cond then  $P$

то дерево доказательства является следующим:

$P \leq \text{ProofCond}$

где  $\text{ProofCond}$  — дерево доказательства условия Cond.

3. Допустим, что  $P_1$  и  $P_2$  — высказывания, деревьями доказательства которых являются  $\text{Proof1}$  и  $\text{Proof2}$ . Если истинно выражение  $P \text{ is } P_1 \text{ and } P_2$ , то дерево доказательства представляет собой конъюнкцию деревьев доказательства  $\text{Proof1}$  и  $\text{Proof2}$ . Если истинно выражение  $P \text{ is } P_1 \text{ or } P_2$ , то дерево доказательства является дизъюнкцией деревьев доказательства  $\text{Proof1}$  и  $\text{Proof2}$ .

Задача формирования деревьев доказательства в программе на языке Prolog решается просто и может быть осуществлена путем модификации предиката `is_true` (см. листинг 15.4), в соответствии с тремя описанными выше ситуациями. Такой модифицированный предикат `is_true` приведен в листинге 15.6. Обратите внимание на то, что деревья доказательства такого рода по существу аналогичны деревьям решения для задач, представленных с помощью графов AND/OR. Процедуру представления деревьев доказательства в некотором удобном для пользователя формате можно реализовать в программе по аналогии с процедурой отображения деревьев решения AND/OR. А процедура, предусматривающая более сложное форматирование вывода деревьев доказательства, входит в состав программы командного интерпретатора, приведенной в главе 16.

В отличие от этого, объяснение предпосылок требуется в течение процесса формирования рассуждений, а не в его конце. Для этого необходимо взаимодействие пользователя с процессом формирования рассуждений. Система запрашивает у пользова-

теля информацию в тот момент, когда ей потребуется данная информация. Получив соответствующий запрос, пользователь может спросить “Why?”, активизировав тем самым подготовку объяснения, для чего нужно данное конкретное действие. Такой характер взаимодействия и способ формирования объяснений в ответ на вопросы “для чего” реализованы в программе, которая входит состав командного интерпретатора, описанного в главе 16.

#### Листинг 15.6. Процедура, в которой формируются деревья доказательства

```
% is_true(P, Proof): Proof - доказательство истинности P

:- op(800, xfx, <=).

is_true(P, P) :-
 fact(P).

is_true(P, P <= CondProof) :-
 if Cond then P,
 is_true(Cond, CondProof).

is_true(P1 and P2, Proof1 and Proof2) :-
 is_true(P1, Proof1),
 is_true(P2, Proof2).

is_true(P1 or P2, Proof) :-
 is_true(P1, Proof)
 ;
 is_true(P2, Proof).
```

## 15.5. Учет неопределенности

### 15.5.1. Простая схема учета неопределенности

В приведенном выше описании применялось представление знаний, в котором подразумевалось, что проблемные области описаны с помощью категорических знаний; это означает, что ответами на любой вопрос является либо “истина”, либо “ложь”, а не что-то среднее между ними. Правила, которые определены как данные (или факты), также являются категорическими: они относятся к типу “категорических импликаций”. Но многие экспертные области не могут быть созданы лишь на основе категорических знаний. Обычно эксперт в своей работе опирается на множество предположений (хотя и четко сформулированных), которые чаще всего являются истинными, но могут быть и исключения. Кроме того, не полностью определенными могут быть и данные о конкретной задаче, и общие правила. Для моделирования неопределенности принимаемым предположениям могут присваиваться некоторые оценки, отличные от “истина” или “ложь”. Такие оценки могут быть выражены с помощью дескрипторов, таких как *истинно*, *весьма вероятно*, *вероятно*, *невероятно*, *невозможно*. Еще один способ состоит в том, что степень доверия может быть выражена с помощью действительного числа из некоторого интервала, например от 0 до 1 или от -5 до +5. Такие числа известны под многими названиями, такими как *коэффициент достоверности*, *мера доверия* или *субъективная вероятность*. Наиболее обоснованный вариант состоит в использовании вероятностей, поскольку такие оценки опираются на солидный математический фундамент. Но для правильного формирования рассуждений с учетом вероятностей (по правилам исчисления вероятностей) обычно требуется гораздо больше усилий, чем при логическом выводе с помощью более простых, произвольных схем учета неопределенности.

Применение вероятностей в связи с использованием байесовских сетей доверия рассматривается в следующем разделе. А в данном разделе описанное выше представление на основе правил дополняется более простой схемой учета неопределенности, в которой вероятности аппроксимируются лишь очень приближенно. Каждому высказыванию (Proposition) присваивается число от 0 до 1, которое служит оценкой его достоверности (CertaintyFactor). В настоящем разделе для представления такой пары применяется следующая форма:

Proposition:CertaintyFactor

Такая система обозначений применяется и к правилам. Поэтому приведенная ниже форма определяет и правило, и степень уверенности в том, что это правило является действительным.

if Condition then Conclusion:Certainty.

При любом представлении данных с учетом неопределенности необходимо предусмотреть способ комбинирования достоверностей высказываний и правил. Например, предположим, что имеются два высказывания, P1 и P2, которые имеют достоверности C1 и C2. Какова достоверность логических комбинаций P1 and P2, P1 or P2? Ниже показана простая схема комбинирования достоверностей. Предположим, что P1 и P2 — высказывания, а c(P1) и c(P2) обозначают их достоверность. В таком случае имеет место следующее:

$c(P1 \text{ and } P2) = \min(c(P1), c(P2))$   
 $c(P1 \text{ or } P2) = \max(c(P1), c(P2))$

А если имеется правило

if P1 then P2:C

то применяется следующее выражение:

$c(P2) = c(P1) * C$

Для простоты предполагается, что в одном и том же утверждении никогда не используется больше одного правила. А если в некотором утверждении в базе знаний имеются два правила, их можно преобразовать с помощью оператора or в эквивалентные правила, которые соответствуют данному утверждению. В листинге 15.7 приведен интерпретатор правил для рассматриваемой схемы учета неопределенности. В этой программе интерпретатора предполагается, что пользователь задает оценки достоверности для фактов (крайних левых узлов в сети правил) с помощью следующего отношения:

given( Proposition, Certainty)

Теперь мы можем “смягчить” (сделать не столь категоричными) некоторые правила в базе знаний (см. рис. 15.2), например, как показано ниже.

```
if
 hall_wet and bathroom_dry
then
 problem_in_kitchen : 0.9.
```

Ситуация, в которой в гостиной вода, в ванной сухо, в кухне не сухо, окно не закрыто, и дождя, по-видимому, не было, но в этом нет полной уверенности, может быть представлена следующим образом:

```
given(hall_wet, 1). % В гостиной вода
given(bathroom_dry, 1). % В ванной сухо
given(kitchen_dry, 0). % В кухне не сухо
given(no_rain, 0.8). % Вероятно, дождя не было, но полной уверенности нет
given(window_closed, 0). % Окно не закрыто
```

### Листинг 15.7. Интерпретатор правил с оценками достоверности

```
% Интерпретатор правил с учетом неопределенности

% certainty(Proposition, Certainty)
```

```

certainty(P, Cert) :-
 given(P, Cert).

certainty(Cond1 and Cond2, Cert) :-
 certainty(Cond1, Cert1),
 certainty(Cond2, Cert2),
 min(Cert1, Cert2, Cert).

certainty(Cond1 or Cond2, Cert) :-
 certainty(Cond1, Cert1),
 certainty(Cond2, Cert2),
 max(Cert1, Cert2, Cert).

certainty(P, Cert) :-
 if Cond then P : C1,
 certainty(Cond, C2),
 Cert is C1 * C2.

```

---

Теперь можно задать системе вопрос об утечке в кухне следующим образом:

```

?- certainty(leak_in_kitchen, C).
C = 0.8

```

Этот вывод получен следующим образом. Тот факт, что в гостиной вода и в ванной сухо, указывает на неисправность в кухне с достоверностью 0,9. Поскольку не исключена возможность, что был дождь, достоверность факта `no_water_from_outside` (вода не поступала снаружи) равна 0,8. Наконец, достоверность факта `leak_in_kitchen` (утечка в кухне) равна минимальному из этих двух значений:

```

min(0.8, 0.9) = 0.8

```

## 15.5.2. Сложности, связанные с учетом неопределенности

Проблемы, связанные с обработкой не полностью определенных знаний, очень широко исследовались и обсуждались. При этом основной темой дискуссий была применимость теории вероятностей при представлении знаний с учетом неопределенности в экспертных системах, с одной стороны, и недостатки произвольных схем учета неопределенности, с другой. Чрезмерно упрощенный подход, используемый в разделе 15.5.1, является примером применения произвольной схемы учета неопределенности и может быть легко подвергнут критике. Например, предположим, что достоверность высказывания *a* равна 0,5, а высказывания *b* — 0. В таком случае в применяемой схеме достоверность выражения *a or b* равна 0,5. Теперь допустим, что достоверность *b* возрастает до 0,5. В данной схеме это изменение вообще не повлияет на достоверность *a or b*, что противоречит здравому смыслу.

Специалисты предложили, применили в своей работе и исследовали многие схемы учета неопределенности. Наиболее распространенный недостаток таких схем обычно связан с игнорированием некоторых зависимостей между высказываниями. Например, предположим, что задано следующее правило:

```

if a or b then c

```

Достоверность высказывания *c* должна зависеть не только от достоверности высказываний *a* и *b*, но также от любой корреляции между *a* и *b*; это означает, что должно учитываться, часто ли эти высказывания становятся истинными одновременно и связаны ли они друг с другом какой-то иной зависимостью. Задача полностью обоснованного представления подобных зависимостей может оказаться гораздо сложнее, чем допускает практика, и требует получения информации, которая обычно является недоступной. Поэтому с такими сложностями часто пытаются справиться, принимая предположение о независимости событий, таких как *a* и *b* в приведенном выше правиле. К сожалению, это предположение обычно не оправдано, а в действи-

тельности чаще всего оно оказывается даже ложным, поэтому может приводить к неправильным и противоречивым результатам. Многие специалисты подчеркивают, что подобный отказ от математически обоснованного подхода к трактовке неопределенности в целом может оказаться небезопасным, а другие доказывают, что применяемые при этом решения “оправданы с точки зрения практики”. Наряду с этим можно часто встретить утверждения, что теория вероятностей, хотя и математически обоснована, является неприменимой на практике и фактически неподходящей для решения реальных задач по следующим причинам.

- Создается впечатление, что люди испытывают затруднения, оперируя с фактическими вероятностями; принятые ими оценки достоверности не полностью соответствуют математическим определениям вероятностей.
- Для математически обоснованной трактовки вероятностей требуется либо информация, которую невозможно получить, либо некоторые упрощающие предположения, которые в действительности не совсем оправданы в практическом приложении. К тому же в последнем случае применяемая трактовка снова становится математически необоснованной.

И наоборот, столь же весомые доводы свидетельствуют в пользу математически обоснованных подходов, базирующихся на теории вероятностей. При анализе обоих приведенных выше возражений, касающихся математической вероятности, были получены убедительные результаты, подтверждающие необходимость использования теории вероятностей. В произвольных схемах, которые якобы “работают на практике”, часто обнаруживаются недостатки, являющиеся следствием упрощений, для которых требуются опасные и необоснованные предположения. В следующем разделе приведено вводное описание такого способа представления не полностью достоверных знаний, как байесовские сети доверия, которые обеспечивают правильную трактовку вероятности и в то же время позволяют применять относительно экономичные методы учета зависимостей.

## Упражнение

- 15.2. Предположим, что в некоторой экспертной системе вероятность объединения двух событий оценивается по той же формуле, как и в применяемой в этом разделе простой схеме учета неопределенности, следующим образом:

$$p(A \text{ or } B) \approx \max(p(A), p(B))$$

При каких условиях эта формула дает результаты, правильные с точки зрения теории вероятностей? В какой ситуации результаты, полученные по данной формуле, содержат наибольшую ошибку и какова эта ошибка?

## 15.6. Байесовские сети доверия

### 15.6.1. Вероятности, достоверности и байесовские сети доверия

Для решения проблемы правильной обработки знаний в условиях неопределенности с применением научно обоснованного метода, который одновременно успешно применяется на практике, разработаны *байесовские сети доверия*, называемые также просто *байесовскими сетями*. В этом разделе показано, что такие две характеристики, как научная обоснованность и практическая применимость, являются трудно достижимыми одновременно, но байесовские сети доверия служат хорошим решением. Вначале определим тему для обсуждения.



Предположим, что состояние “мира” определено с помощью вектора переменных, которые принимают значения случайным образом из своих областей определения (из множеств своих допустимых значений). Во всех рассматриваемых примерах описание ограничивается лишь случайными логическими переменными, которые могут принимать значения “истина” или “ложь”. Например, если речь идет об охране жилого дома, то такими переменными являются взлом (обозначаемый как “burglary”) и тревожный сигнал (“alarm”). Переменная “alarm” принимает истинное значение, когда звучит тревожный сигнал, а переменная “burglary” становится истинной после того, как в дом проникли посторонние. В остальных случаях эти переменные являются ложными. Состояние мира, представленного такими переменными, в определенный момент времени можно полностью описать, указав, какие значения имеют в это время все переменные.

Если переменные являются логическими, то изменения их значений вполне целесообразно считать *событиями*. Например, событие “alarm” происходит, если переменная  $alarm = true$ .

Предположим, что за состоянием переменных наблюдает некоторый *агент* (человек или экспертная система). Обычно этот агент не может с полной уверенностью сообщить, является ли та или иная переменная истинной или ложной. Поэтому агент может лишь рассуждать о вероятности того, что переменная является истинной. В этом контексте вероятности используются для оценки степени уверенности агента. Уверенность агента в том, что он действительно обладает правильной информацией, безусловно, определяется тем, каким объемом знаний об этом мире он обладает. Поэтому подобные оценки достоверности называются также *субъективными вероятностями*; под этим подразумевается, что эти вероятности зависят от носителя знаний как от оценивающего их “субъекта”. В таком случае “субъективный” не означает “произвольный”. Хотя эти вероятности моделируют субъективные представления агента о ситуации, они соответствуют исчислению вероятностей.

Введем некоторую систему обозначений. Допустим, что  $X$  и  $Y$  — высказывания; в таком случае имеет место следующее.

- $X \wedge Y$ . Конъюнкция высказываний  $X$  и  $Y$ .
- $X \vee Y$ . Дизъюнкция высказываний  $X$  и  $Y$ .
- $\sim X$ . Отрицание высказывания  $X$ .

Выражение  $p(X)$  обозначает вероятность того, что высказывание  $X$  является истинным, а выражение  $p(X|Y)$  обозначает условную вероятность того, что  $X$  является истинным, при условии, что истинно  $Y$ .

К типичным вопросам о мире, моделируемом таким образом, относится следующий вопрос: “Если дано, что получены значения некоторых переменных, то какова вероятность получения значений некоторых из оставшихся переменных?” Может также рассматриваться такой вопрос: “Если известно, что наблюдались определенные события, то каковы вероятности некоторых других событий?” Например, обнаружено, что звонил тревожный сигнал. Какова вероятность того, что произошел взлом?

Основная сложность состоит в том, что нужно найти способ учета зависимостей между переменными в рассматриваемой задаче. Предположим, что в задаче учитывается  $n$  логических переменных. При этом потребуется  $2^n - 1$  чисел для определения полного распределения вероятностей среди  $2^n$  возможных состояний мира. Поскольку обычно количество переменных достаточно велико, то количество их сочетаний становится слишком большим! Поэтому учет всех возможных состояний становится не только непрактичным и дорогостоящим в реализации с помощью вычислительной техники, но и не допускает возможности применения разумных оценок всех необходимых вероятностей, поскольку отсутствует достаточный объем информации.

Но, как правило, в действительности требуются не все эти вероятности. При использовании полного распределения вероятностей не принимаются какие-либо предположения, касающиеся того, что некоторые переменные являются независимыми

друг от друга. Но обычно такая чрезмерная осторожность не требуется. К счастью, некоторые события действительно полностью независимы друг от друга.

Поэтому, для того чтобы рассматриваемый вероятностный подход стал применимым на практике, следует воспользоваться тем, что некоторые переменные не зависят друг от друга. Таким образом, необходимо применить удобные средства представления зависимостей между переменными и в то же время получить выигрыш (который сводится к снижению сложности) за счет наличия таких событий, которые действительно не зависят друг от друга.

Байесовские сети доверия предоставляют удобный способ определения того, каким образом зависят друг от друга определенные события и какие события являются независимыми. С помощью байесовских сетей доверия эти сведения можно формализовать естественным и понятным способом.

На рис. 15.4 приведен пример байесовской сети с описанием системы охранной тревожной сигнализации. Датчик может сработать при взломе, когда в помещение проникает постороннее лицо, или во время сильной грозы. Предполагается, что датчик активизирует звуковой тревожный сигнал и предупреждающий телефонный звонок. Типичный вопрос, на который подобная байесовская сеть помогает найти ответ, выглядит примерно так: "Предположим, что стоит прекрасная погода и получен тревожный сигнал. Если известны эти два факты, то какова вероятность взлома?"



*Рис. 15.4. Байесовская сеть. После взлома и проникновения в дом постороннего лица, по всей вероятности, происходит активизация датчика. Предполагается, что датчик активизирует звуковой тревожный сигнал и автоматический телефонный звонок с предупреждающим сообщением. Датчик может также сработать под действием сильной грозы*

Структура этой байесовской сети показывает, что некоторые вероятности являются зависимыми, а другие — независимыми. Например, по ней можно судить, что вероятность взлома не зависит от погоды (от грозы). Но если становится известно, что действительно возник тревожный сигнал, то при этом условии вероятность взлома больше не является независимой от грозы.

Интуитивно ясно, что связи в этой сети указывают на причинную зависимость. Взлом является причиной активизации датчика. Датчик, в свою очередь, может вызвать тревожный сигнал. Поэтому структура данной сети позволяет формировать примерно такие рассуждения: если действительно прозвучал тревожный сигнал, то взлом становится вероятным как одна из причин, которыми объясняется появление тревожного сигнала. Если затем обнаруживается, что в это время была сильная гроза, взлом становится менее вероятным. Появление тревожного сигнала можно объяснить также другой причиной, грозой, поэтому вероятность первой возможной причины уменьшается.

В этом примере рассуждения были и диагностическими, и прогностическими: зная о том, что действительно был тревожный сигнал (последствие или признак взлома), мы поставили *диагноз*, что этот сигнал мог быть вызван взломом. Затем мы узнали о грозе и сформулировали *прогноз*, что она также могла вызвать появление тревожного сигнала.

Теперь определим более формально, что именно обозначено связями в байесовской сети и какого рода вероятностные выводы могут быть сделаны с помощью данной байесовской сети.

Вначале необходимо ввести определение, что узел Z является потомком узла X, если согласно ориентированным связям в сети имеется путь от X до Z.

Теперь предположим, что узлы  $Y_1, Y_2, \dots$  являются родительскими узлами узла X в байесовской сети. По определению в байесовской сети подразумевается использование следующего полезного отношения, определяющего вероятностную независимость: узел X не зависит от узлов, не являющихся его потомками, если известны его родительские узлы. Поэтому, чтобы вычислить вероятность X, достаточно принять во внимание вероятности дочерних узлов X и родительских узлов X, —  $Y_1, Y_2$  и т.д. Все возможные влияния других переменных на X можно учесть с помощью родительских узлов X.

Оказалось, что такая трактовка связей в байесовской сети предоставляет практическую возможность, во-первых, определять вероятностные отношения между переменными в моделируемом мире и, во-вторых, отвечать на вопросы об этом мире.

Чтобы понять, каким образом байесовская сеть используется для представления знаний о моделируемом мире, снова рассмотрим пример сети, приведенный на рис. 15.4. Прежде всего, структура этой сети показывает, какие переменные являются зависимыми и независимыми друг от друга.

Кроме того, связи имеют также естественную причинно-следственную интерпретацию. Чтобы уточнить эту интерпретацию, необходимо определить некоторые вероятности, т.е. присвоить им какие-то конкретные числовые значения. Для узлов, которые не имеют родительских узлов (*коренных причин*), задаются априорные вероятности. В данном случае коренными причинами являются взлом и гроза. Для других узлов X необходимо задать условные (апостериорные) вероятности в следующей форме:

$p(X \mid \text{Состояния родительских узлов узла } X)$

Активизация датчика (обозначим это событие сокращенно как *sensor*) имеет две родительские причины: взлом (*burglary*) и гроза (*lightning*). Существуют четыре возможные комбинации состояний этих двух родительских причин: взлом и гроза, взлом и отсутствие грозы и т.д. Эти сочетания состояний можно записать в виде логических формул:  $\text{burglary} \wedge \text{lightning}$ ,  $\text{burglary} \wedge \sim \text{lightning}$  и т.д. Поэтому полная спецификация рассматриваемой байесовской сети может быть представлена следующим образом:

```
p(burglary) = 0.001
p(lightning) = 0.02
p(sensor | burglary ∧ lightning) = 0.9
p(sensor | burglary ∧ ~lightning) = 0.9
p(sensor | ~burglary ∧ lightning) = 0.1
p(sensor | ~burglary ∧ ~lightning) = 0.001
p(alarm | sensor) = 0.95
p(alarm | ~sensor) = 0.001
p(call | sensor) = 0.9
p(call | ~sensor) = 0.0
```

В этой полной спецификации определены 10 вероятностей. Если бы структура данной сети не была задана (т.е. не было указано, какие события являются независимыми), то для полной спецификации потребовалось бы определить 31 вероятность ( $2^5 - 1 = 31$ ), поскольку для мира, моделируемого  $n$  логическими переменными, количество возможных состояний равно  $2^n$ . Поэтому в результате определения структуры этой сети удалось уменьшить количество рассматриваемых комбинаций с 31 до 10. В сети с большим количеством узлов сокращение объема обрабатываемой информации, безусловно, становится еще более значительным.

Степень сокращения объема обработки информации определяется характером конкретной задачи. Если каждая переменная в рассматриваемой задаче зависит от какой-либо другой переменной, то, безусловно, общее количество анализируемых комбинаций сократить невозможно. А если задача допускает сокращение этого количества, то степень сокращения зависит от структуры байесовской сети доверия. Для решения одной и той же задачи могут быть сформированы разные байесовские сети

доверия, причем некоторые сети являются более подходящими по сравнению с другими. Общее правило состоит в том, что в качественно сформированных сетях сохраняются причинные зависимости между переменными. Это означает, что если  $X$  является причиной  $Y$ , то от  $X$  к  $Y$  должна быть проведена ориентированная связь. Например, хотя в проблемной области охраны от взломов возможно сформировать рассуждения от анализа тревожного сигнала к анализу взлома, это может привести к созданию громоздкой сети, если ее разработка начнется с проведения связи от узла, представляющего тревожный сигнал, к узлу, представляющему взлом. При таком подходе в сети придется применить больше связей. К тому же будет сложнее оценить требуемые вероятности в направлении, не обусловленном очевидными причинами, таким как  $p(\text{burglary}|\text{alarm})$ .

## 15.6.2. Некоторые формулы из области исчисления вероятностей

Ниже приведены некоторые формулы из области исчисления вероятностей, которые потребуются при формализации рассуждений в байесовских сетях. Предположим, что  $X$  и  $Y$  — высказывания. Тогда имеют место следующие зависимости:

$$p(\sim X) = 1 - p(X)$$

$$p(X \wedge Y) = p(X) p(Y|X) = p(Y) p(X|Y)$$

$$p(X) = p(X \wedge Y) + p(X \wedge \sim Y) = p(Y) p(X|Y) + p(\sim Y) p(X|\sim Y)$$

Высказывания  $X$  и  $Y$  называются независимыми, если  $p(X|Y) = p(X)$  и  $p(Y|X) = p(Y)$ . Это означает, что знание о том, что высказывание  $Y$  является истинным, не влияет на достоверность высказывания  $X$  и наоборот. Если  $X$  и  $Y$  не зависят друг от друга, то справедлива следующая формула:

$$p(X \wedge Y) = p(X) p(Y)$$

Высказывания  $X$  и  $Y$  являются несовместимыми, если они не могут быть истинными одновременно:  $p(X \wedge Y) = 0$ ,  $p(X|Y) = 0$  и  $p(Y|X) = 0$ .

Допустим, что  $X_1, \dots, X_n$  — высказывания; в таком случае имеет место следующая формула:

$$p(X_1 \wedge \dots \wedge X_n) = p(X_1) p(X_2|X_1) p(X_3|X_1 \wedge X_2) \dots p(X_n|X_1 \wedge \dots \wedge X_{n-1}).$$

Если все  $X_i$  независимы друг от друга, то эту формулу можно упростить таким образом:

$$p(X_1 \wedge \dots \wedge X_n) = p(X_1) p(X_2) p(X_3) \dots p(X_n)$$

Наконец, нам потребуется теорема Байеса, приведенная ниже.

Эта формула, которую можно вывести из приведенной выше формулы конъюнкции,  $p(X \wedge Y)$ , является удобным средством формирования рассуждений о причинах и следствиях. Если взлом рассматривается как причина тревожного сигнала, то вполне естественно перейти к размышлениям о том, как часто тревожные сигналы бывают вызваны взломами. Вероятность этого события выражается как  $p(\text{alarm}|\text{burglary})$ . Но, услышав тревожный сигнал, мы хотим знать вероятность его причины, которая выражается как  $p(\text{burglary}|\text{alarm})$ . В этом может помочь формула Байеса:

$$p(\text{burglary} | \text{alarm}) = p(\text{burglary}) \frac{p(\text{alarm} | \text{burglary})}{p(\text{alarm})}$$

В одном из вариантов теоремы Байеса учитываются также априорные знания  $B$ . Это позволяет нам следующим образом рассуждать о вероятности гипотезы  $H$ , если имеется свидетельство  $E$ , при наличии априорных знаний  $B$ :

$$p(H|E \wedge B) = p(H|B) \frac{p(E | E \wedge B)}{p(E | B)}$$

### 15.6.3. Формирование рассуждений в байесовских сетях

В данном разделе приведена реализация программы, применяемой для интерпретации байесовских сетей. Задача состоит в том, чтобы этот интерпретатор при наличии некоторой байесовской сети отвечал на вопросы в следующей форме: “Какова вероятность определенных высказываний, если дана вероятность некоторых других высказываний?” Примеры таких запросов приведены ниже.

```
p(burglary | alarm) = ?
p(burglary ^ lightning) = ?
p(burglary | alarm ^ ~lightning) = ?
p(alarm ^ ~call | burglary) = ?
```

Интерпретатор формирует ответ на любой из этих вопросов, рекурсивно применяя правила, описанные ниже.

**1. Вероятность конъюнкции:**

$$p( X_1 \wedge X_2 \mid \text{Cond}) = p( X_1 \mid \text{Cond}) * p( X_2 \mid X_1 \wedge \text{Cond})$$

**2. Вероятность возможного события:**

$$p(X|Y_1 \wedge \dots \wedge X \wedge \dots) = 1$$

**3. Вероятность невозможного события:**

$$p( X \mid Y_1 \wedge \dots \wedge \sim X \wedge \dots) = 0$$

**4. Вероятность отрицания:**

$$p( \sim X \mid \text{Cond}) = 1 - p(X \mid \text{Cond})$$

**5. Если от некоторого условия Cond зависит вероятность события, дочернего по отношению к событию X, то для определения вероятности события X необходимо использовать теорему Байеса следующим образом:**

Если  $\text{Cond}_0 = Y \wedge \text{Cond}$ , где  $Y$  – событие в байесовской сети,  
дочернее по отношению к событию  $X$ ,  
то  $p(X|\text{Cond}_0) = p(X|\text{Cond}) * p(Y|X \wedge \text{Cond}) / p(Y|\text{Cond})$

**6. Если от некоторого условия Cond не зависит вероятность события, дочернего по отношению к событию X, то могут иметь место следующие случаи:**

- а) если событие X не имеет родительских событий, то  $p(X|\text{Cond}) = p(X)$ , при условии, что известна вероятность  $p(X)$ ;**
- б) если событие X имеет родительские события Parents, состояния которых определяются отношением possible\_states, то**

$$p(X|\text{Cond}) = \sum_{S \in \text{possible\_states}(\text{Parents})} p(X|S) p(S|\text{Cond})$$

В качестве примера рассмотрим следующий вопрос: “Какова вероятность взлома, если известно, что прозвучал тревожный сигнал?”

```
p(burglary|alarm) = ?
```

В соответствии с приведенным выше правилом 5:

$$p(\text{burglary}|\text{alarm}) = p(\text{burglary}) * p(\text{alarm}|\text{burglary}) / p(\text{alarm})$$

В соответствии с правилом 6:

$$p(\text{alarm}|\text{burglary}) = p(\text{alarm}|\text{sensor}) p(\text{sensor}|\text{burglary}) + p(\text{alarm}|\sim\text{sensor}) p(\sim\text{sensor}|\text{burglary})$$

В соответствии с правилом 6:

$$p(\text{sensor}|\text{burglary}) = p(\text{sensor}|\text{burglary} \wedge \text{lightning}) p(\text{burglary} \wedge \text{lightning}|\text{burglary}) + p(\text{sensor}|\sim\text{burglary} \wedge \text{lightning})$$

$$p(\sim \text{burglary} \wedge \text{lightning} | \text{burglary}) + p(\text{sensor} | \text{burglary} \wedge \sim \text{lightning}) p(\text{burglary} \wedge \sim \text{lightning} | \text{burglary}) + p(\text{sensor} | \sim \text{burglary} \wedge \sim \text{lightning}) p(\sim \text{burglary} \wedge \sim \text{lightning} | \text{burglary})$$

Применив несколько раз правила 1–4 и используя условные вероятности, заданные в сети, получим следующее:

$$p(\text{sensor} | \text{burglary}) = 0.9 * 0.02 + 0 + 0.9 * 0.98 + 0 = 0.9$$

$$p(\text{alarm} | \text{burglary}) = 0.95 * 0.9 + 0.001 * (1 - 0.9) = 0.8551$$

Применив несколько раз правила 1, 4, 6, получим:

$$p(\text{alarm}) = 0.00467929$$

Наконец, будет получен следующий результат:

$$p(\text{burglary} | \text{alarm}) = 0.001 * 0.8551 / 0.00467929 = 0.182741$$

Программа, проводящая рассуждения в соответствии с описанными выше принципами, приведена в листинге 15.8. Конъюнкция высказываний  $X_1 \wedge X_2 \wedge \dots$  представлена в виде списка высказываний  $[X_1, X_2, \dots]$ . Отрицание  $\sim X$  выражается с помощью термина `not X` языка Prolog. Основным предикатом этой программы является следующий:

```
prob(Proposition, Cond, P)
```

где  $P$  — условная вероятность высказывания `Proposition` при условии `Cond`. В программе предполагается, что байесовская сеть представлена с помощью описанных ниже отношений.

- `parent( ParentNode, Node)`. Определяет структуру сети.
- `p( X, ParentsState, P)`. Представляет вероятность события, имеющего родительские события;  $P$  — условная вероятность некорневого узла  $X$  в зависимости от заданного состояния родительских событий, `ParentsState`.
- `p( X, P)`. Представляет вероятность события, не имеющего родительских событий;  $X$  — корневой узел,  $P$  — его вероятность.

В листинге 15.9 приведен пример определения с помощью этих предикатов байесовской сети, показанной на рис. 15.4. С программой обработки байесовской сети, показанной в листинге 15.8, в которую введено определение сети из листинга 15.9, возможен приведенный ниже диалог. Предположим, что получен предупреждающий телефонный звонок, поэтому требуется узнать вероятность взлома:

```
?- prob(burglary, [call], P).
P = 0.232137
```

Затем стало известно, что в это время была сильная гроза, поэтому запрос принял следующий вид:

```
?- prob(burglary, [call, lightning], P).
P = 0.00892857
```

## Листинг 15.8. Интерпретатор байесовских сетей доверия

```
% Формирование рассуждений в байесовских сетях доверия

% Байесовская сеть доверия представлена приведенными ниже отношениями.
% parent(ParentNode, Node)
% p(Node, ParentStates, Prob)
% Prob - условная вероятность узла Node при заданных значениях
% родительских переменных ParentStates, например:
% p(alarm, [burglary, not earthquake], 0.99)
% p(Node, Prob)
% вероятность узла, не имеющего родительских узлов

% prob(Event, Condition, P):
```

```

% вероятность события Event при условии Condition равна P;
% Event - переменная, ее отрицание или список простых событий,
% представленный в виде их конъюнкции

prob([X | Xs], Cond, P) :- !, % Вероятность конъюнкции
 prob(X, Cond, Px),
 prob(Xs, [X | Cond], PRest),
 P is Px * PRest.

prob([], _, 1) :- !. % Пустая конъюнкция

prob(X, Cond, 1) :-
 member(X, Cond), !. % Из условия Cond следует X

prob(X, Cond, 0) :-
 member(not X, Cond), !. % Из условия Cond следует, что X - ложно

prob(not X, Cond, P) :- !, % Вероятность отрицания
 prob(X, Cond, P0),
 P is 1 - P0.

% Применить закон Байеса, если условие распространяется на дочерние узлы узла X
prob(X, Cond0, P) :-
 delete(Y, Cond0, Cond),
 predecessor(X, Y), !, % Y - дочерний узел узла X
 prob(X, Cond, Px),
 prob(Y, [X | Cond], PyGivenX),
 prob(Y, Cond, Py),
 P is Px * PyGivenX / Py. % Предполагается, что Py > 0

% Случаи, в которых условие не распространяется на дочерние узлы

prob(X, Cond, P) :-
 p(X, P), !. % X - корневой узел; его вероятность задана

prob(X, Cond, P) :- !,
 findall((CONDi,Pi), p(X,CONDi,Pi), CPlist), % Условия, которые
 % распространяются на родительские узлы
 sum_probs(CPlist, Cond, P).

% sum_probs(CondsProbs, Cond, WeigthedSum):
% CondsProbs - список условий и соответствующих вероятностей,
% WeightedSum - взвешенная сумма вероятностей условий Conds
% при заданном условии Cond

sum_probs([], _, 0).

sum_probs([(CONDi,P1) | CondsProbs], COND, P) :-
 prob(CONDi, COND, PC1),
 sum_probs(CondsProbs, COND, PRest),
 P is P1 * PC1 + PRest.

predecessor(X, not Y) :- !, % Переменная Y, к которой применена
 predecessor(X, Y). % операция отрицания

predecessor(X, Y) :-
 parent(X, Y).

predecessor(X, Z) :-
 parent(X, Y),
 predecessor(Y, Z).

member(X, [X | _]).

```

```
member(X, [_ | L]) :-
 member(X, L).

delete(X, [X | L], L).

delete(X, [Y | L], [Y | L2]) :-
 delete(X, L, L2).
```

---

### Листинг 15.9. Спецификация байесовской сети, показанной на рис. 15.4, которая соответствует требованиям программы в листинге 15.8

---

% Байесовская сеть доверия "sensor"

```
parent(burglary, sensor). % Обычно при взломе срабатывает датчик
parent(lightning, sensor). % Датчик может сработать при сильной грозе
parent(sensor, alarm).
parent(sensor, call).

p(burglary, 0.001).
p(lightning, 0.02).
p(sensor, [burglary, lightning], 0.9).
p(sensor, [burglary, not lightning], 0.9).
p(sensor, [not burglary, lightning], 0.1).
p(sensor, [not burglary, not lightning], 0.001).
p(alarm, [sensor], 0.95).
p(alarm, [not sensor], 0.001).
p(call, [sensor], 0.9).
p(call, [not sensor], 0.0).
```

---

Поскольку предупреждающий звонок можно объяснить сильной грозой, взлом становится гораздо менее вероятным. Но если погода была прекрасной, взлом становится вполне вероятным, как показано ниже.

```
?- prob(burglary, [call, not lightning], P).
P = 0.473934
```

Следует отметить, что при разработке показанной здесь реализации интерпретатора байесовских сетей была поставлена задача создать короткую и понятную программу. Поэтому полученная программа отличается довольно низкими эксплуатационными характеристиками. При ее использовании для обработки небольших байесовских сетей, подобных той, которая определена в листинге 15.9, проблемы не возникают, но становятся заметными при обработке более крупных сетей. Однако более эффективная реализация была бы гораздо сложнее.

## 15.7. Семантические сети и фреймы

В данном разделе рассматриваются еще две формальные структуры, широко применяемые для представления знаний: семантические сети и фреймы. Они отличаются от представлений, основанных на правилах, тем, что предназначены для представления некоторым структурированным способом больших множеств фактов. Множество фактов является структурированным и, возможно, сжатым. При сжатии факты удаляются, если они могут быть реконструированы с помощью логического вывода. И в семантических сетях, и во фреймах используется механизм наследования по принципу, аналогичному применяемому в объектно-ориентированном программировании.

Семантические сети и фреймы могут быть легко реализованы на языке Prolog. По сути, такая реализация сводится к неуклонному применению определенных стилей программирования и фиксированной организации программы.



## 15.7.1. Семантические сети

Семантическая сеть состоит из сущностей и отношений между ними. Семантические сети принято представлять в виде графов. Применяется много разных типов семантических сетей, и для их представления используются разные соглашения, но обычно узлы в графе соответствуют сущностям, а отношения изображаются как связи, обозначенные именами отношений. Подобная сеть показана на рис. 15.5. Имя отношения *isa* представляет собой сокращение от “is a” (является). В этой сети представлены следующие факты.

- A bird is a kind of animal (Птицы относятся к царству животных).
- Flying is the normal moving method of birds (Полет — типичный способ передвижения птиц).
- An albatross is a bird (Альбатрос — птица).
- Albert is an albatross, and so is Ross (Альберт — альбатрос, таковым же является Росс).

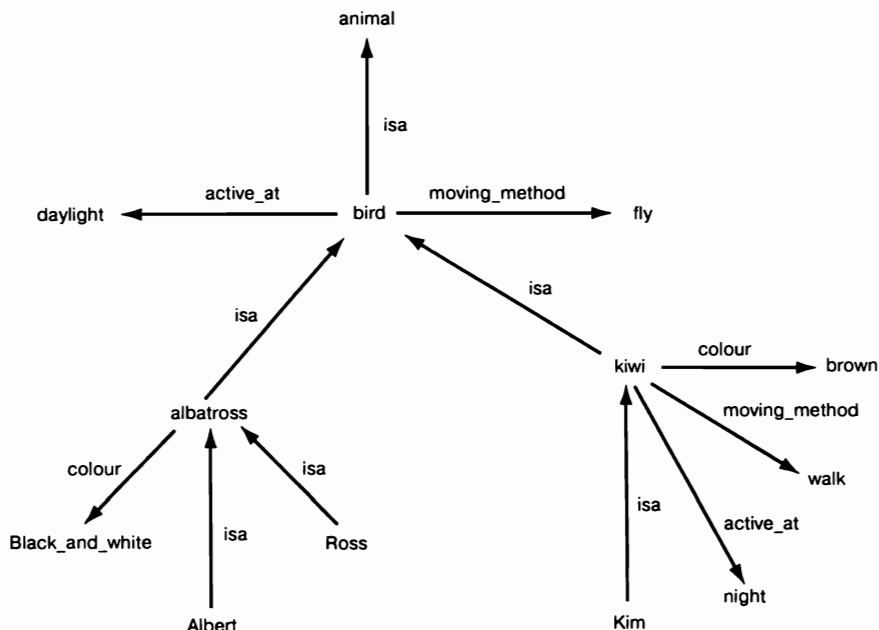


Рис. 15.5. Пример семантической сети

Обратите внимание, что отношение принадлежности (обозначается как “isa”) иногда связывает класс объектов с суперклассом этого класса (животные — это суперкласс класса птиц), а иногда — экземпляр класса с самим классом (Альберт является альбатросом).

Данные, представленные с помощью сети такого рода, можно немедленно перевести на язык фактов Prolog, например, следующим образом:

```
isa(bird, animal). % Птица - это животное
isa(ross, albatross). % Росс - альбатрос
moving_method(bird, fly). % Способ передвижения птиц - полет
moving_method(kiwi, walk). % Способ передвижения киви - ходьба
```

Кроме явно указанных фактов, на основании информации в этой сети могут быть выведены некоторые другие факты. Способы вывода других фактов встраиваются в

средства представления семантической сети конкретного типа как часть этого представления. К типичному встроенному способу вывода относится наследование. Поэтому в сети, приведенной на рис. 15.5, тот факт, что “альбатрос летает”, может быть унаследован от факта “птицы летают”. Аналогичным образом с помощью наследования могут быть получены факты “Росс летает” и “Ким ходит”. Факты наследуются с использованием отношения *isa*. На языке Prolog можно указать следующим образом, что способ передвижения наследуется:

```
moving_method(X, Method) :-
 isa(X, SuperX), % Восхождение по иерархии isa
 moving_method(SuperX, Method).
```

Но если бы для каждого отношения пришлось задавать отдельное правило наследования, то программа стала бы довольно громоздкой. Поэтому проще задать следующим образом более общее правило, касающееся фактов: “Факты могут быть либо явно заданы в сети, либо унаследованы”:

```
fact(Fact) :- % Факт - не переменная; Fact = Rel(Arg1, Arg2)
 Fact, !. % Факт задан в сети явно - наследование не требуется
fact(Fact) :-
 Fact =.. [Rel, Arg1, Arg2],
 isa(Arg1, SuperArg), % Восхождение по иерархии isa
 SuperFact =.. [Rel, SuperArg, Arg2],
 fact(SuperFact).
```

Теперь программе обработки этой семантической сети можно задать некоторые вопросы, например, следующим образом:

```
?- fact(moving_method(kim, Method)).
Method = walk
```

Этот факт унаследован от явно заданного факта, что киви не летают, а ходят. С другой стороны, ответ на следующий вопрос является совсем иным:

```
?- fact(moving_method(albert, Method)).
Method = fly
```

Этот факт был унаследован от класса *bird*. Обратите внимание на то, что унаследованным становится тот факт, который был получен первым при перемещении вверх по иерархии *isa*.

## 15.7.2. Фреймы

В представлении знаний в виде фреймов факты ассоциируются с объектами. В этом контексте под “объектом” подразумевается конкретный физический объект либо более абстрактное понятие, такое как класс объектов или ситуация. Например, в виде фреймов удобно представить типичную ситуацию обмена мнениями или ситуацию игрового конфликта. Подобные ситуации, как правило, имеют некоторую общую, стереотипную структуру, которая может быть дополнена сведениями о конкретном событии.

*Фрейм* — это структура данных, компоненты которой называются *слотами*. Слоты обозначаются именами и применяются для размещения в них информации различного рода. Поэтому в слотах можно найти элементарные значения, ссылки на другие фреймы или процедуры, с помощью которых на основе какой-то иной информации может быть вычислено значение слота. Слот может также оставаться незаполненным. Незаполненные слоты могут заполняться с применением логического вывода. Как и в семантических сетях, наиболее широко применяемым принципом логического вывода является наследование. Если фрейм представляет собой класс объектов (таких как альбатросы), а другой фрейм — суперкласс этого класса (такой как птицы), то фрейм класса может наследовать значения от фрейма суперкласса.

Некоторые знания о птицах можно поместить во фреймы, например, следующим образом:

```
FRAME: bird
a_kind_of: animal
moving_method: fly
active_at: daylight
```

Этот фрейм содержит сведения о классе всех птиц. Три его слота сообщают о том, что птицы относятся (`a_kind_of`) к животным (животные — это суперкласс птиц), что типичным способом передвижения птиц (`moving_method`) является полет и что птицы являются активными (`active_at`) в дневное время. Ниже приведены фреймы для двух подклассов птиц — альбатросов и киви.

```
FRAME: albatross
a_kind_of: bird
colour: black_and_white
size: 115
```

```
FRAME: kiwi
a_kind_of: bird
moving_method: walk
active_at: night
colour: brown
size: 40
```

Альбатрос — это очень типичная птица, которая наследует от фрейма птиц способность летать и дневную активность. Поэтому во фрейме `albatross` нет никаких сведений о способе передвижения `moving_method` и периоде активности `active_at`. С другой стороны, киви — это довольно нетипичная птица, поэтому при ее описании необходимо перекрыть обычные значения `moving_method` и `active_at`, характерные для птиц. Может быть также определен конкретный экземпляр класса, например альбатрос по имени Альберт, следующим образом:

```
FRAME: Albert
instance_of: albatross
size: 120
```

Обратите внимание на различие между двумя отношениями, `a_kind_of` и `instance_of`. Первое является отношением между классом и суперклассом, а второе — отношением между членом класса и классом.

Информация о фреймах, применяемых здесь в качестве примера, может быть представлена на языке Prolog в виде множества фактов, таким образом, чтобы один факт определял значение каждого отдельного слота. Такую задачу можно выполнить несколькими способами. А в данном разделе используется следующий формат для представления таких фактов:

```
Frame_name(Slot, Value)
```

Преимущество этого формата состоит в том, что теперь все факты о конкретном фрейме собраны в виде отношения, имя которого совпадает с именем самого фрейма. В листинге 15.10 показаны некоторые фреймы в таком формате.

#### Листинг 15.10. Некоторые примеры фреймов

```
% фрейм представлен в виде множества фактов Prolog:
% frame_name(Slot, Value)
% Здесь Value - либо простое значение, либо процедура

% фрейм bird - определение типичной птицы

bird(a_kind_of, animal).
bird(moving_method, fly).
bird(active_at, daylight).

% фрейм albatross; альбатрос - это типичная птица, но для его определения
% нужны дополнительные факты: он черно-белый и имеет в среднем длину 115 см

albatross(a_kind_of, bird).
```

```

albatross(colour, black_and_white).
albatross(size, 115).

% Фрейм kiwi; киви - довольно нетипичная птица, поскольку она ходит,
% а не летает, и активна в ночное время

kiwi(a_kind_of, bird).
kiwi(moving_method, walk).
kiwi(active_at, night).
kiwi(size, 40).
kiwi(colour, brown).

% Фрейм albert - экземпляр взрослого альбатроса

albert(instance_of, albatross).
albert(size, 120).

% Фрейм ross - экземпляр альбатроса-птенца

ross(instance_of, albatross).
ross(size, 40).

% Фрейм animal - слот relative_size получает свое значение
% в результате выполнения процедуры relative_size

animal(relative_size, execute(relative_size(Object, Value), Object,
 Value)).

```

---

Для использования подобного набора фреймов необходима процедура, позволяющая осуществлять выборку фактов о значениях слотов. Реализуем такую процедуру выборки фактов как процедуру Prolog следующим образом:

```

value(Frame, Slot, Value)

```

где Value — значение слота Slot во фрейме Frame. Если слот заполнен определенным значением (так что его значение явно задано во фрейме), то оно и принимается в качестве требуемого значения; в противном случае происходит получение нужного значения с помощью логического вывода, например наследования. Чтобы найти значение с помощью наследования, необходимо перейти от текущего фрейма к более общему фрейму в соответствии с отношением a\_kind\_of или instance\_of между фреймами. Такой переход ведет к “родительскому фрейму”, и искомое значение может быть найдено в этом фрейме явно или с помощью дальнейшего наследования. Эта операция непосредственной выборки или выборки путем наследования может быть определена на языке Prolog следующим образом:

```

value(Frame, Slot, Value) :-
 Query =.. [Frame, Slot, Value],
 call(Query), !. % Получение значения с помощью непосредственной выборки
value(Frame, Slot, Value) :-
 parent(Frame, ParentFrame), % Фрейм более общего типа
 value(ParentFrame, Slot, Value).

parent(Frame, ParentFrame) :-
 (Query =.. [Frame, a_kind_of, ParentFrame]
 ;
 Query =.. [Frame, instance_of, ParentFrame]),
 call(Query).

```

Эта процедура вполне позволяет находить значения во фреймах, показанных в листинге 15.10, например, с помощью следующих вопросов:

```

?- value(albert, active_at, AlbertTime).
AlbertTime = daylight
?- value(kiwi, active_at, KiwiTime).
KiwiTime = night

```

Теперь рассмотрим более сложный случай логического вывода, при котором в слоте определена процедура вычисления значения вместо самого значения. Например, в одном из слотов для всех животных может быть задан их размер, установленный с учетом типичного размера взрослого экземпляра особи соответствующего вида. В таком случае относительные размеры двух альбатросов, описанных в листинге 15.10, заданные в процентах, составляют: Альберт — 104%, Росс — 35%. Эти цифры получены как соотношение (в процентах) между размером конкретного индивидуума и типичным размером представителя класса этого индивидуума. Например, для альбатроса Росс это значение получено следующим образом:

```
40/115 * 100% = 34.78%
```

Итак, значение для слота `relative_size` получено путем выполнения некоторой процедуры. Эта процедура является в равной степени применимой для всех животных, поэтому целесообразно определить слот `relative_size` во фрейме `animal` и ввести в этот слот определение соответствующей процедуры. Теперь интерпретатор фреймов `value` должен быть способен ответить на такой вопрос:

```
?- value(ross, relative_size, R).
R = 34.78
```

Способ получения этого результата должен состоять примерно в следующем. Процедура начинает поиск с фрейма `ross` и, не обнаружив значения для `relative_size`, поднимаемся вверх по цепочке отношений `instance_of` (чтобы перейти к фрейму `albatross`), `a_kind_of` (чтобы перейти к фрейму `bird`), и снова `a_kind_of`, чтобы, наконец, дойти до фрейма `animal`. Здесь обнаруживается процедура вычисления относительного размера. Для этой процедуры требуются значения из слота `size` фреймов `ross` и `albatross`. Эти значения могут быть получены с помощью наследования из существующей процедуры `value`. Остается лишь дополнить процедуру `value`, чтобы она учитывала случаи, в которых требуется выполнить процедуру в определенном слоте. Прежде чем это сделать, необходимо продумать, как такая процедура может быть (косвенно) представлена в виде содержимого слота. Предположим, что процедура вычисления относительного размера представлена в виде предиката `Prolog` следующим образом:

```
relative_size(Object, RelativeSize) :-
 value(Object, size, ObjSize),
 value(Object, instance_of, ObjClass),
 value(ObjClass, size, ClassSize),
 RelativeSize is ObjSize/ClassSize * 100. % Процентная доля от размера класса
```

Теперь в слот `relative_size` фрейма `animal` можно ввести вызов этой процедуры. Чтобы предотвратить потерю параметров `Object` (Объект) и `RelativeSize` (Относительный размер) при взаимодействии фреймов, необходимо также определить их как часть информации слота `relative_size`. Все содержимое этого слота можно собрать в один терм `Prolog`, например, как показано ниже.

```
execute(relative_size(Object, RelSize), Object, RelSize)
```

После этого слот `relative_size` фрейма `animal` определяется таким образом:

```
animal(relative_size, execute(relative_size(Obj, Val), Obj, Val)).
```

Теперь можно приступить к модификации процедуры `value` для обработки с ее помощью процедурных слотов. Прежде всего мы должны учесть, что информация, находящаяся в слоте, может представлять собой вызов процедуры, поэтому для ее дальнейшей обработки необходимо осуществить этот вызов. Для такого вызова в качестве параметров могут потребоваться значения слотов первоначально рассматриваемого фрейма. В старой версии процедуры `value` этот фрейм исчезал из памяти после перехода по иерархическому дереву к более общим фреймам. Поэтому необходимо определить первоначальный фрейм в качестве дополнительного параметра. Эта задача решается в следующем фрагменте программы:

```
value(Frame, Slot, Value) :-
 value(Frame, Frame, Slot, Value).
```

```
% Непосредственная выборка информации из слота (супер)фрейма
value(Frame, SuperFrame, Slot, Value) :-
 Query =.. [SuperFrame, Slot, Information],
 call(Query), % Значение, полученное с помощью непосредственной выборки
 process(Information, Frame, Value), !. % Информация может быть задана либо
 % в виде значения, либо как вызов процедуры

% Логический вывод значения с помощью наследования
value(Frame, SuperFrame, Slot, Value) :-
 parent(SuperFrame, ParentSuperFrame),
 value(Frame, ParentSuperFrame, Slot, Value).
% process(Information, Frame, Value)
process(execute(Goal, Frame, Value), Frame, Value) :- !,
 call(Goal).
process(Value, _, Value). % Значение, а не вызов процедуры
```

Введя указанное дополнение в интерпретатор фреймов, мы вступили в область применения такого подхода к программированию, как объектно-ориентированное программирование. Хотя при описании указанного подхода обычно применяется иная терминология, его реализация фактически основана на активизации выполнения процедур, которые принадлежат к разным фреймам.

Применение логического вывода, основанного на переходе между фреймами, характеризуется многими нюансами, которые здесь не рассматриваются. Коснемся лишь вопроса множественного наследования. Необходимость в его использовании возникает, если фрейм имеет больше одного "родительского" фрейма (согласно отношению `instance_of` или `a_kind_of`). При этом унаследованное значение слота может быть получено больше чем от одного родительского фрейма, и возникает вопрос, на каком из них следует остановиться. Процедура `value` в том виде, в каком она определена в этом разделе, просто принимает первое обнаруженное значение, которое найдено с помощью поиска в глубину среди фреймов, обладающих способностью предоставить такое значение. Но могут оказаться более приемлемыми другие стратегии или правила выбора среди нескольких альтернатив.

## Упражнения

### 15.3. Проведите трассировку выполнения следующего запроса:

```
?- value(ross, relative_size, Value).
```

чтобы достичь полного понимания того, как передается информация по сети фреймов в показанном здесь интерпретаторе фреймов.

### 15.4. Допустим, что в виде фреймов представлены геометрические фигуры. Следующие предложения определяют квадрат `s1` и прямоугольник `r2` и задают метод вычисления площади фигуры:

```
s1(instance_of, square).
s1(side, 5).
r2(instance_of, rectangle).
r2(length, 6).
r2(width, 4).
square(a_kind_of, rectangle).
square(length, execute(value(Obj,side,L), Obj, L)).
square(width, execute(value(Obj,side,W), Obj, W)).
rectangle(area, execute(area(Obj,A), Obj, A)).
area(Obj, A) :-
 value(Obj, length, L), value(Obj, width, W),
 A is L*W.
```

Как интерпретатор фреймов, программа которого приведена в этом разделе, ответит на следующий вопрос:

```
?- value(r2, length, A), value(s1, length, B), value(s1, area, C).
```

- К типичным функциям, которые требуются от *экспертной системы*, относятся следующие: решение задач в указанной проблемной области, объяснение процесса решения задачи и работа в условиях недостоверной и неполной информации.
- Любую экспертную систему удобно рассматривать как состоящую из двух модулей: командный интерпретатор и база знаний. *Командный интерпретатор*, в свою очередь, состоит из механизма логического вывода и пользовательского интерфейса.
- Для *создания командного интерпретатора* экспертной системы необходимо принять решения, касающиеся выбора формальной системы представления знаний, механизма логического вывода, средств взаимодействия с пользователем и способа трактовки неопределенности.
- Наиболее широко применяемой формой представления знаний в экспертных системах являются *правила вывода*, или *порождающие правила*.
- В системах, основанных на правилах, главным образом применяются следующие два способа формирования логических рассуждений: *обратный* и *прямой логический вывод*.
- Обычно предусмотрено использование *объяснений* двух типов — объяснение последовательности рассуждений и объяснение предпосылок, которые соответственно связаны с вопросами пользователя “как” и “для чего”. В качестве объяснения первого типа может использоваться дерево доказательства.
- *Проведение рассуждений* с учетом неопределенности можно осуществить с помощью основных средств представления правил и схем прямого или обратного логического вывода. Но при таких дополнениях к правилам обычно принимаются необоснованные предположения, которые чрезмерно упрощают вероятностные зависимости между переменными в проблемной области.
- *Байесовские сети доверия*, называемые также просто *байесовскими сетями*, предоставляют способ использования исчисления вероятностей для представления знаний с учетом неопределенности. Байесовские сети обладают такими характеристиками, как учет вероятностных зависимостей с применением относительно небольшого объема ресурсов, возможность уменьшения объема работы благодаря исключению независимых факторов и естественное представление причинно-следственных связей.
- К другим традиционным схемам представления знаний, подходящим для представления больших множеств фактов некоторым структурированным способом, относятся *семантические сети* и *фреймы*. В них может осуществляться непосредственная выборка фактов или их логический вывод с помощью таких встроенных механизмов, как наследование.
- В этой главе были разработаны *реализации в стиле Prolog* для таких процедур, как прямой и обратный логический вывод, формирование деревьев доказательств, интерпретация правил с учетом неопределенности, формирование рассуждений в байесовских сетях, наследование в семантических сетях и во фреймах.
- В данной главе рассматривались следующие понятия:
  - экспертные системы;
  - база знаний, командный интерпретатор экспертной системы, механизм логического вывода;
  - правила вывода (правила “if-then”), порождающие системы;
  - обратный логический вывод, прямой логический вывод;

- объяснение последовательности рассуждений, объяснение предпосылок;
- категорическое знание, недостоверное знание;
- байесовские сети доверия, или просто байесовские сети;
- семантические сети;
- фреймы;
- наследование.

## Дополнительные источники информации

В [101] представлен один из первых сборников отредактированных статей, в котором рассматриваются различные аспекты экспертных систем и методов представления знаний. Одной из многих книг по экспертным системам ознакомительного характера, которая включает описание некоторых широко известных систем, является [66]. Такие содержательные книги по искусственному интеллекту, как [90], [126] и [133], включают достаточно подробное описание способов представления знаний и формирования рассуждений. Одна из первых экспертных систем, MYCIN, оказавших значительное влияние на развитие этой области, описана в [148], а в [24] приведен сборник статей, касающихся экспериментов с системой MYCIN. Одной из авторитетных книг по формированию рассуждений в экспертных системах с учетом неопределенности и вероятности является [119]. Сборник статей [10] часто упоминается в контексте представления знаний. В [142] приведен сборник важных статей по формированию рассуждений в условиях неопределенности. В [25] и [68] даны основные сведения по байесовским сетям. Классической работой по способам представления знаний в виде фреймов является [103]. В [171] показано, как представление в виде фреймов может использоваться для описания знаний, обоснованных практикой. Различные подходы к представлению знаний для формирования рассуждений в рамках обыденных представлений, в том числе касающихся времени и пространства, описаны в [39]. В [158] приведены результаты изучения проблем наследования. Некоторые примеры, приведенные в данной главе, взяты из [14], [130] и [148].



# Командный интерпретатор экспертной системы

*В этой главе...*

|                                                   |     |
|---------------------------------------------------|-----|
| 16.1. Формат представления знаний                 | 357 |
| 16.2. Проектирование механизма логического вывода | 361 |
| 16.3. Реализация                                  | 366 |
| 16.4. Заключительные замечания                    | 380 |

В данной главе демонстрируется разработка законченного командного интерпретатора экспертной системы на основе правил с использованием метода обратного логического вывода, описанного в главе 15. В этом интерпретаторе реализован механизм доказательства, соответствующий рекомендациям, приведенным в главе 15, а также применены некоторые другие средства, не описанные в вышеупомянутой главе. В рассматриваемой версии интерпретатора разрешено применять переменные в правилах вывода и введено средство, известное под названием *запрос к пользователю*, с помощью которого пользователю передаются приглашения к вводу информации, но только если она действительно требуется в процессе формирования рассуждений.

## 16.1. Формат представления знаний

Рабочий план по созданию командного интерпретатора, который описывается в этой главе, приведен ниже.

1. Выбрать и уточнить формальный способ представления знаний в командном интерпретаторе.
2. Продумать детали реализации механизма логического вывода, который соответствует выбранному способу формального представления.
3. Ввести средства взаимодействия с пользователем.

Начнем с пункта 1 — выбора формального способа представления знаний. В качестве такого способа будут использоваться правила вывода, имеющие синтаксис, аналогичный описанному в главе 15. Но в эти правила потребуется ввести некоторую дополнительную информацию. Например, правилам необходимо присваивать имена. Эти дополнительные сведения можно найти в листинге 16.1, где показана база знаний в формате, выбранном для разрабатываемого командного интерпретатора. Эта база знаний состоит из простых правил, которые позволяют классифицировать животных по их основным характеристикам; при этом предполагается, что в задаче классификации рассматривается лишь несколько видов животных.

**Листинг 16.1. Простая база знаний для классификации животных. Адаптировано из [170].**  
**Отношение "askable" определяет, какие вопросы можно задавать пользователю. Операторы ":", if, then, and, or определены, как показано в листинге 16.3**

---

```
% Небольшая база знаний для классификации животных

:- op(100, xfx, [has, gives, 'does not', eats, lays, isa]).

:- op(100, xf, [swims, flies]).

rule1 :: if
 Animal has hair
 or
 Animal gives milk
then
 Animal isa mammal.

rule2 :: if
 Animal has feathers
 or
 Animal flies and
 Animal lays eggs
then
 Animal isa bird.

rule3 :: if
 Animal isa mammal and
 (Animal eats meat
 or
 Animal has 'pointed teeth' and
 Animal has claws and
 Animal has 'forward pointing eyes')
then
 Animal isa carnivore.

rule4 :: if
 Animal isa carnivore and
 Animal has 'tawny color' and
 Animal has 'dark spots'
 then
 Animal isa cheetah.

rule5 :: if
 Animal isa carnivore and
 Animal has 'tawny color' and
 Animal has 'black stripes'
then
 Animal isa tiger.

rule6 :: if
 Animal isa bird and
 Animal 'does not' fly and
 Animal swims
then
 Animal isa penguin.

rule7 :: if
 Animal isa bird and
 Animal isa 'good flyer'
then
 Animal isa albatross.
```

```

fact :: X isa animal :-
 member(X, [cheetah, tiger, penguin, albatross]).

askable(_ gives _, 'Animal' gives 'What').

askable(_ flies, 'Animal' flies).

askable(_ lays eggs, 'Animal' lays eggs).

askable(_ eats _, 'Animal' eats 'What').

askable(_ has _, 'Animal' has 'Something').

askable(_ 'does not' _, 'Animal' 'does not' fly).

askable(_ swims, 'Animal' swims).

askable(_ isa 'good flyer', 'Animal' isa 'good flyer').

```

---

**Правила в этой базе знаний имеют следующую форму:**

```
RuleName :: if Condition then Conclusion.
```

где Condition (Условие) — множество простых высказываний, объединенных логическими операторами and и or, а Conclusion (Заключение) — простое высказывание, которое не должно содержать логических операторов. Кроме того, в условной части правил разрешается использовать оператор not, хотя и с некоторыми оговорками. В результате применения соответствующих определений операторов Prolog (см. листинг 16.1) эти правила становятся синтаксически допустимыми предложениями Prolog. Оператор and связывает сильнее, чем or, что соответствует обычному соглашению.

Обратите внимание на то, что высказывания в правилах могут представлять собой термы, содержащие переменные. Приведенный ниже пример показывает, почему переменные необходимы и в связи с чем их применение намного увеличивает выразительную мощь системы представления знаний. Рассматриваемый здесь пример базы знаний позволяет находить неисправности в простой электрической сети, которая состоит из некоторых электрических приборов и плавких предохранителей. Пример такой сети показан на рис. 16.1. В качестве одного из правил может служить следующее:

```

if
 light1 is on and % Если светильник 1 включен и
 light1 is not working and % светильник 1 не горит и проверка
 fusel is proved intact % показала, что плавкий предохранитель 1 цел,
then
 light1 is proved broken. % то следует считать, что светильник 1 неисправен

```

Еще одно правило приведено ниже.

```

if
 heater is working % Если нагреватель работает, то следует
then
 fusel is proved intact. % считать, что плавкий предохранитель 1 цел

```

Эти два правила уже опираются на факты (о данной конкретной сети), что прибор light1 соединен с прибором fusel, а приборы light1 и heater подключены к электрической сети с помощью одного и того же плавкого предохранителя. Для другой схемы потребовался бы иной набор правил. Поэтому лучше задавать правила в более общем виде, с использованием переменных Prolog, чтобы их можно было использовать для любой электрической схемы, а затем вводить некоторую дополнительную информацию о конкретной схеме. В таком случае одним из полезных правил может быть следующее: если прибор включен и не работает, а его плавкий пре-

дохранитель цел, то неисправен сам прибор. Такую формулировку можно перевести на применяемый здесь язык формального определения правил следующим образом:

```
broken_rule :: if Device is on and
not (Device is working) and
Device is connected to Fuse and
Fuse is proved intact then
Device is proved broken.
```

База знаний, представленная в этой форме, показана в листинге 16.2.

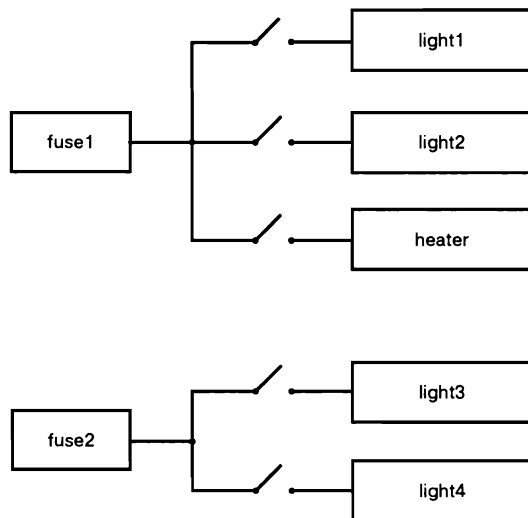


*Рис. 16.1. Соединения между плавкими предохранителями и приборами в простой электрической сети*

#### Листинг 16.2. База знаний для поиска неисправности в электрической схеме, подобной приведенной на рис. 16.1

```
% Небольшая база знаний для поиска неисправности в электрической схеме

% Если прибор включен и не работает, а его плавкий предохранитель цел,
% то прибор неисправен

broken_rule ::
 if
 on(Device) and
 device(Device) and
 (not working(Device)) and
 connected(Device, Fuse) and
 proved(intact(Fuse))
 then
 proved(broken(Device)).

% Если прибор работает, то его плавкий предохранитель цел

fuse_ok_rule ::
 if
 connected(Device, Fuse) and
 working(Device)
 then
 proved(intact(Fuse)).
```

```
% Если к одному плавкому предохранителю подключены два разных прибора, оба они
% включены и не работают, то плавкий предохранитель сгорел. Примечание: при
% этом хотя бы один из приборов должен быть исправным!
```

```
fused_rule ::
 if
 connected(Device1, Fuse) and
 on(Device1) and
 (not working(Device1)) and
 samefuse(Device2, Device1) and
 on(Device2) and
 (not working(Device2))
 then
 proved(failed(Fuse)).
```

```
same_fuse_rule ::
 if
 connected(Device1, Fuse) and
 connected(Device2, Fuse) and
 different(Device1, Device2)
 then
 samefuse(Device1, Device2).
```

```
device_on_rule ::
 if
 working(Device)
 then
 on(Device).
```

```
fact :: different(X, Y) :- not (X = Y).
```

```
fact :: device(heater).
fact :: device(light1).
fact :: device(light2).
fact :: device(light3).
fact :: device(light4).
```

```
fact :: connected(light1, fuse1).
fact :: connected(light2, fuse1).
fact :: connected(heater, fuse1).
fact :: connected(light3, fuse2).
fact :: connected(light4, fuse2).
```

```
askable(on(D), on('Device')).
```

```
askable(working(D), working('Device')).
```

## 16.2. Проектирование механизма логического вывода

### 16.2.1. Требования к организации работы программы

Как было описано в главе 15, правила вывода, подобные приведенным в листингах 16.1 и 16.2, можно перезаписать в виде правил Prolog и в качестве командного интерпретатора экспертной системы непосредственно использовать собственный интерпретатор Prolog. Но в результате этого организация работы программы будет не

совсем удовлетворительной с точки зрения пользователя экспертной системы по двум приведенным ниже причинам.

1. Пользователю не предоставляется удобного способа запрашивать объяснения, например, чтобы узнать, как получен данный ответ.
2. В систему необходимо вводить полную совокупность данных (в виде фактов Prolog), прежде чем появится возможность задавать какие-либо вопросы. Значительная часть этих данных может не потребоваться в процессе формирования командным интерпретатором логического вывода, касающегося рассматриваемой ситуации. Это означает, что пользователю придется выполнять ненужную работу, вводя не относящуюся к делу информацию. Мало того, пользователь вполне может забыть предоставить всю нужную информацию и в этом случае система будет вырабатывать неправильные ответы.

Для устранения этих недостатков необходимо предусмотреть лучший способ взаимодействия пользователя и системы в течение процесса формирования рассуждений и после его завершения. Поэтому при разработке данной системы необходимо обеспечить, чтобы эта система была способной взаимодействовать с пользователем, как показано в следующем примере диалога (ответы пользователя обозначены полужирным шрифтом).

Question, please:

**peter isa tiger.**

Is it true: peter has hair?

**yes.**

Is it true: peter eats meat?

**no.**

Is it true: peter has pointed teeth?

**yes.**

Is it true: peter has claws?

**why.** % Пользователь просит объяснить, для чего нужна эта информация

To investigate, by rule3, peter isa carnivore

To investigate, by rule3, peter isa tiger

This was your question

Is it true: peter has claws?

**yes.**

Is it true: peter has forward pointing eyes?

**yes.**

Is it true: peter has tawny colour?

**yes.**

Is it true: peter has black stripes?

**yes.**

(peter isa tiger) is true

Would you like to see how?

**yes.**

peter isa tiger

was derived by rule5 from

peter isa carnivore

was derived by rule3 from

peter isa mammal

was derived by rule1 from

peter has hair

was told

and

peter has pointed teeth

was told

and

peter has claws

was told

and

peter has forward pointing eyes

was told

and

peter has tawny colour  
was told  
and  
peter has black stripes  
was told

Этот диалог показывает, что система задает пользователю вопросы, касающиеся “простейших” сведений, например, следующим образом:

Is it true: peter eats meat?      % Правда ли, что Питер питается мясом?

Эти вопросы касаются информации, которую нельзя найти в базе знаний или вывести из другой информации. Пользователь может отвечать на такие запросы следующими способами.

1. Передать в качестве ответа на запрос необходимую информацию.
2. Спросить систему, для чего потребовалась эта информация (вводя вопрос “Why?”).

Последний вариант вопроса дает возможность пользователю понять, в чем состоят текущие намерения системы. Пользователь спрашивает, для чего нужна затребованная информация, если запрос системы кажется не относящимся к делу или для ответа на него требуется дополнительная работа со стороны пользователя. Из объяснения системы пользователь может понять, стоит ли та информация, которую запрашивает система, дополнительных усилий для ее получения. Например, предположим, что система спрашивает: “Это животное питается мясом?” В таком случае пользователь, еще не зная ответа и не имея сведений о том, чем питается интересующее его животное, может решить, что нет смысла продолжать наблюдение лишь для того, чтобы узнать, действительно ли это животное питается мясом.

Для получения определенного представления о процессе формирования рассуждений в системе могут использоваться средства трассировки Prolog. Но, как правило, оказывается, что такое средство трассировки является недостаточно гибким для решения задачи, стоящей перед командным интерпретатором. Поэтому было решено создать специальное средство интерпретации на основе системы Prolog вместо применения собственного механизма интерпретации Prolog, который оказался неудовлетворительным с точки зрения данного типа взаимодействия с пользователем. Этот новый интерпретатор будет включать другие средства взаимодействия с пользователем.

## 16.2.2. Организация процесса формирования рассуждений

В системе для поиска ответа на некоторый вопрос (сформулированный в виде высказывания) может применяться несколько способов в соответствии с описанными ниже принципами.

Чтобы найти ответ Answ на вопрос Q, необходимо воспользоваться одним из следующих правил.

- Если ответ на вопрос Q можно найти в виде факта в базе знаний, то ответ Answ состоит в том, что “Q is true” (высказывание Q является истинным).
- Если в базе знаний имеется правило в форме  
‘if Condition then Q’  
то проверить условие Condition и использовать полученный результат для формирования ответа Answ на вопрос Q.
- Если Q — вопрос, обозначенный как “askable” (запрашиваемый), то передать пользователю запрос, касающийся Q.
- Если Q имеет форму Q1 and Q2, то проверить Q1, а затем

- если высказывание Q1 является ложным, то сформировать ответ Answ, что “Q is false”, иначе проверить Q2 и выполнить соответствующую логическую операцию над ответами на вопросы Q1 и Q2 для получения ответа Answ.
- Если Q имеет форму Q1 or Q2, то проверить Q1, а затем
  - если высказывание Q1 является истинным, то вернуть в качестве ответа Answ ответ на вопрос Q1; в противном случае проверить Q2 и, если высказывание Q2 является истинным, вернуть в качестве ответа Answ ответ на вопрос Q2.

Задача поиска ответов на вопросы в форме

not Q

является более сложной и будет рассматриваться позже.

### 16.2.3. Формирование ответов на вопросы, требующие обоснования необходимости получения запрашиваемой информации

Пользователь может задать вопрос, для чего системе нужна информация (в форме “Why?”), если система запрашивает у него некоторые сведения и он хочет получить обоснование такой необходимости. Предположим, что система вывела следующий запрос:

Is a true?

Пользователь может в ответ ввести следующее:

Why?

Соответствующее объяснение может быть сформировано примерно так, как показано ниже.

Поскольку:

Я могу использовать a, чтобы проверить b в соответствии с правилом Ra, и

Я могу использовать b, чтобы проверить c в соответствии с правилом Rb, и

...

Я могу использовать y, чтобы проверить z в соответствии с правилом Ry, и

z был вашим первоначальным вопросом.

Это объяснение представляет собой описание назначения информации, затребованной от пользователя. Объяснение демонстрируется в виде цепочки термов (состоящих из правил и целей), которая соединяет данный фрагмент информации с первоначальным вопросом пользователя. Такую цепочку принято называть *трассировкой*. Любая трассировка может быть наглядно представлена в виде цепочки правил, которая соединяет цель, исследуемую в данный момент, и главную цель в дереве вопросов AND/OR (рис. 16.2). Поэтому формирование ответа на запросы “Why?” осуществляется путем перемещения по дереву поиска вверх от текущей к главной цели. Чтобы иметь возможность решить задачу формирования такого объяснения, необходимо явно сохранять трассировку в течение всего процесса обратного логического вывода.



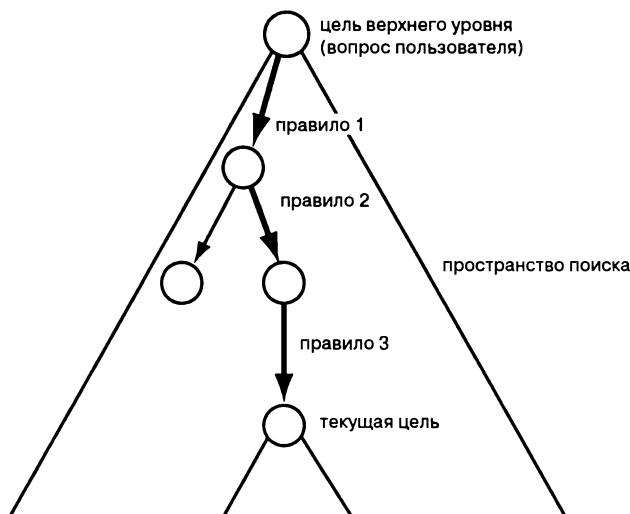


Рис. 16.2. Объяснение необходимости предоставления запрошенной информации. В ответ на вопрос пользователя, для чего системе требуется информация, касающаяся текущей цели, формируется цепочка правил и целей, соединяющая текущую цель и первоначальный вопрос пользователя, который находится в начале цепочки. Таку цепочку принято называть трассировкой

#### 16.2.4. Формирование ответов на вопросы, касающиеся описания последовательности рассуждений

После того как система выдаст ответ на вопрос пользователя, последний может пожелать узнать, как система пришла к такому выводу, введя вопрос в форме "How?". На подобные вопросы, касающиеся описания последовательности рассуждений, следует отвечать, показывая действительные результаты, т.е. промежуточные правила и подцели, с помощью которых было получено данное заключение. При использовании в системе языка правил, подобного описанному в данной главе, соответствующая демонстрация последовательности рассуждений представляет собой дерево доказательства. Программа формирования деревьев доказательства в определенной форме уже рассматривалась в главе 15, а в этой главе такой же принцип применяется в более развитом виде. Одним из удобных способов представления деревьев доказательства в качестве объяснения последовательности рассуждений является использование отступов в тексте, показывающих структуру дерева, например, следующим образом:

```
peter isa carnivore
 was derived by rule3 from
 peter isa mammal
 was derived by rule1 from
 peter has hair
 was told
and
peter eats meat
 was told
```

## 16.3. Реализация

В данном разделе рассматривается реализация командного интерпретатора в соответствии с принципами, описанными в предыдущем разделе. На рис. 16.3 показаны основные объекты, обрабатываемые этим командным интерпретатором. На этом рисунке Goal — рассматриваемый вопрос; Trace — цепочка дочерних целей и правил, соединяющая цель Goal с запросом верхнего уровня; Answer — дерево доказательств для цели Goal.

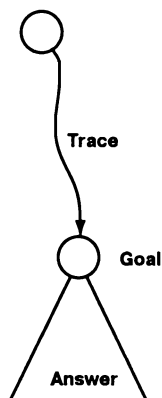


Рис. 16.3. Параметры отношения `explore( Goal, Trace, Answer)`, где Answer — дерево доказательства для цели Goal

Основные процедуры командного интерпретатора приведены ниже.

- `explore( Goal, Trace, Answer)`. Находит ответ Answer на вопрос Goal.
- `useranswer( Goal, Trace, Answer)`. Вырабатывает решения для цели Goal, отмеченной как “askable” (запрашиваемая), задавая пользователю вопросы, касающиеся цели Goal, и отвечая на вопросы “Why?”.
- `present( Answer)`. Показывает полученные результаты и отвечает на вопросы “How?”.
- `expert`. Эта управляющая процедура предусмотрена для правильного ввода в действие перечисленных выше процедур.

Эти четыре основные процедуры описаны в следующих разделах и представлены на языке Prolog в листингах 16.3–16.6. Для получения законченной программы командного интерпретатора необходимо свести воедино все программы, приведенные в указанных листингах.

### 16.3.1. Процедура `explore`

Основой командного интерпретатора является процедура

```
explore(Goal, Trace, Answer)
```

Эта процедура находит в стиле обратного логического вывода ответ Answer на указанный вопрос Goal с использованием принципов, описанных в разделе 16.2.2. Для этого она находит Goal как факт в базе знаний, или применяет к базе знаний некоторое правило, или запрашивает пользователя, или обрабатывает Goal как комбинацию подцелей, соединенную операторами AND или OR.

- **Назначение и структура параметров этой процедуры описаны ниже.**
- **Goal.** Это — рассматриваемый вопрос, представленный в виде комбинации простых высказываний, связанных операторами AND/OR, например, следующим образом:  
(X has feathers) or (X flies) and (X lays eggs)
- **Trace.** Это — цепочка дочерних целей и правил, соединяющая текущую цель Goal с первоначальной целью верхнего уровня и представленная как список элементов в форме:  
Goal by Rule

Это означает, что выполнена проверка цели Goal с помощью правила Rule. Например, допустим, что целью верхнего уровня является “peter isa tiger” (Питер — тигр), а проверяемой в данный момент целью является “peter eats meat” (Питер питается мясом). Соответствующая трассировка, сформированная согласно базе знаний, которая приведена в листинге 16.1, показана ниже.

```
[(peter isa carnivore) by rule3, (peter isa tiger) by rule5]
```

Эта трассировка означает следующее.

- Я могу использовать правило “peter eats meat”, чтобы проверить с помощью правила rule3 истинность высказывания “peter isa carnivore” (Питер — плотоядное животное). Кроме того, я могу использовать высказывание “peter isa carnivore”, чтобы проверить с помощью правила rule5 истинность высказывания “peter isa tiger”.
- **Answer.** Это — дерево доказательств (т.е. дерево решения AND/OR) для вопроса Goal. Общая форма ответа Answer состоит в следующем:  
Conclusion was Found

где Found представляет обоснование для заключения Conclusion. Ниже приведены три примера ответов, иллюстрирующих различные ситуации, возникающие при формировании ответов.

#### 1. Обнаружен факт:

```
(connected(heater, fusel) is true) was 'found as a fact'
```

#### 2. Получен ответ на запрос к пользователю:

```
(peter eats meat) is false was told
```

#### 3. Сформирован логический вывод на основании имеющихся и полученных сведений:

```
(peter isa carnivore) is true was ('derived by' rule3 from (peter isa mammal)
is true was ('derived by' rule1 from (peter has hair) is true was told) and
(peter eats meat) is true was told)
```

Код процедуры explore на языке Prolog приведен в листинге 16.3. В этом коде реализованы принципы, описанные в разделе 16.2.2, с использованием структур данных, указанных выше.

### Листинг 16.3. Основная процедура командного интерпретатора экспертной системы

```
% Процедура
%
% explore(Goal, Trace, Answer)
%
% находит ответ Answer, соответствующий заданной цели Goal, где Trace - цепочка
% родительских целей и правил, с помощью которых процедура explore пытается
% найти содержательный ответ на вопрос. Ответ Answer может принять ложное
% значение, только если были проверены все варианты и для всех них получен
% ложный результат

:- op(900, xfx, ::).
```

```

:- op(800, xfx, was).
:- op(870, fx, if).
:- op(880, xfx, then).
:- op(550, xfy, or).
:- op(540, xfy, and).
:- op(300, fx, 'derived by').
:- op(600, xfx, from).
:- op(600, xfx, by).

```

% В этой программе предполагается, что директивы op( 700, xfx, is)  
% и op( 900, fx, not) уже введены

```

explore(Goal, Trace, Goal is true was 'found as a fact') :-
 fact :: Goal.

```

% Предполагается, что для цели каждого типа предусмотрено только одно правило

```

explore(Goal, Trace,
 Goal is TruthValue was 'derived by' Rule from Answer) :-
 Rule :: if Condition then Goal, % Правило Rule, соответствующее цели Goal
 explore(Condition, [Goal by Rule | Trace], Answer),
 truth(Answer, TruthValue).
explore(Goal1 and Goal2, Trace, Answer) :- !,
 explore(Goal1, Trace, Answer1),
 continue(Answer1, Goal1 and Goal2, Trace, Answer).
explore(Goal1 or Goal2, Trace, Answer) :-
 exploreyes(Goal1, Trace, Answer) % Ответ, соответствующий цели Goal1
 ;
 exploreyes(Goal2, Trace, Answer). % Ответ, соответствующий цели Goal2
explore(Goal1 or Goal2, Trace, Answer1 and Answer2) :- !,
 not exploreyes(Goal1, Trace, _),
 not exploreyes(Goal2, Trace, _), % Соответствующий ответ отсутствует
 explore(Goal1, Trace, Answer1), % Ответ Answer1 должен быть ложным
 explore(Goal2, Trace, Answer2). % Ответ Answer2 должен быть ложным
explore(Goal, Trace, Goal is Answer was told) :-
 useranswer(Goal, Trace, Answer). % Ответ, предоставленный пользователем

exploreyes(Goal, Trace, Answer) :-
 explore(Goal, Trace, Answer),
 positive(Answer).
continue(Answer1, Goal1 and Goal2, Trace, Answer) :-
 positive(Answer1),
 explore(Goal2, Trace, Answer2),
 (positive(Answer2),
 Answer = Answer1 and Answer2
 ;
 negative(Answer2),
 Answer = Answer2
).
continue(Answer1, Goal1 and Goal2, _, Answer1) :-
 negative(Answer1).
truth(Question is TruthValue was Found, TruthValue) :- !.
truth(Answer1 and Answer2, TruthValue) :-
 truth(Answer1, true),
 truth(Answer2, true), !,
 TruthValue = true
 ;
 TruthValue = false.
positive(Answer) :-
 truth(Answer, true).
negative(Answer) :-
 truth(Answer, false).

getreply(Reply) :-
 read(Answer),
 means(Answer, Reply),! % Ответ имеет смысл для программы?

```

```

;
nl, write('Answer unknown, try again please'), nl, % Нет
getreply(Reply). % Сделать еще одну попытку
means(yes, yes).
means(y, yes).
means(no, no).
means(n, no).
means(why, why).
means(w, why).

```

---

## 16.3.2. Процедура useranswer

Процедура `useranswer` представляет собой реализацию средства “запрос к пользователю”. В случае необходимости эта процедура запрашивает у пользователя информацию, а также объясняет ему, для чего она потребовалась. Прежде чем приступить к разработке процедуры `useranswer`, рассмотрим следующую полезную вспомогательную процедуру:

```
getreply(Reply)
```

В ходе диалога от пользователя часто требуется, чтобы он ответил словом “yes”, “no” или “why”. Назначение процедуры `getreply` состоит в получении подобного ответа от пользователя, а также в правильном преобразовании этого ответа, если пользователь вводит сокращение (“y” или “n”) или допускает опечатку. А если ответ пользователя невозможно понять, то `getreply` запрашивает у него другой ответ.

Следует отметить, что процедура `getreply` должна использоваться с осторожностью, поскольку в ней предусмотрено взаимодействие с пользователем с помощью предикатов `read` и `write`, поэтому она может трактоваться только процедурно, а не декларативно. Например, если процедура `getreply` будет вызвана на выполнение следующим образом:

```
getreply(yes)
```

а пользователь введет “no”, то процедура не сможет сопоставить два разных атома и выдаст ответ: “Answer unknown, try again please” (Ответ непонятен, сделайте, пожалуйста, еще одну попытку). Поэтому процедуру `getreply` следует вызывать с неконкретизированным параметром, например, как показано ниже.

```

getreply(Reply),
(Reply = yes, interpretyes(...)
;
Reply = no, interpretno(...)
;
...)

```

### Процедура

```
useranswer(Goal, Trace, Answer)
```

запрашивает у пользователя информацию, касающуюся цели `Goal`, а результатом этого запроса является ответ `Answer`. Трассировка `Trace` используется для выдачи объяснения в том случае, если пользователь спрашивает “why”. Процедура `useranswer` должна вначале проверить, относится ли `Goal` к информации того типа, о которой можно спрашивать пользователя. В данном командном интерпретаторе цели такого рода обозначены как “askable” (запрашиваемые). Для начала предположим, что запрашиваемые сведения определены с помощью следующего отношения:

```
askable(Goal)
```

Это определение будет уточнено позднее. Если цель `Goal` относится к типу “askable”, то эта цель отображается и пользователь должен указать, является ли она истинной или ложной. В том случае, если пользователь спрашивает, для чего потребовалась эта информация, введя вопрос “why”, отображается трассировка `Trace`.

Цель Goal является истинной, если пользователь задает также необходимые значения переменных в цели Goal (при условии, что они имеются). Первая попытка решить эти задачи в программе может состоять в следующем:

```

useranswer(Goal, Trace, Answer) :-
 askable(Goal), % Можно ли обращаться к пользователю с запросом,
 % касающимся цели Goal?
 ask(Goal, Trace, Answer). % Обратиться к пользователю с запросом,
 % касающимся цели Goal

ask(Goal, Trace, Answer) :-
 introduce(Goal), % Вывести запрос к пользователю
 getreply(Reply), % Прочитать ответ пользователя
 process(Reply, Goal, Trace, Answer). % Обработать ответ
process(why, Goal, Trace, Answer) :-
 showtrace(Trace), % Пользователь спросил, для чего
 ask(Goal, Trace, Answer). % нужна эта информация
process(yes, Goal, Trace, Answer) :- % Показать трассировку
 showtrace(Trace), % Снова вывести запрос
 process(yes, Goal, Trace, Answer) :- % Пользователь сообщил, что цель Goal
 % является истинной

 Answer = true,
 askvars(Goal) % Задать вопрос, касающийся переменных
 ;
 ask(Goal, Trace, Answer). % Запросить дополнительные решения
process(no, Goal, Trace, false). % Пользователь сообщил, что цель Goal
 % является ложной

introduce(Goal) :-
 nl, write(Ts it true: '),
 write(Goal), write(?), nl.

```

После вызова процедуры askvars( Goal) пользователь получает запрос ввести значение каждой переменной в цели Goal, следующим образом:

```

askvars(Term) :-
 var(Term), !, % Переменная?
 nl, write(Term), write(' = '),
 read(Term), % Прочитать значение переменной
askvars(Term) :-
 Term =.. [Functor | Args], % Получить параметры структуры
 askarglist(Args). % Задать вопрос, касающийся переменных в параметрах

askarglist([]).
askarglist([Term | Terms]) :-
 askvars(Term),
 askarglist(Terms).

```

Проведем несколько экспериментов с процедурой useranswer. Например, предположим, что в виде “askable” объявлено бинарное отношение eats, как показано ниже.

```
askable(X eats Y).
```

В следующем диалоге между системой Prolog и пользователем текст, введенный пользователем, обозначен полужирным шрифтом.

```

?- useranswer(peter eats meat, [], Answer).
Is it true: peter eats meat? % Запрос к пользователю
yes. % Ответ пользователя
Answer = true

```

Более интересный пример, в котором используются переменные, может выглядеть следующим образом:

```

?- useranswer(Who eats What, [], Answer).
Is it true: _17 eats _18? % Система Prolog присваивает переменным
 % внутренние имена
yes.
_17 = peter. % Запрос, касающийся переменных
_18 = meat.

```

```

Answer = true
Who = peter
What = meat % Перебор с возвратами для получения дополнительных решений
Is it true: _17 eats _18?
yes.
_17 = susan.
_18 = bananas.
Answer = true
Who = susan
What = bananas;
Is it true: _17 eats _18?
no.
Answer = false

```

### 16.3.3. Усовершенствование процедуры useranswer

Один из недостатков первой версии процедуры useranswer, который проявился в приведенном выше диалоге, состоит в том, что сформированные системой Prolog имена переменных в выводе этой системы кажутся невыразительными. При отображении диалога для пользователя символы типа \_17 необходимо заменять некоторыми осмысленными словами.

Еще один, более серьезный недостаток этой версии процедуры useranswer состоит в следующем. Если в дальнейшем процедура useranswer будет применена для достижения аналогичной цели, то пользователю придется повторно вводить все те же сведения, что и раньше. Если этой экспертной системе в процессе формирования в ней рассуждений придется дважды проверить одну и ту же запрашиваемую цель, она будет раздражать пользователя, снова повторяя один и тот же диалог, а не используя информацию, предоставленную пользователем ранее.

Устраним эти два недостатка. Прежде всего, для усовершенствования внешнего вида запросов к пользователю можно предусмотреть некоторый стандартный формат для каждой запрашиваемой цели. Для этого достаточно добавить в отношение askable второй параметр, обозначающий этот формат, как показано в следующем примере:

```
askable(X eats Y, 'z' eats 'Something').
```

Затем при выдаче пользователю запроса каждую переменную в этом запросе необходимо заменить ключевыми словами в указанном формате, например, следующим образом:

```

?- useranswer(X eats Y, [], Answer).
Is it true: Animal eats Something?
yes.
Animal = peter.
Something = meat.
Answer = true
X = peter
Y = meat

```

В усовершенствованной версии процедуры useranswer, приведенной в листинге 16.4, такое форматирование запросов осуществляется с помощью процедуры format( Goal, ExternFormat, Question, Vars0, Variables)

где Goal — формируемая цель, а ExternFormat задает внешний формат для Goal, который определен следующим образом:

```
askable(Goal, ExternFormat)
```

Далее, Question — это цель Goal, отформатированная в соответствии с форматом ExternFormat. Параметр Variables представляет собой список переменных, которые появляются в цели Goal в сопровождении соответствующих им ключевых слов (как указано в формате ExternFormat), введенных в список Vars0, например, как показано ниже.

```
?- format(X gives documents to Y,
'Who' gives 'What' to 'Whom',
Question, [], Variables).
Question = 'Who' gives documents to 'Whom',
Variables = [X/'Who', Y/'Whom'].
```

#### Листинг 16.4. Командный интерпретатор экспертной системы: процедура, которая запрашивает у пользователя необходимую информацию и отвечает на вопросы "why"

```
% Процедура
%
% useranswer(Goal, Trace, Answer)
%
% формирует с помощью перебора с возвратами решения, связанные с достижением
% цели Goal, запрашивая у пользователя информацию.
% Trace - это цепочка родительских целей и правил, применяемая для формирования
% объяснения необходимости затребованной информации

useranswer(Goal, Trace, Answer) :-
 askable(Goal, _, % Цель, информацию о которой можно запросить
 % у пользователя
 freshcopy(Goal, Copy), % Переменные в цели Goal переименованы
 useranswer(Goal, Copy, Trace, Answer, 1).

% Не следует повторно задавать вопрос, касающийся уже конкретизированной цели

useranswer(Goal, _, _, N) :-
 N > 1, % Вопрос должен быть задан повторно?
 instantiate(Goal),!, % В цели Goal переменные отсутствуют
 fail. % Вопрос не следует задавать повторно

% Можно ли предположить, что цель Goal является истинной или ложной
% при всех конкретизациях?

useranswer(Goal, Copy, _, Answer, _) :-
 wastold(Copy, Answer, _),
 instance_of(Copy, Goal),!. % Предполагаемый ответ Answer для цели Goal

% Выполнить выборку известных решений для цели Goal, проиндексированных числами
% от N и выше

useranswer(Goal, _, _, true, N) :-
 wastold(Goal, true, M),
 M>=N.

% Есть ли уже вся необходимая информация о цели Goal?

useranswer(Goal, Copy, _, Answer, _) :-
 end_answers(Copy),
 instance_of(Copy, Goal),!, % Вся необходимая информация
 % о цели Goal уже имеется
 fail.

% Запросить у пользователя (дополнительные) решения

useranswer(Goal, _, Trace, Answer, N) :-
 askuser(Goal, Trace, Answer, N).
askuser(Goal, Trace, Answer, N) :-
 askable(Goal, ExternFormat),
 format(Goal, ExternFormat, Question, [], Variables), % Определить формат
 % вопроса
 ask(Goal, Question, Variables, Trace, Answer, N).
ask(Goal, Question, Variables, Trace, Answer, N) :-
 nl,
 (Variables = [],!,
 write('Is it true:')) % Вывести запрос с приглашением
```



```

;
write('Any (more) solution to:') % Вывести запрос с приглашением
),
write(Question), write('? '),
getreply(Reply), !, % Ответ равен yes/no/why
process(Reply, Goal, Question, Variables, Trace, Answer, N).

process(why, Goal, Question, Variables, Trace, Answer, N) :-
 showtrace(Trace),
 ask(Goal, Question, Variables, Trace, Answer, N).

process(yes, Goal, _, Variables, Trace, true, N) :-
 nextindex(Next), % Получить новый незанятый индекс для факта wastold
 Next1 is Next + 1,
 (askvars(Variables),
 assertz(wastold(Goal, true, Next)) % Внести решение в базу данных
 ;
 freshcopy(Goal, Copy), % Копия цели Goal
 useranswer(Goal, Copy, Trace, Answer, Next1) % Есть еще ответы?
).

process(no, Goal, _, _, false, N) :-
 freshcopy(Goal, Copy),
 wastold(Copy, true, _), !, % 'no' означает, что решений больше нет
 assertz(end_answers(Goal)), % Отметить конец списка решений
 fail
 ;
 nextindex(Next), % Следующий незанятый индекс для факта wastold
 assertz(wastold(Goal, false, Next)). % 'no' означает, что решений больше нет

format(Var, Name, Name, Vars, [Var/Name | Vars]) :-
 var(Var), !.

format(Atom, Name, Atom, Vars, Vars) :-
 atomic(Atom), !,
 atomic(Name).

format(Goal, Form, Question, Vars0, Vars) :-
 Goal =.. [Functor | Args1],
 Form =.. [Functor | Forms],
 formatall(Args1, Forms, Args2, Vars0, Vars),
 Question =.. [Functor | Args2].

formatall([], [], [], Vars, Vars).

formatall([X | XL], [F | FL], [Q | QL], Vars0, Vars) :-
 formatall(XL, FL, QL, Vars0, Vars1),
 format(X, F, Q, Vars1, Vars).

askvars([]).

askvars([Variable/Name | Variables]) :-
 nl, write(Name), write(' = '),
 read(Variable),
 askvars(Variables).

showtrace([]) :-
 nl, write('This was your question'), nl.
showtrace([Goal by Rule | Trace]) :-
 nl, write('To investigate, by '),
 write(Rule), write(', '),
 write(Goal),
 showtrace(Trace).

instantiate(Term) :-
 numbervars(Term, 0, 0). % В терме Term переменные отсутствуют

```

```

% instance_of(T1, T2): T2 - экземпляр T1; это означает, что терм T1 является
% более общим или столь же общим, как и T2

instance_of(Term, Term1) :- % Экземпляром терма Term является Term1
 freshcopy(Term1, Term2), % Копия терма Term1 с обновленным множеством
 % переменных

 numbervars(Term2, 0, _), !,
 Term = Term2. % Эта цель достигается успешно, если
 % Term1 - экземпляр терма Term

freshcopy(Term, FreshTerm) :- % Создать копию терма Term с переименованными
 % переменными

 asserta(copy(Term)),
 retract(copy(FreshTerm)), !.

nextindex(Next) :- % Следующий незанятый индекс для факта wastold
 retract(lastindex(Last)), !,
 Next is Last + 1,
 assert(lastindex(Next)).

% Инициализировать динамические процедуры lastindex/1, wastold/3, end_answers/1

:- assertz(lastindex(0)),
 assertz(wastold(dummy, false, 0)),
 assertz(end_answers(dummy)).

```

Еще одно усовершенствование, которое позволило бы избежать повторных вопросов к пользователю, реализовать немного труднее. Прежде всего, для этого необходимо запоминать все ответы пользователя, чтобы можно было осуществить их выборку в некоторый последующий момент. Это можно обеспечить, внося в базу данных ответы пользователя как элементы некоторого отношения, например, следующим образом:

```
assert(wastold(mary gives documents to friends, true)).
```

Но в той ситуации, когда пользователь предоставил несколько решений, относящихся к одной и той же цели, в базу данных будет внесено несколько фактов, касающихся этой цели. В этом и состоит сложность. Предположим, что в разных местах в базе данных присутствует несколько вариантов одной и той же цели (причем цель одна, а переменные в ней разные), например, как показано ниже.

```

(X has Y) and % Первый экземпляр факта - Goal1
...
(X1 has Y1) and % Второй экземпляр факта - Goal2
...

```

Кроме того, предположим, что пользователь запросил (с помощью перебора с возвратами) несколько решений для цели Goal1. После этого процесс формирования рассуждений перешел к цели Goal2. А поскольку уже имеются некоторые решения для Goal1, то хотелось бы, чтобы система применяла их автоматически также и к цели Goal2 (поскольку они, безусловно, позволяют достичь и Goal2). Теперь предположим, что система пытается применить эти решения для Goal2, но ни одно из них не позволяет больше достичь какой-либо иной цели. Поэтому система должна возвратиться к Goal2 и потребовать от пользователя дополнительные решения. Если пользователь предоставит дополнительные решения, их также нужно запомнить. В том случае, если система в дальнейшем вернется к цели Goal1, то эти новые решения необходимо также автоматически применить к цели Goal1.

Для того чтобы можно было правильно использовать информацию, предоставленную пользователем на разных этапах работы с программой, необходимо проиндексировать эту информацию. Поэтому внесенные в базу данных факты будут иметь следующую форму:

```
wastold(Goal, TruthValue, Index)
```

где Index — счетчик ответов, предоставленных пользователем. Процедура

```
useranswer(Goal, Trace, Answer)
```

должна следить за количеством решений, которые уже были выработаны с помощью перебора с возвратами. Такую задачу можно решить с использованием еще одной процедуры, `useranswer`, с четырьмя параметрами:

```
useranswer(Goal, Trace, Answer, N)
```

где `N` — целое число. В вызове этой процедуры необходимо вырабатывать решения для цели `Goal`, проиндексированные целым числом `N` или большим числом. А вызов `useranswer( Goal, Trace, Answer)`

предназначен для выработки всех решений для цели `Goal`. Решения индексируются целыми числами, начиная с 1, поэтому имеет место следующее отношение:

```
useranswer(Goal, Trace, Answer) :-
 useranswer(Goal, Trace, Answer, 1).
```

Общая схема функционирования процедуры

```
useranswer(Goal, Trace, Answer, N)
```

состоит в том, что необходимо формировать решения для цели `Goal`, вначале осуществляя выборку известных решений, проиндексированных числами от `N` и больше. После того как известные решения будут исчерпаны, нужно начать выдавать пользователю запросы, касающиеся цели `Goal`, и вносить в базу данных полученные таким образом новые решения, проиндексированные должным образом последовательно возрастающими числами. После того как пользователь сообщит, что решений больше нет, внести в базу данных следующий факт:

```
end_answers(Goal)
```

А если пользователь сразу же сообщит, что решений вообще не существует, то непосредственно внести в базу данных такой факт:

```
wastold(Goal, false, Index)
```

При выборке решений процедура `useranswer` должна правильно интерпретировать такую информацию.

Но есть еще одна сложность. Пользователь имеет также право задавать общие решения, оставляя некоторые переменные неконкретизированными. А если есть возможность осуществить выборку положительного решения, более общего или столь же общего, как `Goal`, то, безусловно, нет смысла продолжать задавать вопросы, касающиеся `Goal`, если уже известно самое общее решение. Если же является истинным следующее высказывание:

```
wastold(Goal, false, _)
```

то должно быть принято аналогичное решение.

Все эти требования учтены в программе `useranswer`, приведенной в листинге 16.4. Введен еще один параметр, `Copy` (копия цели `Goal`), который используется в некоторых согласованиях вместо `Goal`, чтобы нельзя было уничтожить переменные в цели `Goal`. В этой программе применяются также два вспомогательных отношения. Одним из них является следующее:

```
instantiated(Term)
```

Это отношение принимает истинное значение, если терм `Term` не содержит переменных. Другим из них является

```
instance_of(Term, Term1)
```

где `Term1` — экземпляр терма `Term`; это означает, что `Term` является, по меньшей мере, таким же общим, как `Term1`, например, как показано ниже.

```
instance_of(X gives information to Y, mary gives information to Z)
```

В этих двух процедурах используется еще одна процедура:

```
numbervars(Term, N, M)
```

Эта процедура “нумерует” переменные в терме `Term`, заменяя каждую переменную в этом терме некоторым вновь сформированным термом таким образом, чтобы

эти “пронумерованные” термы соответствовали целым числам между от N до M-1. Например, предположим, что эти термы представлены в следующей форме:

```
var/0, var/1, var/2, ...
```

В таком случае вопрос

```
?- Term = f(X, t(a, Y, X)), numbervars(Term, 5, M).
```

приведет к получению такого результата:

```
Term = f(var/5, t(a, var/6, var/5))
X = var/5
Y = var/6
M = 7
```

Подобная процедура `numbervars` часто предоставляется в системе Prolog в виде встроенного предиката. В противном случае ее можно ввести в программу, как показано ниже.

```
numbervars(Term, N, Nplus1) :-
 var(Term),!,
 Term = var/N,
 Nplus1 is N + 1.
numbervars(Term, N, M) :-
 Term =.. [Functor | Args], % Структура или атом
 numberargs(Args, N, M). % Количество переменных в параметрах

numberargs([], N, N) :- !.
numberargs([X | L], N, M) :-
 numbervars(X, N, N1),
 numberargs(L, N1, M).
```

## 16.3.4. Процедура `present`

Процедура

```
present(Answer)
```

приведенная в листинге 16.5, отображает окончательный результат сеанса экспертной консультации и вырабатывает объяснение последовательности рассуждений. Параметр `Answer` включает и ответ на вопрос пользователя, и дерево доказательства, показывающее, как было получено это заключение. Процедура `present` вначале выдает заключение. А если пользователь затем пожелает увидеть, как было сформировано это заключение, введя вопрос “how”, то в некоторой приемлемой форме отображается дерево доказательства, которое представляет собой объяснение последовательности рассуждений. Форма такого объяснения показана на примерах в разделах 16.2.1 и 16.2.4.

**Листинг 16.5. Командный интерпретатор экспертной системы: процедура, которая отображает окончательный результат и выводит объяснение последовательности рассуждений**

```
% Процедура, которая отображает результат экспертной консультации и выводит
% объяснение последовательности рассуждений
```

```
present(Answer) :-
 nl, showconclusion(Answer),
 nl, write('Would you like to see how? '),
 getreply(Reply),
 (Reply = yes, !,
 show(Answer) % Показать дерево решения
 ;
 true
).
showconclusion(Answer1 and Answer2) :- !,
 showconclusion(Answer1), write('and '),
 showconclusion(Answer2).
```

```

showconclusion(Conclusion was Found) :-
 write(Conclusion).

% Процедура show отображает полное дерево решения

show(Solution) :-
 nl, show(Solution, 0), !. % Установить отступ, равный 0

show(Answer1 and Answer2, H) :- !, % Установить отступ, равный H
 show(Answer1, H),
 tab(H), write(and), nl,
 show(Answer2, H).

show(Answer was Found, H) :- % Отступ на H
 tab(H), writeans(Answer), % Показать экспертное заключение
 nl, tab(H),
 write('was '),
 showl(Found, H). % Показать подтверждение

showl(Derived from Answer, H) :- !,
 write(Derived), write('from'), % Вывести имя правила
 nl, H1 is H + 4,
 show(Answer, H1). % Показать исходное высказывание

showl(Found, _) :- % Переменная Found может принимать значения 'told'
 % (указано пользователем) или 'found as fact' (найдено как факт)
 write(Found), nl.

writeans(Goal is true) :- !,
 write(Goal). % Исключить в выводе слова 'is true'
 % (является истинным)

writeans(Answer) :- % Это - отрицательный ответ
 write(Answer).

```

---

### 16.3.5. Управляющая процедура верхнего уровня

Для обеспечения удобного доступа к командному интерпретатору экспертной системы из среды интерпретатора Prolog требуется управляющая процедура, образцом которой может служить процедура `expert` (листинг 16.6). После вызова на выполнение процедура `expert` координирует работу трех основных модулей командного интерпретатора (см. листинги 16.3–16.5), например, следующим образом:

```

?- expert.
Question, please: % Приглашение пользователю к вводу вопроса
X isa animal and goliath isa X. % Вопрос пользователя
Is it true: goliath has hair?
...

```

**Листинг 16.6. Управляющая процедура командного интерпретатора экспертной системы.**  
**Командный интерпретатор можно вызвать на выполнение из системы Prolog с помощью процедуры `expert`**

---

```

% Управляющая процедура верхнего уровня

expert :-
 getquestion(Question), % Вывести приглашение пользователю к вводу вопроса
 (answeryes(Question) % Попытаться найти положительный ответ
 ;
 answerno(Question) % Если положительного ответа нет, найти отрицательный
).
answeryes(Question) :- % Поиск положительных ответов на вопрос Question
 markstatus(negative), % Положительных ответов еще нет
 explore(Question, [], Answer), % Трассировка пуста

```

```

positive(Answer), % Выполнить поиск положительных ответов
markstatus(positive), % Положительный ответ найден
present(Answer), nl,
write('More solutions? '),
getreply(Reply), % Получить от пользователя ответ на запрос
Reply = no. % В противном случае возвратиться к процедуре explore
answerno(Question) :- % Выполнить поиск отрицательного ответа на вопрос
retract(no_positive_answer_yet), !, % Положительного ответа не было?
explore(Question, [], Answer),
negative(Answer),
present(Answer), nl,
write('More negative solutions? '),
getreply(Reply),
Reply = no. % В противном случае возвратиться к процедуре
markstatus(negative) :-
assert(no_positive_answer_yet).
markstatus(positive) :-
retract(no_positive_answer_yet), !
;
true.
getquestion(Question) :-
nl, write('Question, please'), nl,
read(Question).

```

---

### 16.3.6. Пояснения к программе командного интерпретатора

Создается впечатление, что в некоторых фрагментах рассматриваемой программы командного интерпретатора отсутствует декларативная ясность, характерная для программ Prolog. Причина этого состоит в том, что в подобном командном интерпретаторе приходится обеспечивать более жесткое управление процессом выполнения, поскольку предполагается, что экспертная система не только находит ответ, но и обеспечивает его поиск таким способом, который представляется разумным для пользователя, постоянно взаимодействующего с системой. Поэтому в программе пришлось реализовать определенный процесс решения задачи, а не просто логический вывод на основе отношений, связывающих между собой входную и выходную информацию. В связи с этим результирующая программа действительно характеризуется наличием более ярко выраженных процедурных свойств, чем обычно. Она относится к одному из таких примеров, в которых нельзя полагаться на собственный процедурный механизм Prolog, поэтому требуется подробно регламентировать процедурное поведение программы.

### 16.3.7. Отрицаемые цели

На первый взгляд кажется, что вполне можно разрешить использовать отрицание в левых частях правил, а следовательно, и в вопросах, которые проверяются процедурой explore. Прямолинейная попытка применения вопросов с отрицаниями может состоять в следующем:

```

explore(not Goal, Trace, Answer) :- !,
explore(Goal, Trace, Answer1),
invert(Answer1, Answer). % Операция отрицания истинностного значения
invert(Goal is true was Found, (not Goal) is false was Found).
invert(Goal is false was Found, (not Goal) is true was Found).

```

Такая попытка завершается успехом, только если цель Goal конкретизирована, а в ином случае возникают проблемы. Рассмотрим, в частности, следующий пример:

```

?- expert.
Question, please:
not (X eats meat).
Any (more) solution to: Animal eats meat?

```

```
yes.
Animal = tiger.
```

После этого система выдает следующий ответ:

```
not (tiger eats meat) is false % Высказывание not (tiger eats meat) ложно
```

Этот ответ не является удовлетворительным. Проблема заключается в том, что под рассматриваемым вопросом пользователи часто подразумевают совсем иное:

```
not (X eats meat) % Найти такой объект X, который не питается мясом
```

Пользователь фактически хочет спросить: “Есть ли сведения о таком объекте X, что X не питается мясом?” Но способ, с помощью которого этот вопрос интерпретируется процедурой `explore` (в соответствии с ее определением), состоит в следующем.

1. Есть ли сведения о некотором объекте X, таком, что X питается мясом?
2. Да, мясом питается тигр.
3. Поэтому высказывание `not ( tiger eats meat)` является ложным.

Короче говоря, в системе применяется следующая интерпретация этого вопроса: “Является ли истинным, что ни один объект X не ест мяса?” Поэтому система даст положительный ответ на этот вопрос лишь в том случае, если никто из известных ей объектов не питается мясом. Эту мысль можно выразить иначе: процедура `explore` отвечает на вопрос с отрицанием таким образом, как если бы к объекту X был применен квантор всеобщности:

для всех X: является ли истинным высказывание `not (X eats meat)`?

а не так, как если бы к объекту X был применен квантор существования, в чем состояло намерение пользователя, задающего вопрос с отрицанием:

для некоторого X: есть ли такой X, что истинно высказывание `not (X eats meat)`?

Если рассматриваемый вопрос является конкретизированным, то проблемы при его обработке не возникают. В противном случае правильная трактовка вопросов с отрицанием становится более сложной. Некоторые возможные варианты анализа этой проблемы описаны ниже.

Чтобы проверить истинность цели `not Goal`, выполнить проверку `Goal`, а затем:

- если `Goal` является ложной, то `not Goal` истинна;
- если `Goal'` — решение `Goal`
  - и `Goal'` является столь же общей, что и `Goal`,
  - то `not Goal` ложна;
- если `Goal'` — решение `Goal` и
  - `Goal'` является более конкретной, чем `Goal`,
  - то мы не можем сказать ничего определенного о `not Goal`.

Этих сложностей можно избежать, разрешив использовать только конкретизированные отрицаемые цели. Такое требование часто можно осуществить с применением более правильных формулировок правил в базе знаний. В листинге 16.2 подобная задача решена с помощью следующей модификации правила определения неисправного прибора `broken_rule`:

```
broken_rule :: if
 on(Device) and
 device(Device) and % Конкретизировать переменную Device
 not working(Device) and
 connected(Device, Fuse) and
 proved(intact(Fuse))
then
 proved(broken(Device)).
```

Условие

```
device(Device)
```

“защищает” стоящее за ним условие

```
not working(Device)
```

от использования в неконкретизированном виде.

## Упражнение

### 16.1. База знаний может содержать циклы, как показано ниже.

```
rule1:: if bottle_empty then john_drunk. % Если бутылка пуста, то Джон пьян
rule2:: if john_drunk then bottle_empty. % Если Джон пьян, то бутылка пуста
```

При использовании такой базы знаний процедура `explore` может начать переходить по циклу между одними и теми же целями. Модифицируйте процедуру `explore`, чтобы предотвратить подобное заикливание. Для этого можно применить трассировку `Trace`. Но необходимо соблюдать осторожность: если текущая цель согласуется с предыдущей целью, такую ситуацию нельзя рассматривать как цикл, при условии, что текущая цель является более общей, чем предыдущая.

## 16.4. Заключительные замечания

Рассматриваемый в этой главе командный интерпретатор экспертной системы может быть во многом усовершенствован. В связи с этим следует сделать несколько критических замечаний и предложений по его доработке.

Рассматриваемая программа представляет собой прямолинейную реализацию основных идей, и в ней не уделено достаточно внимания вопросам эффективности. Для более эффективной реализации потребовались бы более сложные структуры данных, индексированная или иерархическая организация правил и т.д.

Применяемая процедура `explore` восприимчива к образованию циклов, возникающих, если в правилах базы знаний одна и та же цель упоминается “циклически” (и как условие, и как заключение). Этот недостаток можно легко исправить, введя проверку на циклы в процедуре `explore`; для этого достаточно проверить, не является ли текущая цель экземпляром другой цели, которая уже входит в состав трассировки `Trace`.

В качестве объяснения последовательности рассуждений отображается все дерево доказательства. В случае большого дерева доказательства было бы лучше отобразить только верхнюю часть дерева, а затем дать возможность пользователю “переходить” по остальной части дерева в соответствии с его пожеланиями. Тогда пользователь сможет выборочно изучать дерево доказательства с помощью примерно таких команд: “Перейти вниз на один уровень”, “Перейти вниз на два уровня”, ..., “Подняться вверх”, “Достаточно”.

В этом командном интерпретаторе при формировании объяснений последовательности рассуждений (в ответ на вопрос “как”) и назначения затребованной информации (в ответ на вопрос “для чего”) правила упоминаются только под их именами, а сами правила не отображаются явно. Во время сеанса получения экспертной консультации пользователю должна быть предоставлена возможность применять режим явного вывода правил.

Практика показала, что задача формулировки таких запросов к пользователю, чтобы диалог выглядел естественным, является довольно сложной. Применяемые в данной главе решения по определенной степени оправданы, но в некоторых обстоятельствах могут обнаруживаться другие проблемы, например, как показано ниже.

```
Is it true: susan flies? % Сьюзен может летать?
no.
Is it true: susan isa good flyer? % Сьюзен хорошо летает?
```



Ответ на последний вопрос заведомо должен быть отрицательным, поскольку Сьюзен вообще не может летать! Еще один подобный пример приведен ниже.

Any (more) solution to: Somebody flies?

yes.

Somebody = bird.

Is it true: albatross flies? % Это можно выяснить с помощью логического вывода

Чтобы устранить подобные недостатки, необходимо ввести дополнительные отношения между понятиями, с которыми оперирует экспертная система. Как правило, эти новые отношения задают иерархические связи между объектами и показывают, как происходит наследование свойств. Это можно осуществить с использованием представлений в виде семантических сетей или фреймов (см. главу 15).

Еще одно усовершенствование процедуры получения ответов на запросы к пользователю может предусматривать применение оптимальной стратегии выдачи запросов. Задача оптимизации может состоять в максимальном уменьшении количества запросов, на которые должен ответить пользователь, прежде чем будет сформулировано заключение. Безусловно, возможно наличие альтернативных стратегий, и то, какая из них в конечном итоге окажется оптимальной, зависит от ответов пользователя. Решение о том, какой альтернативной стратегии следует придерживаться, может быть основано на некоторых априорных вероятностях, позволяющих дать вероятностную оценку “стоимости” каждой альтернативы. Кроме того, может потребоваться обновлять эти оценки после каждого ответа пользователя.

Иногда рассматривается еще один критерий оптимизации: длина вывода некоторого заключения. Дело в том, что чем короче вывод, тем проще объяснение последовательности рассуждений. Кроме того, сложность объяснений можно уменьшить, выбирая подходя к обработке отдельных тривиальных правил, не представляющих интереса для пользователя. Такие правила не нужно помещать в трассировки Trace и ответы Answer, формируемые процедурой explore. В этом случае может потребоваться, чтобы в базе знаний было указано, какие правила “подлежат трассировке” и поэтому должны появляться в объяснениях, а какие — нет.

Управление интеллектуальной экспертной системой должно быть организовано с учетом законов теории вероятностей, чтобы эта система могла выбирать наиболее вероятные в данный момент гипотезы среди многих конкурентов. Экспертная система должна запрашивать у пользователя такую информацию, которая позволяет легче проводить различия среди наиболее вероятных гипотез.

Экспертные системы, рассматриваемые в данной главе, относятся к классификационному или “аналитическому” типу. Антиподом этих систем являются системы “синтетического” типа, перед которыми стоит задача создать что-то новое. В последнем случае результатом становится формирование плана действий по выполнению некоторого задания, например, составление плана действий робота, разработка конфигурации компьютера, которая полностью соответствует заданной спецификации, или поиск форсированной комбинации в шахматах. Приведенный в этой главе пример системной диагностики неисправностей может быть дополнен естественным образом, чтобы он охватывал и некоторые действия. Например, если диагностику неисправностей нельзя провести из-за того, что некоторые приборы выключены, то система может предложить “Включите светильник 3 и сообщите о результатах”. Эта задача может быть связана с проблемой оптимального планирования, т.е. максимально возможного уменьшения количества действий, необходимых для формирования некоторого заключения.

## Проект

Изучите критические замечания и возможные дополнения к рассматриваемому командному интерпретатору экспертной системы, приведенные выше, затем спроектируйте и реализуйте соответствующие усовершенствования.

## Резюме

*Командный интерпретатор*, разработанный и запрограммированный в этой главе, обладает следующими характеристиками.

- Интерпретирует правила вывода.
- Предоставляет объяснения последовательности рассуждений (в ответ на вопрос “как”) и назначения затребованной информации (в ответ на вопрос “для чего”).
- Выводит запросы к пользователю для получения необходимой информации.

## Дополнительные источники информации

Проект приведенного в этой главе командного интерпретатора экспертной системы в определенной степени является аналогичным описанному в [62]. Некоторые примеры, используемые в данной главе, адаптированы из [170].

# Планирование

*В этой главе...*

|                                                                                                                |     |
|----------------------------------------------------------------------------------------------------------------|-----|
| 17.1. Представление действий                                                                                   | 383 |
| 17.2. Разработка планов с помощью анализа целей и средств                                                      | 387 |
| 17.3. Защита целей                                                                                             | 390 |
| 17.4. Процедурные аспекты и режим поиска в ширину                                                              | 393 |
| 17.5. Регрессия целей                                                                                          | 395 |
| 17.6. Сочетание планирования по принципу анализа целей и средств с эвристическим поиском по заданному критерию | 398 |
| 17.7. Неконкретизированные действия и планирование с частичным упорядочением                                   | 401 |

Планирование — это одна из тем в проблематике искусственного интеллекта, которая всегда привлекает значительный интерес. Она касается формирования рассуждений о результатах действий и упорядочении возможных действий для достижения заданного суммарного эффекта. В этой главе описано несколько простых планировщиков, которые иллюстрируют принципы планирования.

## 17.1. Представление действий

Пример задачи планирования приведен на рис. 17.1. Ее можно решить по методу поиска в пространстве состояний (см. главу 11). Но в данной главе эта задача будет представлена таким образом, что появится возможность явно рассуждать о результатах возможных действий, среди которых планировщик может выбирать наиболее приемлемые.

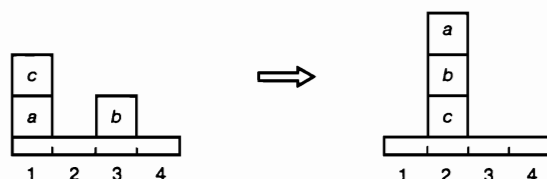


*Рис. 17.1. Проблема планирования в мире блоков — найти последовательность действий, которые позволяют достичь следующих целей: блок *a* должен стоять на блоке *b*, а блок *b* — на блоке *c*. Такие действия должны преобразовать начальное состояние (показанное слева) в конечное состояние (справа)*

Действия изменяют текущее состояние планируемого мира, вызывая тем самым переход в новое состояние. Но ни одно действие обычно не изменяет все в текущем состоянии; затрагивается только отдельный компонент (компоненты) этого состояния. Поэтому в качественном представлении должна учитываться такая “локализация” результатов действий. Для упрощения рассуждений о подобных локальных результатах действий состояние удобнее всего представлять в виде списка связей, которые в данный момент являются истинными. Безусловно, при этом следует упоминать лишь такие связи, которые касаются данной задачи планирования. Если речь идет о планировании в мире блоков, то подобными связями являются следующие:

```
on(Block, Object)
```

и

```
clear(Object)
```

В последнем выражении утверждается, что верхняя грань объекта `Object` открыта (на ней нет других блоков). При планировании в мире блоков такая связь является важной, поскольку блок, подлежащий перемещению, должен быть открытым сверху, а другой блок (или место), на который перемещается этот блок, также должен иметь открытую верхнюю поверхность. Открытый сверху объект `Object` может представлять собой блок или место на плоскости. В данном примере `a`, `b` и `c` — блоки, `a 1`, `2`, `3` и `4` — места. В таком случае начальное состояние мира (см. рис. 17.1) можно представить с помощью следующего списка связей:

```
[clear(2), clear(4), clear(b), clear(c), on(a, 1), on(b, 3), on(c, a)]
```

Обратите внимание на то, что мы проводим различие между отношением и связью. Например, `on( c, a)` — это связь, т.е. конкретный экземпляр отношения `on`. Отношение `on`, с другой стороны, является множеством всех связей `on`.

Каждое возможное действие определяется в терминах его предпосылок и результатов. Точнее, каждое действие задается с помощью трех информационных структур.

1. Предпосылки — условия, которые должны соблюдаться в некоторой ситуации для того, чтобы данное действие стало возможным.
2. Список добавления — список связей, установленных данным действием в текущей ситуации; это — условия, которые становятся истинными после выполнения этого действия.
3. Список удаления — список связей, разрушаемых в результате данного действия.

Предпосылки будут определяться с помощью следующей процедуры:

```
can(Action, Cond)
```

Эта процедура содержит информацию о том, что действие `Action` может быть выполнено только в той ситуации, в которой соблюдается условие `Cond`.

Результаты действия определяются следующими процедурами:

```
adds(Action, AddRels)
```

```
deletes(Action, DelRels)
```

где `AddRels` — список связей, установленных в результате выполнения действия `Action`. После выполнения действия `Action` в некотором состоянии к этому состоянию добавляются связи `AddRels` для получения нового состояния. С другой стороны, `DelRels` — это список связей, уничтожаемых действием `Action`. Эти связи удаляются из состояния, к которому применяется действие `Action`.

В рассматриваемом мире блоков возможно действие только следующего типа:

```
move(Block, From, To)
```

где `Block` — это перемещаемый блок, `From` — текущая позиция блока, а `To` — его новая позиция. Полное определение этого действия приведено в листинге 17.1.

## Листинг 17.1. Определение пространства планирования для мира блоков

```
% Определение действия move(Block, From, To) в мире блоков

% can(Action, Condition):
% действие Action возможно, если условие Condition является истинным

can(move(Block, From, To), [clear(Block), clear(To), on(Block, From)]) :-
 is_block(Block), % Перемещаемый блок
 object(To), % To - это блок или место
 To \== Block, % Блок не может быть поставлен сам на себя
 object(From), % From - это блок или место
 From \== To, % Перемещение происходит в новую позицию
 Block \== From. % Блок не может быть снят сам с себя

% adds(Action, Relationships):
% в результате действия Action устанавливаются связи Relationships

adds(move(X,From,To), [on(X,To), clear(From)]).

% deletes(Action, Relationships):
% в результате действия Action уничтожаются связи Relationships

deletes(move(X,From,To), [on(X,From), clear(To)]).

object(X) :- % X является одним из объектов, если
 place(X) % X - место
 ; % или
 is_block(X). % X - блок

% Мир блоков

is_block(a).
is_block(b).
is_block(c).

place(1).
place(2).
place(3).
place(4).

% Состояние в мире блоков
%
% c
% a b
% ====
% места 1234

state1([clear(2), clear(4), clear(b), clear(c), on(a,1), on(b,3),
 on(c,a)]).
```

Определение возможных действий для задачи планирования неявно содержит определение пространства всех возможных планов, поэтому такое определение называют также *пространством планирования*.

Планирование в мире блоков представляет собой традиционную область экспериментов по планированию, которая обычно связана с задачей составления программ для роботов при использовании роботов для построения конструкций из блоков. Но можно легко найти много примеров задач планирования и в нашей повседневной жизни. В листинге 17.2 приведено определение пространства планирования для манипулирования фотокамерой. В этой задаче планирования речь идет о подготовке фотокамеры к работе, т.е. вставке новой пленки и замене аккумулятора в случае необходимости. План вставки новой пленки в данном пространстве планирования состоит в следующем: открыть футляр и вынуть фотокамеру, перемотать старую плен-

ку, открыть отсек для пленки, вынуть старую пленку, вставить новую пленку, закрыть отсек для пленки. Состояние, в котором фотокамера готова для получения снимков, определяется следующим образом:

```
[slot_closed(battery), slot_closed(film), in(battery),
ok(battery), in(film), film_at_start, film_unused]
```

### Листинг 17.2. Определение пространства планирования для манипулирования фотокамерой

```
% Пространство планирования для подготовки фотокамеры к работе
% Открыть футляр
can(open_case, [camera_in_case]).
adds(open_case, [camera_outside_case]).
deletes(open_case, [camera_in_case]).

% Закрыть футляр
can(close_case, [camera_outside_case, slot_closed(film),
slot_closed(battery)]).
adds(close_case, [camera_in_case]).
deletes(close_case, [camera_outside_case]).

% Открыть отсек для пленки
can(open_slot(X), [camera_outside_case, slot_closed(X)]).
adds(open_slot(X), [slot_open(X)]).
deletes(open_slot(X), [slot_closed(X)]).

% Закрыть отсек для пленки
can(close_slot(X), [camera_outside_case, slot_open(X)]).
adds(close_slot(X), [slot_closed(X)]).
deletes(close_slot(X), [slot_open(X)]).

% Перемотать пленку
can(rewind, [camera_outside_case, in(film), film_at_end]).
adds(rewind, [film_at_start]).
deletes(rewind, [film_at_end]).

% Вынуть аккумулятор или пленку
can(remove(battery), [slot_open(battery), in(battery)]).
can(remove(film), [slot_open(film), in(film), film_at_start]).
adds(remove(X), [slot_empty(X)]).
deletes(remove(X), [in(X)]).

% Вставить новый аккумулятор или пленку
can(insert_new(X), [slot_open(X), slot_empty(X)]).
adds(insert_new(battery), [in(battery), ok(battery)]).
adds(insert_new(film), [in(film), film_at_start, film_unused]).
deletes(insert_new(X), [slot_empty(X)]).

% Снять фотографии
can(take_pictures, [in(film), film_at_start, film_unused,
in(battery), ok(battery), slot_closed(film), slot_closed(battery)]).
adds(take_pictures, [film_at_end]).
deletes(take_pictures, [film_at_start, film_unused]).

% Состояние, в котором пленка израсходована, а аккумулятор разряжен
% (Примечание. Аккумулятор считается разряженным, потому что связь
% ok(battery) не включена в это состояние.)
state1([camera_in_case, slot_closed(film), slot_closed(battery),
in(film), film_at_end, in(battery)]).
```

Цель плана определяется в терминах связей, которые должны быть установлены после его выполнения. Для задачи мира блоков (см. рис. 17.1) задачу можно поставить как следующий список связей:

```
[on(a, b), on(b, c)]
```

Для задачи с фотокамерой список связей, который гарантирует, что фотокамера готова к съемке, состоит в следующем:

```
[in(film), film_at_start, film_unused, in(battery),
ok(battery), slot_closed(film), slot_closed(battery)]
```

В приведенном ниже разделе будет показано, как можно воспользоваться таким представлением в процессе, известном как “анализ целей и средств”, с помощью которого может быть разработан план.

## Упражнения

- 17.1. Расширьте определение мира прямоугольных блоков, приведенное в листинге 17.1, для включения в него объектов других типов, таких как пирамиды, шары и ящики. Введите соответствующее дополнительное условие “допускает установку в столбик” в отношении `can`. Например, установка на пирамиде прямоугольного блока для построения столбика не допускается; шар можно класть в ящик, но не ставить на блок (поскольку он скатится с верхней поверхности блока).
- 17.2. Определите пространство планирования для задачи с обезьяной и бананом (см. главу 2), в которой применяются действия “walk” (перейти), “push” (передвинуть), “climb” (залезть) и “grasp” (схватить).

## 17.2. Разработка планов с помощью анализа целей и средств

Рассмотрим начальное состояние задачи планирования (см. рис. 17.1). Предположим, что цель состоит в следующем: `on( a, b)`. Задача планировщика заключается в том, что он должен найти план (т.е. последовательность действий), который позволяет достичь этой цели. Типичный планировщик формирует свои рассуждения, как описано ниже.

1. Найти действие, которое позволит достичь состояния `on( a, b)`. Изучение отношения `adds` показывает, что такое действие имеет форму

```
move(a, From, b)
```

для любой позиции `From`. Соответствующее действие, безусловно, должно быть частью плана, но его не всегда возможно осуществить немедленно в указанном начальном состоянии.

2. Теперь создадим возможность выполнения действия `move( a, From, b)`. Рассмотрим отношение `can`, чтобы найти предпосылки этого действия. Они состоят в следующем:

```
[clear(a), clear(b), on(a, From)]
```

В этом начальном состоянии уже имеются связи `clear( b)` и `on( a, From)` (где `From = 1`), но нет связи `clear( a)`. Теперь планировщик сосредоточивается на связи `clear( a)` как новой цели, которая должна быть достигнута.

3. Снова рассмотрим отношение `adds`, чтобы найти действие, позволяющее достичь цели `clear( a)`. Это — любое действие в следующей форме:

```
move(Block, a, To)
```

Предпосылки этого действия состоят в следующем:

```
[clear(Block), clear(To), on(Block, a)]
```

Эти предпосылки удовлетворяются в рассматриваемой начальной ситуации, если имеет место:

Block = c и To = 2

Поэтому в начальном состоянии может быть выполнено действие `move( c, a, 2)`, которое приводит к новому состоянию. Это новое состояние получено из начального состояния следующим образом:

- удаление из начального состояния всех связей, которые разрушаются в результате действия `move( c, a, 2)`;
- добавление в итоговый список всех связей, которые создаются в результате этого действия.

При этом создается такой список:

```
[clear(a), clear(b), clear(c), clear(4), on(a, 1), on(b, 3), on(c, 2)]
```

Теперь может быть выполнено действие `move( a, 1, b)`, в результате которого достигается конечная цель `on( a, b)`. Найденный план можно представить в виде следующего списка:

```
[move(c, a, 2), move(a, 1, b)]
```

Такой стиль формирования рассуждений называется *анализом целей и средств*. В качестве средств рассматриваются возможные действия, а в качестве целей — достигаемые состояния. Обратите внимание на то, что в приведенном выше примере правильный план был найден немедленно, без какого-либо перебора с возвратами. Таким образом, этот пример показывает, что данный способ формирования рассуждений о целях и результатах действий ориентирует процесс планирования в правильном направлении. К сожалению, нельзя утверждать, что при использовании такого способа всегда можно избежать перебора с возвратами. Справедливо противоположное утверждение, что комбинаторная сложность и поиск — это типичные атрибуты планирования.

Принцип планирования с применением анализа целей и средств иллюстрируется на рис. 17.2. Он может быть сформулирован, как описано ниже.

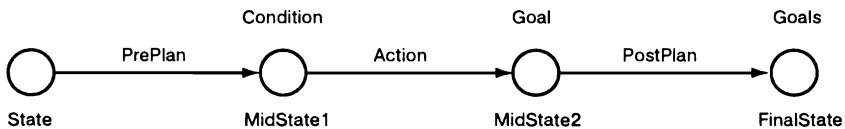


Рис. 17.2. Принцип планирования с применением анализа целей и средств

Чтобы найти в состоянии `State` решения для списка целей `Goals`, ведущие к состоянию `FinalState`, необходимо применить описанную ниже процедуру.

Если все цели `Goals` в состоянии `State` являются истинными, то `FinalState` = `State`. В противном случае выполнить следующие действия.

1. Выбрать в списке `Goals` цель `Goal`, для которой все еще не найдено решение.
2. Найти действие `Action`, которое добавляет цель `Goal` к текущему состоянию.
3. Обеспечить возможность выполнения действия `Action`, решив задачу создания предпосылок `Condition` действия `Action`, что приводит к созданию промежуточного состояния `MidState1`.
4. Применить действие `Action` к состоянию `MidState1` и получить состояние `MidState2` (в состоянии `MidState2` цель `Goal` является истинной).
5. Найти решения для целей в списке `Goals` в состоянии `MidState2`, что приведет к конечному состоянию `FinalState`.



Этот принцип может быть реализован в программе на языке Prolog, приведенной в листинге 17.3, в виде следующей процедуры:

```
plan(State, Goals, Plan, FinalState)
```

где State и FinalState начальное и конечное состояния плана, Goals — список достигаемых целей, а Plan — список действий, позволяющих достичь этих целей. Следует отметить, что в этой программе планирования предполагается использовать определение пространства планирования, в котором все действия и цели являются полностью конкретизированными, т.е. не содержат каких-либо переменных. Для переменных требуется более сложная организация программы. Эта тема рассматривается ниже.

### Листинг 17.3. Простой планировщик с применением анализа целей и средств

```
% Простой планировщик с применением анализа целей и средств
% plan(State, Goals, Plan, FinalState)

plan(State, Goals, [], State) :- % План пуст
 satisfied(State, Goals). % В состоянии State цели Goals истинны

% На основании того, какой способ декомпозиции плана на этапы используется
% в предикате conc, можно считать, что поиск плана PrePlan, создающего
% предпосылки для действия Action, осуществляется в режиме поиска в ширину.
% Но длина остальной части плана не ограничена, и достижение целей происходит
% по принципу поиска в глубину

plan(State, Goals, Plan, FinalState) :-
 conc(PrePlan, [Action | PostPlan], Plan), % Выполнить декомпозицию плана
 select(State, Goals, Goal), % Выбрать цель
 achieves(Action, Goal), % Соответствующее действие
 can(Action, Condition),
 plan(State, Condition, PrePlan, MidState1), % Создать предпосылки для
 % действия Action
 apply(MidState1, Action, MidState2), % Применить действие Action
 plan(MidState2, Goals, PostPlan, FinalState). % Достичь остальных целей

% satisfied(State, Goals): в состоянии State цели Goals являются истинными

satisfied(State, []).

satisfied(State, [Goal | Goals]) :-
 member(Goal, State),
 satisfied(State, Goals).

select(State, Goals, Goal) :-
 member(Goal, Goals),
 not member(Goal, State). % Цель Goal еще не достигнута

% achieves(Action, Goal): цель Goal - это список добавления для действия Action

achieves(Action, Goal) :-
 adds(Action, Goals),
 member(Goal, Goals).

% apply(State, Action, NewState):
% действие Action, выполненное в состоянии State, создает
% новое состояние NewState

apply(State, Action, NewState) :-
 deletes(Action, DelList),
 delete_all(State, DelList, State1), !,
 adds(Action, AddList),
 conc(AddList, State1, NewState).

% delete_all(L1, L2, Diff):
```

% Diff - это разность множеств, которые определены в виде списков L1 и L2

```
delete_all([], _, []).
```

```
delete_all([X | L1], L2, Diff) :-
 member(X, L2), !,
 delete_all(L1, L2, Diff).
```

```
delete_all([X | L1], L2, [X | Diff]) :-
 delete_all(L1, L2, Diff).
```

---

Теперь этим планировщиком можно воспользоваться для поиска плана по установке блока а на блок b, начиная с начального состояния, показанного на рис. 17.1, следующим образом:

```
?- Start = [clear(2), clear(4), clear(b), clear(c), on(a, 1), on(b, 3),
on(c, a)], plan(Start, [on(a, b)], Plan, FinState).
Plan = [move(c, a, 2), move(a, 1, b)]
FinState = [on(a, b), clear(1), on(c, 2), clear(a), clear(4), clear(c),
on(b, 3)]
```

Для того чтобы применить данный планировщик в мире пользователей фотокамеры, представим себе начальное состояние, в котором аккумулятор разряжен, а пленка в фотокамере уже использована. Ниже приведены запросы для поиска плана замены аккумулятора, FixBattery, и плана подготовки фотокамеры для получения снимков, FixCamera.

```
?- Start = [camera_in_case, slot_closed(film), slot_closed(battery),
in(film), film_at_end, in(battery)],
plan(Start, [ok(battery)], FixBattery, _).
FixBattery = [open_case, open_slot(battery), remove(battery),
insert_new(battery)]
?- Start = [camera_in_case, slot_closed(film), slot_closed(battery),
in(film), film_at_end, in(battery)],
can(take_pictures, CameraReady), % Условие для фотографирования
plan(Start, CameraReady, FixCamera, FinState).
CameraReady = [in(film), film_at_start, film_unused, in(battery),
ok(battery), slot_closed(film), slot_closed(battery)]
FixCamera = [open_case, rewind, open_slot(film), remove(film),
insert_new(film), open_slot(battery), remove(battery),
insert_new(battery), close_slot(film), close_slot(battery)]
FinState = [slot_closed(battery), slot_closed(film), in(battery),
ok(battery), in(film), film_at_start, film_unused, camera_outside_case]
```

Весь этот процесс прошел очень гладко, и в обоих случаях найдены кратчайшие планы. Но в дальнейших экспериментах с этим планировщиком обнаруживаются некоторые сложности. Недостатки рассматриваемой программы и возможные пути ее усовершенствования описаны в следующем разделе.

## Упражнение

- 17.3. Выполните ручную трассировку процесса планирования с применением анализа целей и средств для достижения цели `on( a, 3)` из начального состояния, показанного на рис. 17.1.

## 17.3. Защита целей

После проведения экспериментов по использованию данного планировщика в мире блоков и в мире фотокамеры обнаруживается, что мир блоков является гораздо более сложным. Это может на первый взгляд показаться удивительным, поскольку определение мира блоков выглядит проще по сравнению с миром фотокамеры. Истинная причина большей сложности мира блоков заключается в том, что он характе-

ризуется большим количеством комбинаторных вариантов. В мире блоков планировщик обычно вынужден выбирать среди весьма значительного количества действий, которые являются возможными с точки зрения принципа целей и средств, а чем больше вариантов выбора, тем выше комбинаторная сложность. Поэтому эксперименты с планировщиком в мире блоков предьявляют к этой программе более высокие требования и позволяют выявить некоторые ее недостатки.

Еще раз вернемся к задаче, показанной на рис. 17.1. Предположим, что Start — это описание начального состояния на рис. 17.1. В таком случае данную задачу можно сформулировать в виде следующей цели:

```
plan(Start, [on(a, b), on(b, c)], Plan, _)
```

План, найденный планировщиком, показан ниже.

```
Plan = [move(b, 3, c),
move(b, c, 3),
move(c, a, 2),
move(a, 1, b),
move(a, b, 1),
move(b, 3, c),
move(a, 1, b)]
```

Безусловно, что этот план, состоящий из семи этапов, является далеко не самым удачным! Для кратчайшего возможного плана решения этой задачи требуются только три этапа. Проанализируем, почему для данного планировщика потребовалось так много целей. Как показано ниже, причина состоит в том, что эта программа на разных этапах планирования преследует различные цели, как показано ниже.

```
move(b, 3, c), чтобы достичь цели on(b, c)
move(b, c, 3), чтобы достичь цели clear(c) для обеспечения
дальнейшего перемещения
move(c, a, 2), чтобы достичь цели clear(a) для обеспечения
возможности выполнения действия move(a, 1, b)
move(a, 1, b), чтобы достичь цели on(a, b)
move(a, b, 1), чтобы достичь цели clear(b) для обеспечения
возможности выполнения действия move(b, 3, c)
move(b, 3, c), чтобы достичь цели on(b, c) (повторно)
move(a, 1, b), чтобы достичь цели on(a, b) (повторно)
```

Обнаруживающийся здесь недостаток состоит в том, что планировщик иногда уничтожает цели, которые уже были достигнуты. Планировщик легко достиг одной из двух заданных целей, `on( b, c)`, но затем немедленно ее уничтожил, приступив к работе над другой целью, `on( a, b)`. После этого он снова перешел к попыткам достичь цели `on( b, c)`. Она была достигнута за два хода, но между тем была разрушена цель `on( a, b)`. К счастью, в следующий раз цель `on( a, b)` была достигнута без повторного разрушения цели `on( b, c)`. Такое довольно неорганизованное поведение в следующем примере приводит к еще более ярко выраженным отрицательным результатам — к полной неудаче:

```
plan(Start, [clear(2), clear(3)], Plan, _)
```

Теперь планировщик до бесконечности продлевает показанную ниже последовательность действий.

```
move(b, 3, 2), чтобы достичь цели clear(3)
move(b, 2, 3), чтобы достичь цели clear(2)
move(b, 3, 2), чтобы достичь цели clear(3)
move(b, 2, 3), чтобы достичь цели clear(2)
...
```

После каждого действия достигается одна из целей и вместе с тем разрушается другая. К сожалению, пространство планирования определено таким образом, что при выборе способа перемещения блока `b` из его текущей позиции в новую позицию места 2 и 3 всегда рассматриваются в первую очередь.

Из приведенных выше примеров следует одна очевидная идея, что планировщик должен всегда пытаться сохранить уже достигнутые цели. Эту задачу можно решить,

сопровождая список уже достигнутых целей и в дальнейшем избегая таких действий, которые уничтожают цели в этом списке. Поэтому введем новый параметр в отношение `plan`, таким образом:

```
plan(State, Goals, ProtectedGoals, Plan, FinalState)
```

где `ProtectedGoals` — это список целей, “защищаемых” планом `Plan`. Это означает, что ни одно из действий в плане `Plan` не должно удалять ни одну из целей в списке `ProtectedGoals`. После достижения новой цели она добавляется к списку защищенных целей. Программа, приведенная в листинге 17.4, представляет собой модификацию программы планировщика (см. листинг 17.3) с реализованной защитой целей. Теперь задача освобождения мест 2 и 3 решается с помощью следующего плана, состоящего из двух этапов:

```
move(b, 3, 2), чтобы достичь цели clear(3)
move(b, 2, 4), чтобы достичь цели clear(2) и вместе с тем
 защитить цель clear(3)
```

#### Листинг 17.4. Планировщик с применением анализа целей и средств, в котором осуществляется защита целей. Предикаты `satisfied`, `select`, `achieves` и `apply` приведены в листинге 17.3

```
% Планировщик с применением анализа целей и средств, в котором
% осуществляется защита целей

plan(InitialState, Goals, Plan, FinalState) :-
 plan(InitialState, Goals, [], Plan, FinalState).

% plan(InitialState, Goals, ProtectedGoals, Plan, FinalState):
% цели Goals являются истинными в состоянии FinalState, защищенные
% цели никогда не разрушаются планом Plan

plan(State, Goals, _, [], State) :-
 satisfied(State, Goals). % Цели Goals являются истинными
 % в состоянии State

plan(State, Goals, Protected, Plan, FinalState) :-
 conc(PrePlan, [Action | PostPlan], Plan), % Выполнить декомпозицию плана
 select(State, Goals, Goal), % Выбрать недостигнутую цель
 achieves(Action, Goal),
 can(Action, Condition),
 preserves(Action, Protected), % Не уничтожать защищенные цели
 plan(State, Condition, Protected, PrePlan, MidState1),
 apply(MidState1, Action, MidState2),
 plan(MidState2, Goals, [Goal | Protected], PostPlan, FinalState).

% preserves(Action, Goals):
% действие Action не приводит к разрушению ни одной из целей Goals

preserves(Action, Goals) :- % Действие Action не разрушает цели Goals
 deletes(Action, Relations),
 not (member(Goal, Relations),
 member(Goal, Goals)).
```

Очевидно, что эта программа работает гораздо лучше, чем прежняя, хотя все еще не позволяет найти оптимальное решение, поскольку фактически требуется только одно действие — `move( b, 3, 4)`.

Слишком длинные планы являются следствием стратегии поиска, применяемой в рассматриваемом планировщике. Для оптимизации длины планов необходимо изучить поведение планировщика в процессе поиска. Эта задача будет выполнена в следующем разделе.

## 17.4. Процедурные аспекты и режим поиска в ширину

В программах планировщиков, приведенных в листингах 17.3 и 17.4, по сути используются стратегии поиска в глубину, но не в полной мере. Для того чтобы получить исчерпывающее представление о том, что происходит, необходимо определить, в какой последовательности планировщик вырабатывает возможные планы. В этом отношении очень важной является следующая цель в процедуре `plan`:

```
conc(PrePlan, [Action | PostPlan], Plan)
```

В данный момент переменная `Plan` еще не конкретизирована и процедура `conc` вырабатывает с помощью перебора с возвратами альтернативные варианты для переменной `PrePlan` в следующем порядке:

```
PrePlan = [];
PrePlan = [_];
PrePlan = [_, _];
PrePlan = [_, _, _];
...
```

Вначале появляются более короткие варианты для переменной `PrePlan`. Переменная `PrePlan` устанавливает предпосылки для действия `Action`. Этот процесс сводится к поиску действия, предпосылки которого могут быть созданы с помощью наиболее короткого плана из всех возможных (по методу итеративного углубления). С другой стороны, список вариантов для переменной `PostPlan` является полностью неконкретизированным и поэтому его длина не ограничена. Таким образом, результирующее поведение процедуры поиска представляет собой в “глобальном аспекте” поиск в глубину, а в “локальном аспекте” — поиск в ширину. Он является поиском в глубину по отношению к определению с помощью прямого логического вывода тех действий, которые добавляются к формируемому плану. Предпосылки для каждого действия создаются с помощью “предварительного плана”. С другой стороны, поиск этого предварительного плана происходит в режиме поиска в ширину.

Один из способов максимального сокращения длины планов состоит в том, что планировщик можно вынудить использовать режим поиска в ширину таким образом, чтобы все короткие варианты планов рассматривались прежде, чем длинные. Такую стратегию можно реализовать, встроив планировщик в процедуру, которая вырабатывает варианты планов в порядке возрастания длины. Например, следующая программа приводит к реализации режима итеративного углубления:

```
breadth_first_plan(State, Goals, Plan, FinalState) :-
 candidate(Plan), % Вначале формировать короткие планы
 plan(State, Goals, Plan, FinalState).

candidate([]).
candidate([First | Rest]) :-
 candidate(Rest).
```

Более изящное решение состоит в том, что в программе можно достичь аналогичного эффекта, вставив соответствующий генератор вариантов плана непосредственно в процедуру `plan`. Такой подходящий генератор представляет собой следующую процедуру:

```
conc(Plan, _, _)
```

Эта процедура с помощью перебора с возвратами вырабатывает списки действий все возрастающей длины. Теперь в программу планировщика (см. листинг 17.3) можно внести следующие изменения:

```
plan(State, Goals, Plan, FinState) :-
 conc(Plan, _, _), % В первую очередь формировать более короткие списки
 % возможных действий
 conc(PrePlan, [Action | PostPlan], Plan),
 ...
```

Аналогичная модификация позволяет также перевести в режим поиска в ширину программу планировщика с защитой целей, приведенную в листинге 17.4.

Проверим работу модифицированных планировщиков, которые теперь работают в режиме поиска в ширину, на двух примерах задач. Если предположить, что Start — это начальное состояние, показанное на рис. 17.1, то цель

```
plan(Start, [clear(2), clear(3)], Plan, _)
```

приводит к получению такого плана:

```
Plan = [move(b, 3, 4)]
```

Данный план уже является оптимальным. Но задача, показанная на рис. 17.1, все еще является источником определенных проблем. После постановки цели

```
plan(Start, [on(a, b), on(b, c)], Plan, _)
```

вырабатывается такой план:

```
move(c, a, 2)
move(b, 3, a)
move(b, a, c)
move(a, 1, b)
```

Этот результат был получен от обоих планировщиков, с защитой и без защиты целей, работающих в режиме поиска в ширину. Второе из приведенных выше действий является излишним и, безусловно, не имеет смысла. Рассмотрим, как оно вообще попало в план и почему даже поиск в ширину приводит к созданию плана, более длинного, чем оптимальный.

Для этого необходимо ответить на два вопроса. Во-первых, в результате какой цепочки рассуждений планировщик приходит к формированию приведенного выше не совсем разумного плана? Во-вторых, почему планировщик не находит оптимальный план, в который не включено это загадочное действие `move( b, 3, a)`? Начнем с поиска ответа на первый вопрос. Последнее действие, `move( a, 1, b)`, позволяет достичь цели `on( a, b)`, а первые три действия создают предпосылки для действия `move( a, 1, b)`, в частности, создают состояние `clear( a)`. После третьего действия открывается верхняя грань блока `a`, при этом частью предпосылок для третьего действия является `on( b, a)`. Такое состояние достигается с помощью второго действия, `move( b, 3, a)`. Первое действие открывает верхнюю грань блока `a` для обеспечения возможности второго действия. Таким образом, найдено объяснение хода рассуждений, лежащего в основе полученного громоздкого плана, а также показано, какие странные “идеи” могут обнаруживаться во время планирования с помощью анализа целей и средств.

Второй вопрос состоит в том, почему после действия `move( c, a, 2)` планировщик не приступает немедленно к рассмотрению действия `move( b, 3, c)`, которое ведет к созданию оптимального плана. Причина этого состоит в том, что планировщик постоянно работает над целью `on( a, b)`. Действие `move( b, 3, c)` является полностью излишним с точки зрения достижения этой цели, и поэтому попытки его использования не предпринимаются. В этом четырехэтапном плане достигается цель `on( a, b)`, а также, по стечению обстоятельств, — цель `on( b, c)`. Поэтому достижение цели `on( b, c)` является исключительно результатом удачи, а не каких-либо сознательных усилий со стороны планировщика. Слепо преследуя лишь цель `on( a, b)` и относящиеся к этому предпосылки, планировщик не видит причин для выполнения действия `move( b, 3, c)` перед действием `move( b, 3, a)`.

Из приведенного выше примера следует, что механизм планирования по принципу целей и средств в том виде, в каком он реализован в рассматриваемых планировщиках, является неполным. Он не предоставляет для процесса планирования возможность проверить все возможные действия, ведущие к цели. Причина этого состоит в его узкой направленности. Планировщик рассматривает только те действия, которые относятся к текущей цели, и игнорирует другие цели до тех пор, пока текущая цель не будет достигнута. Поэтому он не вырабатывает планы, в которых чередуются действия, относящиеся к разным целям (кроме как в результате счастливо-

го стечения обстоятельств). Основным способом достижения требуемой полноты анализа, которая гарантирует включение оптимальных планов в реализуемую схему планирования, является обеспечение взаимодействия различных целей. Такая задача будет решена в следующем разделе с помощью механизма регрессии целей.

## Упражнение

## 17.5. Регрессия целей

состояние S0: Goals0 ----> состояние S: Goals

1. В состоянии  $S_0$  должно быть возможно действие  $A$ , поэтому цели  $Goals_0$  должны формировать предпосылки для  $A$ .
2. Для каждой цели  $G$  в списке  $Goals$  должно быть справедливо одно из следующих утверждений:
  - цель  $G$  добавлена в результате действия  $A$ ;
  - цель  $G$  имеется в списке  $Goals_0$ , и выполнение действия  $A$  не привело к ее удалению.

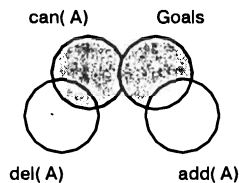


Рис. 17.3. Отношения между различными множествами условий при осуществлении регрессии целей с помощью действия A. Затененная область показывает соответствующие цели в списке Goals0, полученные с результатом регрессии:  $Goals0 = can(A) \cup Goals - add(A)$ . Обратите внимание на то, что пересечение между списком Goals и списком удаления действия A должно быть пустым

Механизм регрессии целей может использоваться при планировании, как описано ниже.

Чтобы достичь списка целей Goals из некоторого начального состояния StartState, необходимо выполнить следующие действия.

- Если в состоянии StartState все цели в списке Goals являются истинными, то достаточно применить пустой план.
- В противном случае выбрать цель G из списка Goals и некоторое действие A, в результате которого в этот список добавляется цель G. Затем выполнить регрессию целей в списке Goals с помощью действия A, получив список NewGoals, и найти план для достижения целей списка NewGoals из состояния StartState.

Эту стратегию можно улучшить, заметив, что некоторые комбинации целей являются невозможными. Например, цели `on( a, b)` и `clear( b)` не могут быть истинными одновременно. Такое условие можно сформулировать в виде следующего отношения:

```
impossible(Goal, Goals)
```

Это отношение указывает, что цель Goal является невозможной в сочетании с целями Goals, т.е. цели Goal и Goals никогда не будут достигнуты вместе, поскольку они несовместимы. Для рассматриваемого мира блоков такие несовместимые комбинации могут быть определены следующим образом:

```
impossible(on(X, X), _). % Блок не может быть поставлен сам на себя
impossible(on(X, Y), Goals) :-
 member(clear(Y), Goals)
;
member(on(X, Y1), Goals), Y1 \== Y % Блок не может находиться
 % одновременно в двух местах
;
member(on(X1, Y), Goals), X1 \== X. % Два блока не могут находиться
 % одновременно в одном и том же месте
impossible(clear(X), Goals) :-
 member(on(_, X), Goals).
```

Программа планировщика, основанная на принципах регрессии целей, описанных выше, приведена в листинге 17.5. В этой программе возможные планы рассматриваются в режиме поиска в ширину, при котором предпринимается попытка в первую очередь найти самые короткие планы. Реализация этого требования обеспечивается благодаря использованию в процедуре `plan` следующей цели:

```
conc(PrePlan, [Action], Plan)
```



Такой планировщик находит оптимальный трехэтапный план для задачи, показанной на рис. 17.1.

### Листинг 17.5. Планировщик, основанный на регрессии целей, осуществляет поиск в режиме поиска в ширину

```
% Планировщик с регрессией целей, работающий в режиме поиска в ширину
% plan(State, Goals, Plan)
plan(State, Goals, []) :-
 satisfied(State, Goals). % Цели Goals являются истинными
 % в состоянии State

plan(State, Goals, Plan) :-
 conc(PrePlan, [Action], Plan), % Выполнить декомпозицию плана,
 % достигая эффекта поиска в ширину
 select(State, Goals, Goal), % Выбрать цель
 achieves(Action, Goal),
 can(Action, Condition), % Обеспечить отсутствие переменных
 % в действии Action
 preserves(Action, Goals), % Защитить цели Goals
 regress(Goals, Action, RegressedGoals), % Выполнить регрессию целей Goals
 % с помощью действия Action
 plan(State, RegressedGoals, PrePlan).

satisfied(State, Goals) :-
 delete_all(Goals, State, []). % Все цели Goals, которые определены
 % в состоянии State

select(State, Goals, Goal) :- % Выбрать цель Goal из списка целей Goals
 member(Goal, Goals). % Простой принцип выбора

achieves(Action, Goal) :-
 adds(Action, Goals),
 member(Goal, Goals).

preserves(Action, Goals) :- % Действие Action не приводит
 % к уничтожению цели Goals
 deletes(Action, Relations),
 not (member(Goal, Relations),
 member(Goal, Goals)).

regress(Goals, Action, RegressedGoals) :- % Выполнить регрессию целей Goals
 % с помощью действия Action
 adds(Action, NewRelations),
 delete_all(Goals, NewRelations, RestGoals),
 can(Action, Condition),
 addnew(Condition, RestGoals, RegressedGoals). % Ввести дополнительные
 % предпосылки, проверить возможность действий

% addnew(NewGoals, OldGoals, AllGoals):
% OldGoals - это объединение целей NewGoals и OldGoals;
% цели NewGoals и OldGoals должны быть совместимыми

addnew([], L, L).

addnew([Goal | _], Goals, _) :-
 impossible(Goal, Goals), % Цель Goal несовместима с целями Goals
 !,
 fail. % Добавление невозможно

addnew([X | L1], L2, L3) :-
 member(X, L2), !, % Игнорировать дубликат
 addnew(L1, L2, L3).

addnew([X | L1], L2, [X | L3]) :-
 addnew(L1, L2, L3).
```

```
% delete_all(L1, L2, Diff):
% Diff- это разность множеств, которые определены в виде списков L1 и L2

delete_all([], _, []).

delete_all([X | L1], L2, Diff) :-
 member(X, L2), !,
 delete_all(L1, L2, Diff).

delete_all([X | L1], L2, [X | Diff]) :-
 delete_all(L1, L2, Diff).
```

---

## Упражнение

17.5. Выполните трассировку процесса планирования, основанного на регрессии целей, для достижения цели `on( a, b)` из начального состояния, показанного на рис. 17.1. Предположим, что этот план состоит в следующем:

```
[move(c, a, 2), move(a, 1, b)]
```

Если список целей после выполнения второго действия плана представляет собой `[ on( a, b)]`, то каковым является регрессированный (переведенный в предшествующее состояние) список целей перед вторым и перед первым действием?

## 17.6. Сочетание планирования по принципу анализа целей и средств с эвристическим поиском по заданному критерию

В планировщиках, рассматриваемых до сих пор, использовались лишь очень простые стратегии поиска: поиск в глубину или поиск в ширину (с итеративным углублением) или сочетание этих двух стратегий. Такие стратегии являются полностью нецеленаправленными в том смысле, что в них при обосновании выбора среди множества вариантов не применяются какие-либо знания, касающиеся той или иной проблемной области. Поэтому они являются очень неэффективными, за исключением некоторых частных случаев. Может рассматриваться несколько возможных способов введения в эти планировщики эвристического управления, основанного на знаниях в конкретной проблемной области. Некоторые очевидные участки, на которых в планировщики могут быть введены знания, касающиеся планирования конкретной проблемной области, перечислены ниже.

- Отношение `select( State, Goals, Goal)`, в котором принимается решение по выбору последовательности осуществления попыток достижения целей. Например, одно из полезных сведений о построении конструкций из блоков состоит в том, что в любое время каждый блок должен стоять на надежной опоре и поэтому конструкции необходимо строить в последовательности снизу вверх. Правило эвристического выбора, основанное на знании об этом, должно указывать, что “самые верхние” связи `on` должны формироваться в последнюю очередь (т.е. они должны выбираться планировщиком с регрессией целей в первую очередь, поскольку он формирует планы, переходя от конца плана к его началу). Еще одна эвристика должна подсказывать, что выбор тех целей, которые уже являются истинными в начальном состоянии, должен быть отложен до последнего момента.
- Отношение `achieves( Action, Goal)`, в котором принимается решение о том, какое из альтернативных действий должно быть опробовано для достижения заданной цели. (В рассматриваемых планировщиках варианты фактиче-

ски вырабатываются также при выполнении предиката `can`, когда действия становятся неконкретизированными.) Некоторые действия кажутся лучшими, например, потому, что они позволяют достичь нескольких целей одновременно; на основании своего опыта мы можем указать, что предпосылки некоторых действий проще создать по сравнению с другими.

- Решение о том, какое из альтернативных множеств регрессированных целей должно рассматриваться в следующую очередь, — прежде всего продолжить работу над тем из них, которое выглядит как более простое, поскольку именно с его помощью, вероятно, удастся достичь более короткого плана.

Последняя возможность показывает, каким образом можно реализовать в планировщике режим поиска по заданному критерию. Для этого требуется оценивать с помощью эвристических функций сложность альтернативных множеств целей, а затем продолжать разворачивать наиболее перспективное из альтернативных множеств целей.

Чтобы воспользоваться программой поиска по заданному критерию (см. главу 12), необходимо формализовать соответствующее пространство состояний и эвристическую функцию; иными словами, необходимо определить перечисленные ниже компоненты.

1. Отношение выбора преемника между узлами в пространстве состояний `s( Node1, Node2, Cost)`.
2. Целевые узлы поиска, заданные с помощью отношения `goal( Node)`.
3. Эвристическая функция, представленная в виде отношения `h( Node, HeuristicEstimate)`.
4. Начальный узел поиска.

Одним из способов формирования такого пространства состояний является определение соответствия между множествами целей и узлами в пространстве состояний. При этом в пространстве состояний должна существовать связь между двумя множествами целей, `Goals1` и `Goals2`, если есть такое действие `A`, для которого справедливы следующие утверждения.

1. Действие `A` добавляет некоторую цель в множество `Goals1`.
2. Действие `A` не уничтожает ни одной цели в множестве `Goals1`.
3. Множество `Goals2` является результатом регрессии множества `Goals1` с помощью действия `A`, как определено отношением `regress`, приведенным в листинге 17.5:

```
regress(Goals1, A, Goals2)
```

Для упрощения предположим, что все действия имеют одинаковую стоимость, поэтому присвоим стоимость 1 всем связям в пространстве состояний. В таком случае отношение определения преемника `s` должно выглядеть примерно так:

```
s(Goals1, Goals2, 1) :-
 member(Goal, Goals1), % Выбрать цель
 achieves(Action, Goal), % Соответствующее действие
 can(Action, Condition),
 preserves(Action, Goals1),
 regress(Goals1, Action, Goals2).
```

Любое множество целей, которое является истинным в начальном состоянии плана, представляет собой целевой узел поиска в пространстве состояний. Начальным узлом для поиска является список целей, которые должны быть достигнуты с помощью плана.

Хотя описанное выше представление содержит всю необходимую информацию, оно имеет один небольшой недостаток. Он связан с тем фактом, что применяемая программа поиска по заданному критерию находит путь решения как последовательность состояний и не включает действия, обеспечивающие переход между состояниями. Например, последовательность состояний (списков целей) для достижения

состояния `on( a, b)` из начального состояния, показанного на рис. 17.1, является таковой:

```
[[clear(c), clear(2), on(c, a), clear(b), on(a, 1)], % Эти цели являются
 % истинными в начальном состоянии
[clear(a), clear(b), on(a, 1)], % Цели, истинные после выполнения
 % действия move(c, a, 2)
[on(a, b)]] % Цели, истинные после выполнения
 % действия move(a, 1, b)
```

Обратите внимание на то, что эта программа поиска по заданному критерию возвращает путь решения в обратном порядке. В данном случае это удобно, поскольку планы формируются от конца к началу, и в связи с этим обратная последовательность, возвращенная программой поиска, соответствует фактическому порядку действий в плане. Но не совсем удобно то, что действия в плане явно не упоминаются, хотя и могут быть реконструированы с учетом различий в указанных списках целей. Тем не менее действия можно легко ввести в такое описание пути решения. Для этого достаточно добавить в каждое состояние действие, которое вызывает переход в это состояние. Поэтому в качестве узлов в пространстве состояний применяются пары в следующей форме:

Goals -> Action

Такое уточненное представление используется в реализации пространства состояний, которое показано в листинге 17.6. В этой реализации применяется очень грубая эвристическая функция, которая определяет количество еще не конкретизированных целей в списке целей.

**Листинг 17.6. Определение пространства состояний для задачи планирования по принципу целей и средств с использованием регрессии целей. Отношения `satisfied`, `achieves`, `preserves`, `regress`, `addnew` и `delete_all` приведены в листинге 17.5**

---

```
% Определение пространства состояний для задачи планирования по принципу
% целей и средств с использованием регрессии целей
:- op(300, xfy, ->).

s(Goals -> NextAction, NewGoals -> Action, 1) :- % Все стоимости равны 1
 member(Goal, Goals),
 achieves(Action, Goal),
 can(Action, Condition),
 preserves(Action, Goals),
 regress(Goals, Action, NewGoals).

goal(Goals -> Action) :-
 start(State), % Определяемое пользователем начальное состояние
 satisfied(State, Goals). % Цели Goals являются истинными в начальном
 % состоянии

h(Goals -> Action, H) :- % Эвристическая оценка
 start(State),
 delete_all(Goals, State, Unsatisfied), % Недостигнутые цели
 length(Unsatisfied, H). % Количество недостигнутых целей
```

---

Теперь определение пространства состояний (см. листинг 17.6) можно использовать в программах поиска по заданному критерию (см. главу 12) следующим образом. Необходимо применить для консультации определение задачи планирования в виде отношений `adds`, `deletes` и `can` (которое приведено в листинге 17.1 для мира блоков). Кроме того, программе требуется предоставить отношение `impossible`, а также отношение `start`, которое описывает начальное состояние плана. Для состояния, показанного на рис. 17.1, последнее отношение имеет следующий вид:

```
start([on(a, 1), on(b, 3), on(c, a), clear(b), clear(c), clear(2),
 clear(4)]).
```

Теперь, чтобы решить задачу, показанную на рис. 17.1, с помощью планировщика, работающего по принципу целей и средств с учетом заданного критерия, необходимо вызвать процедуру поиска по заданному критерию следующим образом:

```
?- bestfirst([on(a, b), on(b, c)] -> stop, Plan).
```

Здесь дополнительно введено пустое действие `stop`, поскольку в применяемом представлении за каждым списком целей должно следовать, по требованиям синтаксиса, какое-либо действие. Но `[ on( a, b), on( b, c) ]` — это конечный список целей плана, за которым фактически не следует действие, поэтому приходится применять пустое действие. Ниже приведено решение, которое представляет собой список, состоящий из списков целей и соединяющих их действий.

```
Plan = [
[clear(2), on(c, a), clear(c), on(b, 3), clear(b), on(a, 1)] ->
 move(c, a, 2),
[clear(c), on(b, 3), clear(a), clear(b), on(a, 1)] -> move(b, 3, c),
[clear(a), clear(b), on(a, 1), on(b, c)] -> move(a, 1, b),
[on(a, b), on(b, c)] -> stop]
```

Хотя в этом планировщике по заданному критерию используются лишь простейшие эвристические оценки, он работает намного быстрее по сравнению с другими планировщиками, описанными в этой главе.

## Упражнения

- 17.6. Рассмотрим простейшую эвристическую функцию для мира блоков, приведенную в листинге 17.6. Удовлетворяет ли эта функция условию теоремы допустимости для поиска по заданному критерию? (Теорема допустимости описана в главе 12.)
- 17.7. Применяемая в этой главе в качестве примера эвристическая функция для мира блоков подсчитывает количество целей, которые должны быть достигнуты. Это — очень грубая оценка, поскольку некоторых целей достичь, безусловно, сложнее по сравнению с другими. Например, достичь цели `on( a, b)` весьма просто, если блоки `a` и `b` уже имеют открытую верхнюю грань, но гораздо сложнее, если блоки `a` и `b` спрятаны под высокими столбиками из других блоков. Поэтому лучшая эвристическая функция могла бы предусматривать попытку оценить сложность достижения отдельных целей, например, с учетом количества блоков, которые должны быть сняты с интересующего нас блока, прежде чем появится возможность его переместить. Предложите такие лучшие эвристические функции и проведите с ними эксперименты.
- 17.8. Модифицируйте определение пространства состояний планирования, приведенное в листинге 17.6, чтобы ввести в него стоимость действий, следующим образом:

```
s(State1, State2, Cost)
```

Стоимость `Cost` может, например, зависеть от веса перемещенного объекта и расстояния, на которое он перемещается. Используйте это определение для поиска планов с минимальной стоимостью в мире блоков.

## 17.7. Неконкретизированные действия и планирование с частичным упорядочением

Планировщики, разработанные в этой главе, реализованы в виде программ, составленных настолько просто, насколько это возможно, в основном для иллюстрации расматриваемых принципов. Поэтому не предпринимались какие-либо усилия по повышению их эффективности. Но значительное повышение эффективности программ может быть достигнуто благодаря использованию лучших способов представления и

соответствующих структур данных. Описанные здесь планировщики можно также усовершенствовать с помощью двух других важных механизмов: применение неконкретизированных переменных в целях и действиях и планирование с частичным упорядочением. Эти направления усовершенствования кратко описаны в данном разделе.

## 17.7.1. Неконкретизированные действия и цели

Все алгоритмы, применяемые в этой главе, были значительно упрощены благодаря тому, что к ним предъявлялось требование, чтобы все цели для планировщика всегда были полностью конкретизированы. Это требование соблюдалось в силу того, что использовалось соответствующее ему пространство планирования (отношения `adds`, `deletes` и `can`). Например, в листинге 17.1 полная конкретизация переменных обеспечивается с помощью отношения `can`, которое определено следующим образом:

```
can(move(Block, From, To), [clear(Block), clear(To), on(Block, From)]) :-
 block(Block),
 object(To),
 ...
```

Конкретизацию переменных обеспечивают такие цели, как `block( Block)`, показанная выше. Но конкретизация может привести к выработке многочисленных не относящихся к делу альтернативных действий, которые также должны учитываться планировщиком. Например, рассмотрим ситуацию, показанную на рис. 17.1, в которой к планировщику поступает запрос на достижение цели `clear( a)`. В отношении `achieves` предлагается следующее общее действие для достижения цели `clear( a)`:

```
move(Something, a, Somewhere)
```

В таком случае для поиска предпосылок достижения этой цели применяется следующее отношение:

```
can(move(Something, a, Somewhere), Condition)
```

В результате того, что используется перебор с возвратами, планировщик вынужден рассматривать всевозможные варианты конкретизации переменных `Something` и `Somewhere`. Поэтому проверяются все следующие действия, прежде чем будет найдено одно из них, позволяющее достичь цели:

```
move(b, a, 1)
move(b, a, 2)
move(b, a, 3)
move(b, a, 4)
move(b, a, c)
move(c, a, 1)
move(c, a, 2)
```

Более выразительное представление, позволяющее исключить эти неэффективные операции, может предусматривать применение в целях неконкретизированных переменных. Например, для мира блоков одна из попыток определить такое альтернативное отношение `can` может состоять в следующем:

```
can(move(Block, From, To), [clear(Block), clear(To), on(Block, From)]).
```

Теперь снова рассмотрим ситуацию, показанную на рис. 17.1, и цель `clear( a)`. И в этом случае отношение `achieves` предусматривает использование такого действия:

```
move(Something, a, Somewhere)
```

Но на этот раз при обработке отношения `can` переменные остаются неконкретизированными и список предпосылок достижения цели принимает следующий вид:

```
[clear(Something), clear(Somewhere), on(Something, a)]
```

Обратите внимание на то, что этот список целей, которых теперь должен попытаться достичь планировщик, содержит переменные. Данный список целей в начальном состоянии достигается немедленно, если применяется следующая конкретизация:

```
Something = c
Somewhere = 2
```

Ключом к достигнутому повышению эффективности, при котором правильное действие обнаруживается практически без какого-либо поиска, является то, что множества вариантов действий и целей заменяются неконкретизированными действиями и целями. Их конкретизация откладывается до более позднего времени, когда становится очевидно, какими значениями должны быть конкретизированы эти переменные. С другой стороны, спецификация, приведенная в листинге 17.1, вынуждает программу немедленно выполнять конкретизацию действий, а значит, и целей в предпосылках действий.

Приведенный выше пример показывает, что использование представления с помощью переменных открывает широкие возможности. Но реализация данного метода связана с определенными сложностями. Прежде всего, показанная выше попытка иначе определить отношение `can` является недопустимой, поскольку она позволяет использовать в ситуации, показанной на рис. 17.1, следующее действие:

```
move(c, a, c)
```

В результате возникает невероятное положение, в котором блок `c` должен стоять сам на себе! Поэтому более качественное определение отношения `can` не должно допускать, чтобы была предпринята попытка поставить блок сам на себя, а также должно учитывать все прочие аналогичные ограничения. Такое определение приведено ниже.

```
can(move(Block, From, To),
 [clear(Block), clear(To), on(Block, From),
 different(Block, To), different(From, To), different(Block, From)]).
```

Здесь `different( X, Y)` означает, что переменные `X` и `Y` не могут обозначать один и тот же объект. Условие наподобие `different( X, Y)` не зависит от состояния мира. Поэтому его значение не может стать истинным в результате какого-либо действия, но его соблюдение должно контролироваться путем проверки соответствующего предиката. Один из способов учета подобных фиктивных целей состоит во введении следующего дополнительного предложения в процедуру `satisfied`, применяемую в рассматриваемых планировщиках:

```
satisfied(State, [Goal | Goals]) :-
 holds(Goal), % Цель Goal независима от состояния State
 satisfied(Goals).
```

В соответствии с этим, такие предикаты, как `different( X, Y)`, должны быть определены процедурой `holds` следующим образом:

```
holds(different(X, Y))
```

Подобное определение может быть сформулировано в соответствии с приведенными ниже рекомендациями.

- Если `X` и `Y` не согласуются, то предикат `different( X, Y)` принимает истинное значение.
- Если `X` и `Y` буквально совпадают друг с другом (`X == Y`), то это условие становится ложным и всегда будет оставаться ложным независимо от дальнейших действий в плане. В таком случае подобные условия могут учитываться таким же образом, как и цели, объявленные в отношении `impossible`.
- В противном случае (`X` и `Y` согласуются, но не являются полностью одинаковыми) оказывается, что пока невозможно оценить текущую ситуацию. Решение о том, следует ли использовать эти переменные для обозначения одинакового объекта или этого делать нельзя, должно быть отложено до тех пор, пока переменные `X` и `Y` не станут более конкретизированными.

Как показывает этот пример, проверку условий, подобных `different( X, Y)`, которые являются независимыми от состояния, иногда приходится откладывать. Поэтому было бы целесообразно представлять такие условия в качестве дополнительного параметра процедуры `plan` и учитывать их отдельно от тех целей, которые достигаются с помощью действий.

Это — не единственная сложность, обусловленная введением переменных. Например, рассмотрим следующее действие:

```
move(a, 1, X)
```

Приводит ли это действие к удалению отношения `clear( b)`? Да, если  $X = b$ , и нет, если `different( X, b)`. Это означает, что существуют две возможности и со значением  $X$  связаны два соответствующих варианта: в одном варианте  $X$  равен  $b$ , а в другом вводится дополнительное условие `different( X, Y)`.

## 17.7.2. Планирование с частичным упорядочением

Один из недостатков рассматриваемых планировщиков состоит в том, что в них рассматриваются все возможные варианты упорядочения действий, даже в тех случаях, если эти действия являются полностью независимыми. В качестве примера рассмотрим задачу планирования, показанную на рис. 17.4, в которой целью является составление двух столбиков блоков из двух отдельных наборов блоков, которые уже и так достаточно разделены. Эти два столбика можно составить независимо друг от друга с помощью следующих двух планов, соответствующих каждому отдельному столбику:

```
Plan1 = [move(b, a, c), move(a, 1, b)]
Plan2 = [move(e, d, f), move(d, 8, e)]
```

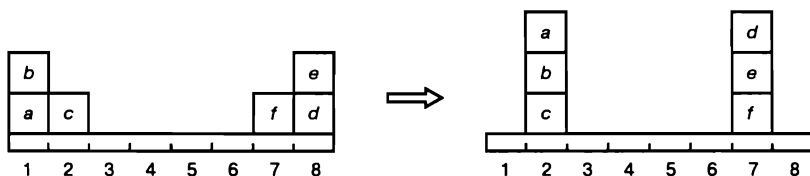


Рис. 17.4. Задача планирования, состоящая из двух независимых подзадач

Важной отличительной особенностью этой ситуации является то, что эти два плана не связаны друг с другом. Поэтому имеет значение лишь порядок действий в каждом отдельном плане, но не важно, в какой последовательности выполняются сами планы: вначале Plan1, а затем Plan2, или вначале Plan2, а затем Plan1, или даже происходит ли переключение между ними и выполняется часть одного плана, а затем часть другого. Например, ниже показана одна из допустимых последовательностей выполнения.

```
[move(b, a, c), move(e, d, f), move(d, 8, e), move(a, 1, b)]
```

Несмотря на это, рассматриваемые в этой главе планировщики в процессе планирования скорее всего будут учитывать все 24 перестановки четырех действий, хотя имеются, по сути, лишь четыре варианта: по две перестановки для каждого из двух независимых планов, составляющих общий план. Возникающая при этом проблема является следствием того факта, что рассматриваемые планировщики строго ориентированы на полное упорядочение всех действий в плане. Поэтому возможное усовершенствование состоит в том, что если в какой-то ситуации порядок действий не имеет значения, то отношение предшествования между действиями может оставаться неопределенным. Таким образом, разрабатываемые планы могут представлять собой частично упорядоченные множества действий, а не полностью упорядоченные последовательности. Планировщики, которые допускают частичное упорядочение, называют *планировщиками с частичным упорядочением* (иногда также *нелинейными планировщиками*).

Рассмотрим основной принцип планирования с частичным упорядочением, снова обратившись к примеру, приведенному на рис. 17.1. Ниже приведено краткое описание того, как нелинейный планировщик может решить эту задачу. Анализируя цели



on( a, b) и on( b, c), планировщик приходит к выводу, что в план должны быть обязательно включены следующие два действия:

```
M1 = move(a, X, b)
M2 = move(b, Y, c)
```

Иного способа достичь этих двух целей не существует. Но порядок, в котором должны быть выполнены два указанных действия, еще не задан. Теперь рассмотрим предпосылки этих двух действий. Предпосылка для действия move( a, X, b) содержит условие clear( a). Это условие не соблюдается в начальном состоянии, поэтому требуется еще какое-то действие в следующей форме:

```
M3 = move(U, a, V)
```

Оно должно предшествовать действию M1, поэтому теперь необходимо учитывать следующее ограничение при выборе последовательности действий:

```
before(M3, M1)
```

После этого рассмотрим, не могут ли действия M2 и M3 заменяться одним и тем же действием, с помощью которого достигаются цели обоих действий. Это условие не соблюдается, поэтому план должен включать три разных действия. Затем планировщик должен ответить на вопрос о том, есть ли такая перестановка трех действий [ M1, M2, M3], что M3 предшествует M1, перестановка выполнима в начальном состоянии и общие цели достигаются в результирующем состоянии. С учетом приведенного выше ограничения предшествования должны рассматриваться только следующие три из общего количества перестановок, равного шести:

```
[M3, M1, M2]
[M3, M2, M1]
[M2, M3, M1]
```

Ограничениям, применяемым в данном сеансе выполнения программы, соответствует вторая из этих перестановок, если используется такая конкретизация: U = c, V = 2, X = 1, Y = 3. Как показывает этот пример, благодаря использованию планирования с частичным упорядочением нельзя полностью устранить комбинаторную сложность, а можно лишь ее уменьшить.

## Проекты

С использованием методов, описанных в этой главе, разработайте программу планирования для применения в более интересном варианте простого мира блоков, который рассматривался в этой главе. На рис. 17.5 показан пример задачи, которая должна быть решена в этом новом мире блоков. Он состоит из блоков разных размеров, и в нем должна учитываться устойчивость конструкций. Чтобы упростить данную задачу, примите предположение, что блоки могут занимать только целое количество позиций и всегда должны стоять на надежной опоре, таким образом, чтобы построенные из них конструкции были полностью устойчивыми. Кроме того, примите предположение, что блоки никогда не бывают повернуты роботом и траектории их перемещения остаются простыми: блок поднимается прямо вверх до тех пор, пока не будет находиться выше всех других блоков, после этого перемещается по горизонтали, а затем опускается прямо вниз. Разработайте специализированные эвристические функции, предназначенные для использования этим планировщиком.

В мире роботов более реальная и интересная задача по сравнению с той, что представлена в листинге 17.1, может также предусматривать действия по восприятию общей картины, осуществляемые с помощью телевизионной камеры или контактного датчика. Например, действие look( Position, Object) позволяет распознать объект, обнаруженный телевизионной камерой в позиции Position (т.е. конкретизировать переменную Object). В подобном мире становится реальным предположение, что сцена действия не полностью известна для робота, поэтому он может включить в свой план такие действия, единственной целью которых является получение информации. Такую задачу можно дополнительно усложнить с учетом того факта, что некоторые наблюдения подобного рода не могут быть выполнены немедленно (напри-

мер, объекты, закрытые другими объектами, нельзя рассмотреть с помощью телевизионной камеры, которая показывает вид сверху). Введите другие соответствующие отношения с определением целей и в случае необходимости внесите исправления в рассматриваемые программы планировщиков.

Откорректируйте планировщик с регрессией целей, приведенный в листинге 17.5, таким образом, чтобы он правильно обрабатывал переменные в целях и действиях, в соответствии с описанием, содержащимся в разделе 17.7.1.

Напишите программу нелинейного планировщика в соответствии с общими рекомендациями, изложенными в разделе 17.7.2.

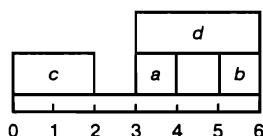


Рис. 17.5. Задача планирования для другого мира блоков: достичь целей  $on(a, c)$ ,  $on(b, c)$ ,  $on(c, d)$

## Резюме

- В планировании возможные действия должны быть представлены с помощью таких средств, которые позволяют явно формировать рассуждения об их результатах и предпосылках. Указанное требование можно выполнить, задавая для каждого действия его *предпосылки*, а также соответствующие ему *списки связей*: список добавления (связи, устанавливаемые этим действием) и список удаления (связи, уничтожаемые этим действием).
- Процедура составления планов по *принципу целей и средств* основана на поиске действий, с помощью которых достигаются заданные цели и создаются предпосылки для таких действий.
- *Защитой целей* называется механизм, который предотвращает разрушение планировщиком уже достигнутых целей.
- Планирование по принципу целей и средств предусматривает поиск в пространстве возможных действий. Поэтому для решения задач планирования могут также применяться обычные *методы поиска*: поиск в глубину, поиск в ширину и поиск по заданному критерию.
- Для *уменьшения сложности поиска* на некоторых этапах планирования по принципу целей и средств могут использоваться знания о данной конкретной проблемной области, которые, в частности, позволяют определить, какой цели в заданном списке целей следует пытаться достичь на очередном этапе и какое действие среди всех вариантов действий необходимо выполнить в первую очередь, а также применить эвристическую оценку сложности достижения целей в списке целей при поиске по заданному критерию.
- *Регрессия целей* — это процесс, позволяющий определить, какие цели должны быть истинными перед выполнением некоторого действия для того, чтобы можно было гарантировать истинность определенных целей после этого действия. В процессе планирования с применением регрессии целей обычно предусматривается обратный логический вывод действий.
- Применение *неконкретизированных переменных* в целях и действиях позволяет повысить эффективность планирования, но, с другой стороны, приводит к значительному усложнению планировщика.

- В подходе к планированию с частичным упорядочением учитывается тот факт, что действия в планах не всегда должны быть полностью упорядоченными. Если последовательность выполнения действий при любой такой возможности остается неопределенной, это позволяет упростить обработку множеств эквивалентных перестановок действий.
- В данной главе рассматриваются следующие понятия:
  - предпосылки действия, списки добавления, списки удаления;
  - планирование по принципу целей и средств;
  - защита целей;
  - регрессия целей;
  - планирование с частичным упорядочением.

## Дополнительные источники информации

Результаты одних из первых исследований основных принципов решения задач и планирования с использованием анализа целей и средств в искусственном интеллекте описаны в [113]. Эти принципы были реализованы в известной программе GPS (General Problem Solver — общий решатель задач); результаты исследования поведения этой программы подробно описаны в [46]. Важную роль в развитии искусственного интеллекта сыграла также программа планирования STRIPS [49], [50], которую можно рассматривать как одну из реализаций GPS. В программе STRIPS применен способ представления пространства планирования (с помощью отношений *adds*, *deletes* и *can*), который используется также и в этой главе. Двумя другими способами представления данных для планирования на основе логики (не рассматриваемыми в данной главе) являются исчисление ситуаций [80] и исчисление событий [82], [143]. Механизмы программы STRIPS и различные относящиеся к ней идеи и усовершенствования описаны в [116], где представлены также изящные формулировки алгоритмов планирования на основе логики, предложенные Грином (Green) и Ковальски (Kowalski). Одним из первых планировщиков на языке Prolog, представляющих значительный интерес, является программа WARPLAN [164]. Эту программу можно рассматривать как еще одну реализацию программы STRIPS, усовершенствованную в определенном отношении. Описание программы WARPLAN можно найти и в других источниках, например в [33]. В [163] приведены результаты исследования феноменов взаимодействия конъюнктивных целей, а в их числе рассматривается также принцип регрессии целей. Задача регрессии целей связана с определением слабейших предпосылок, используемых при доказательстве правильности программы. Сложности, которые могут возникнуть перед планировщиком с защитой целей во время решения задачи, приведенной на рис. 17.1, известны в литературе как *аномалия Зюссмана* (Sussman) (см., например, [163]). Одни из первых результатов разработки процедуры планирования с частичным упорядочением описаны в [134] и [157]. В [29] рассматриваются попытки предоставить единообразную теоретическую инфраструктуру для описания и исследования различных механизмов планирования с частичным упорядочением. В [166] приведен обзор результатов в области планирования с частичным упорядочением. С точки зрения теории желательно гарантировать полноту решений планировщика, но этого можно добиться лишь за счет значительного увеличения комбинаторной сложности. Поэтому намного более перспективный подход с точки зрения практики предложен в [30], где исследуются различные способы применения в планировании знаний о конкретной проблемной области для уменьшения комбинаторной сложности. В [6] приведен отредактированный сборник классических работ по планированию. Большой объем сведений по планированию приведен в таких книгах по искусственному интеллекту общего характера, как [126] и [133]. Очень ценные статьи по планированию иногда появляются в журнале *Artificial Intelligence* (см., например, специальный выпуск по планированию и составлению расписаний [4]).

# Машинное обучение

### В этой главе...

|                                                                                |     |
|--------------------------------------------------------------------------------|-----|
| 18.1. Введение                                                                 | 408 |
| 18.2. Проблема изучения понятий на примерах                                    | 409 |
| 18.3. Подробный пример формирования реляционных описаний в результате обучения | 414 |
| 18.4. Обучение с помощью простых правил вывода                                 | 419 |
| 18.5. Логический вывод деревьев решения                                        | 426 |
| 18.6. Обучение по зашумленным данным и отсечение частей деревьев               | 433 |
| 18.7. Успех обучения                                                           | 439 |

Из всех форм обучения освоение понятий на примерах является самым распространенным и наглядным. Выбор алгоритмов обучения зависит от того, на каком языке представлены изучаемые понятия. В данной главе приведены программы, способные к изучению понятий, представленных в виде правил вывода и деревьев решения. В ней также рассматривается отсечение частей деревьев решения как метод обучения по зашумленным данным в тех условиях, когда примеры могут содержать ошибки.

## 18.1. Введение

Существует несколько форм обучения, начиная с *усвоения сообщенных знаний* и заканчивая *обучением в результате открытия*. В первом случае ученику явно сообщают понятия, которые он должен усвоить. В этом смысле программирование представляет собой своего рода обучение путем усвоения сообщенных знаний. Основная нагрузка при обучении такого типа возлагается на учителя, хотя задача ученика может также оказаться трудной, поскольку ему иногда нелегко понять, что хочет сказать учитель. Поэтому при обучении путем усвоения сообщенных знаний может потребоваться интеллектуальное взаимодействие ученика и учителя, в том числе наличие у ученика достаточного объема знаний о самом учителе. На другом краю спектра методов обучения, при использовании обучения в результате открытия, ученик самостоятельно открывает новые понятия путем неорганизованных наблюдений или в ходе планирования и проведения экспериментов в той среде, где он находится. В этом обучении учитель не участвует, и вся тяжесть ложится на ученика. Роль наставника играет среда самого ученика.

Между этими двумя крайностями находится еще одна форма обучения — обучение на примерах. В этом случае инициатива распределяется между учителем и учеником. Учитель предоставляет примеры для изучения, а от ученика требуется, чтобы он пришел к определенным выводам в отношении этих примеров, иными словами,

создал своего рода теорию, объясняющую смысл предъявленных примеров. Учитель может помочь ученику, выбирая хорошие учебные примеры и описывая эти примеры на языке, обеспечивающем составление изящных общих правил. В определенном смысле при обучении на основе примеров используется известное эмпирическое наблюдение, что специалистам в своей области (которые играют роль учителей) проще подготовить хорошие примеры, чем разрабатывать явно сформулированные и полные общие теории. С другой стороны, для ученика задача прийти к какому-то общему выводу на основе примеров может оказаться сложной.

Обучение на примерах называется также *индуктивным обучением*. Индуктивное обучение — это наиболее глубоко исследованный способ обучения в искусственном интеллекте, и исследования, проведенные в этой области, принесли много фундаментальных результатов. На примерах может быть организовано обучение выполнению заданий нескольких типов: с их помощью можно научить программу диагностировать заболевания людей или болезни растений; предсказывать погоду; прогнозировать биологическую активность нового химического соединения; определять способность химических веществ к биологическому разложению; предсказывать механические свойства стали на основе ее химических характеристик; принимать лучшие финансовые решения; управлять динамической системой или повышать эффективность решения задач символического интегрирования.

Методы машинного обучения были применены для решения всех этих конкретных задач и многих других проблем. Разработаны удобные методы, которые могут эффективно использоваться в сложных приложениях. Одна из прикладных областей относится к приобретению знаний для экспертных систем. Накопление знаний происходит автоматически на основе примеров, что позволяет устранить узкое место в работе экспертных систем, связанное с приобретением знаний. Еще одним способом использования методов машинного обучения является *обнаружение скрытых закономерностей в базах данных* (Knowledge Discovery in Databases — KDD), называемое также *интеллектуальным анализом данных*. Данные в базе данных используются в качестве примеров для индуктивного обучения, что позволяет открывать интересные закономерности в больших объемах данных. Например, в результате интеллектуального анализа данных можно обнаружить, что клиент супермаркета, который покупает спагетти, скорее всего, купит также сыр пармезан, и соответствующим образом организовать торговлю.

В данной главе в основном рассматриваются методы изучения понятий на примерах. Вначале определим проблему изучения понятий на примерах более формально. Затем для иллюстрации основных идей в этой области рассмотрим подробный пример изучения понятий, представленных с помощью семантических сетей. После этого рассмотрим, как происходит логический вывод правил и деревьев решения.

## 18.2. Проблема изучения понятий на примерах

### 18.2.1. Понятия, представленные в виде множеств

Проблема изучения понятий на примерах может быть формализована следующим образом. Предположим, что  $U$  — универсальное множество объектов; иными словами, все объекты, которые могут встретиться ученику. В принципе множество  $U$  может иметь неограниченные размеры. Понятие  $S$  можно формально представить как подмножество объектов множества  $U$ . Изучить понятие  $S$  означает научиться распознавать объекты в подмножестве  $S$ . Другими словами, после изучения  $S$  система способна для любого объекта  $X$  в множестве  $U$  определить, принадлежит ли  $X$  к  $S$ .

Это определение понятия является достаточно общим для того, чтобы на его основе можно было формализовать такие разнообразные понятия, как арка, некоторая болезнь, арифметическая операция умножения или определение ядовитого гриба, например, как показано ниже.

- **Определение ядовитого гриба.** В множестве  $U$  грибов понятие “ядовитый” охватывает подмножество всех ядовитых грибов.
- **Понятие арки в мире блоков.** Универсальное множество  $U$  представляет собой множество всех конструкций, построенных из блоков в мире блоков. Арка — это подмножество множества  $U$ , содержащее все конструкции, подобные арке, и ничто иное.
- **Понятие умножения.** Универсальным множеством  $U$  является множество всех троек чисел, а  $Mult$  — это множество всех троек чисел  $(a, b, c)$ , таких, что  $a * b = c$ . Более формальное определение этого множества приведено ниже.  

$$Mult = \{(a, b, c) \mid a * b = c\}$$
- **Понятие некоторой болезни  $D$ .** Если  $U$  — множество всех возможных описаний симптомов пациентов в терминах некоторого выборочного набора симптомов, то  $D$  — множество всех таких описаний симптомов, которые характеризуют пациентов, страдающих от рассматриваемой болезни.

## 18.2.2. Примеры и гипотезы

Чтобы ввести определенную терминологию, рассмотрим следующую гипотетическую проблему изучения того, является ли гриб съедобным или ядовитым. Например, предположим, что было собрано определенное количество грибов и для каждого из них получено мнение эксперта. Допустим, что каждый гриб можно достаточно полно описать по его высоте и ширине (необходимо сделать оговорку, что это лишь пример и такое предположение просто нереально!). В этом случае применяется формулировка, что каждый из рассматриваемых примеров объектов имеет два атрибута: высоту и ширину (в сантиметрах). На сей раз оба атрибута являются числовыми. Кроме того, для каждого примера гриба указан также его класс — “ядовитый” или “съедобный”. С точки зрения изучения понятия “съедобный” эти два значения класса могут быть соответствующим образом обозначены с помощью знаков “+” (съедобный) и “-” (несъедобный). Согласно этому, указанные съедобные грибы являются положительными примерами, а ядовитые — отрицательными примерами понятия “съедобный”.

На рис. 18.1 показаны данные, применяемые для обучения. Таким образом, научиться распознавать грибы означает приобрести способность отнести новый гриб к одному из двух классов, “+” или “-”. Теперь предположим, что нам предъявлен новый гриб, который имеет атрибуты  $W = 3$ ,  $H = 1$ . Является ли он съедобным или ядовитым? Рассматривая примеры, приведенные на рис. 18.1, большинство людей без колебаний отвечают “съедобный”. Безусловно, нет никакой гарантии, что именно этот гриб действительно является съедобным, и такое утверждение для многих становится неожиданным. Поэтому данная классификация все еще относится к области гипотез. Но эта гипотеза выглядит весьма вероятной, поскольку значения атрибутов этого гриба аналогичны атрибутам многих известных съедобных грибов, но отличаются от всех ядовитых грибов. Как правило, основное допущение в машинном обучении состоит в том, что объекты, которые в определенной степени выглядят аналогичными друг другу, принадлежат к одному и тому же классу. В общем, наш мир к нам снисходителен, поскольку в реальной жизни это допущение о принадлежности похожих друг на друга объектов к одному и тому же классу обычно оправдывается. Именно поэтому появляется возможность организовать машинное обучение на примерах. Но остается нерешенным еще один вопрос — как определить, что два объекта аналогичны, а другие два — нет. Каковым является явный или неявный критерий *аналогичности*? Обучающиеся системы в значительной степени отличаются друг от друга именно в этом отношении.

По тем же признакам аналогичности еще один гриб с размерами  $W = 5$  и  $H = 4$ , вполне очевидно, может оказаться ядовитым. Но в отношении гриба с размерами  $W = 2$  и

$N = 2$  решение принять сложнее, и любой вариант его классификации кажется необоснованным и рискованным.

Обычно результатом обучения становится описание понятия, или создание классификатора, позволяющего определять принадлежность новых объектов к конкретному классу. Такой классификатор может быть определен различными способами с использованием разных *формальных представлений*. Для таких формальных представлений есть еще одно название — *языки описания понятий*, или *языки гипотез*. Они именуются языками гипотез по той причине, что позволяют описать гипотезы ученика в отношении целевого понятия, сформулированные на основе обучающих данных. Обычно ученик не совсем уверен в том, что гипотеза, полученная на основе этих данных, действительно соответствует целевому понятию.

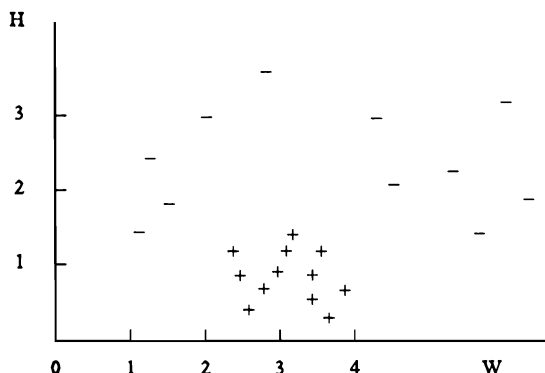


Рис. 18.1. Примеры для обучения способности различать грибы. Атрибутами являются размеры гриба —  $W$  (ширина) и  $N$  (высота). Знаками "плюс" обозначены примеры съедобных грибов, а знаками "минус" — ядовитых

Ниже приведены некоторые возможные гипотезы, которые могут быть выведены на основании данных о грибах.

Гипотеза 1:

если  $2 < W$  и  $W < 4$  и  $N < 2$ , то "съедобный", иначе "ядовитый"

Гипотеза 2:

если  $N > W$ , то "ядовитый", иначе если  $N > 6 - W$ , то "ядовитый", иначе "съедобный"

Гипотеза 3:

если  $N < 3 - (W - 3)^2$ , то "съедобный", иначе "ядовитый"

Эти гипотезы показаны графически на рис. 18.2. Все они сформулированы в виде правил вывода. Еще одним языком гипотез, который широко применяется в области машинного обучения проблематики искусственного интеллекта, являются деревья решения. Гипотеза 1 представлена в виде дерева решения на рис. 18.3.

Все эти три гипотезы являются *совместимыми с данными* — они позволяют отнести все учебные объекты к тому же классу, который указан в этих примерах. Но при классификации новых объектов между этими гипотезами возникают различия. Например, согласно гипотезе 1 гриб с размерами  $W = 3$  и  $N = 2.5$  относится к ядовитым, а согласно гипотезам 2 и 3 этот гриб является съедобным. С точки зрения понятия "съедобный" гипотезу 1 можно назвать наиболее конкретной из этих трех гипотез, а гипотезы 2 и 3 с этой точки зрения считаются более общими, чем гипотеза 1. Множество грибов, являющихся ядовитыми согласно гипотезе 1, представляет собой подмножество тех грибов, которые соответствуют гипотезе 2 или 3. С другой стороны, гипотеза 2 не является ни более общей, ни более конкретной, чем гипотеза 3.

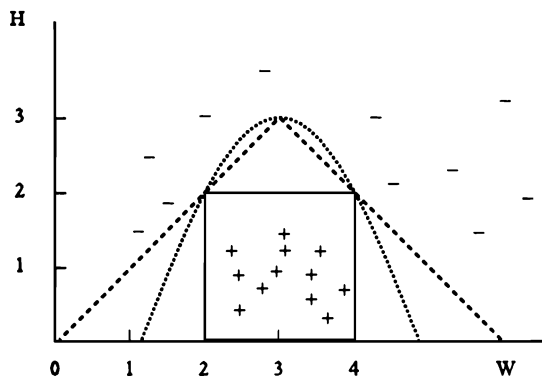


Рис. 18.2. Три гипотезы о съедобных грибах; область действия гипотезы 1 обозначена сплошной линией, гипотезы 2 — штриховой, гипотезы 3 — пунктирной

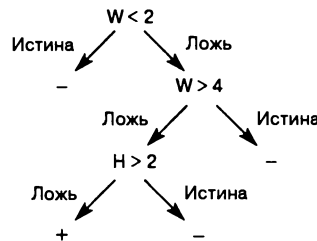


Рис. 18.3. Гипотеза 1, представленная в виде дерева решения. Внутренние узлы дерева обозначены атрибутами, листья — названиями классов, а ветви соответствуют значениям атрибутов. Например, самая левая ветвь соответствует  $W < 2$ , а самый левый лист указывает, что соответствующий ему гриб является ядовитым (имеет класс “-”). Объект соответствует определенному листу, если он удовлетворяет всем условиям на пути от корня к данному листу

### 18.2.3. Языки описания объектов и понятий

Для обучения любого рода требуются язык описания объектов и язык описания понятий (язык гипотез). Как правило, описания подразделяются на два основных типа.

- Реляционные описания.
- Описания на основе атрибутов и значений.

В реляционном описании объект представлен с помощью его компонентов и отношений между ними. Например, реляционное описание арки заключается в следующем: арка — это конструкция, состоящая из трех компонентов (двух стоек и перекладины); каждый компонент представляет собой блок; обе стойки поддерживают перекладину; стойки не соприкасаются. Такое описание является реляционным, поскольку в нем рассматриваются отношения между компонентами. С другой стороны, в описании на основе атрибутов и значений объект рассматривается с точки зрения его глобальных характеристик. Такое описание представляет собой вектор значений атрибутов. Например, в описании на основе атрибутов и значений могут использоваться такие атрибуты, как ширина, высота и цвет. Поэтому описание некоторой ар-



ки на основе атрибутов и значений может состоять в следующем: длина — 9 м, высота — 7 м, цвет — желтый.

Описания на основе атрибутов и значений представляют собой частный случай реляционных описаний. Атрибуты могут быть формально представлены как компоненты некоторого объекта. Поэтому обычно описание на основе атрибутов и значений можно легко перевести на язык реляционных описаний. С другой стороны, задача преобразования реляционного описания в описание на основе атрибутов и значений часто является сложной, а иногда неосуществимой.

Безусловно, языки описания, которые могут использоваться в машинном обучении, аналогичны тем, которые служат для представления знаний в целом. Ниже перечислены некоторые формальные средства, часто используемые в машинном обучении.

- Векторы атрибутов и значений, применяемые для представления объектов.
- Правила вывода, применяемые для представления понятий.
- Деревья решения, применяемые для представления понятий.
- Семантические сети.
- Средства представления логики предикатов различных типов (например, язык Prolog).

Семантические сети, правила вывода и деревья решения рассматриваются ниже в этой главе. Использование логики предикатов в машинном обучении называется *индуктивным логическим программированием* (Inductive Logic Programming — ILP). Эта тема рассматривается в главе 19.

## 18.2.4. Точность гипотез

Проблема обучения на примерах обычно формулируется следующим образом. Имеется некоторое целевое понятие  $C$ , которое необходимо освоить. Кроме того, существует некоторый язык гипотез  $L$ , на котором могут формулироваться гипотезы, касающиеся  $C$ . Определение понятия  $C$  не дано, и единственным источником информации для освоения понятия  $C$  является множество классифицированных примеров. Обычно примеры заданы в виде пар (Object, Class), где Class указывает, к какому понятию относится объект Object. Целью обучения является составление на языке гипотез  $L$  формулы  $H$ , в максимально возможной степени соответствующей целевому понятию  $C$ . Но как узнать, насколько хорошо формула  $H$  соответствует понятию  $C$ ? Единственный способ оценить, насколько полно  $H$  соответствует  $C$ , состоит в использовании множества примеров  $S$ . Оценка качества  $H$  осуществляется на множестве примеров  $S$ . Если  $H$  обычно классифицирует примеры в  $S$  правильно (т.е. относит их к тем же классам, какие указаны в этих примерах), то можно надеяться, что  $H$  позволит столь же правильно классифицировать другие, новые объекты. Поэтому обоснованный подход состоит в том, что среди возможных гипотез должна быть выбрана такая гипотеза, которая позволяет снова отнести все примеры объектов к тому же классу, который указан в множестве  $S$ . Такая гипотеза называется *совместимой с данными*. Совместимая гипотеза характеризуется 100%-ной точностью классификации учебных данных. Но, безусловно, нас больше интересует точность предсказания с помощью некоторой гипотезы: "Насколько точно эта гипотеза предсказывает класс новых объектов, не заданных в  $S$ ?" *Точность предсказания* — это вероятность правильной классификации объектов, выбранных случайным образом в области определения задачи обучения. Хотя на первый взгляд это может показаться удивительным, но иногда обнаруживается, что гипотезы, которые достигают наивысшей точности при распознавании учебных данных  $S$ , не показывают столь же впечатляющих достижений при распознавании новых данных, не принадлежащих к  $S$ . Это наблюдение особенно характерно для обучения по зашумленным данным, когда учебные данные содержат ошибки. Данная тема рассматривается в разделе 18.6.

Чаще всего критерием успеха в индуктивном машинном обучении является точность предсказаний на основе логически выведенных гипотез. Но есть и другие кри-

терии успеха; наиболее широко известным из них является критерий постижимости, или “понятности”, выведенных гипотез. Он определяет, насколько осмысленными являются выведенные гипотезы для человека-эксперта. Эта тема рассматривается более подробно в разделе 18.7.

## 18.3. Подробный пример формирования реляционных описаний в результате обучения

Обучение способности распознавать в мире блоков конструкции и, в частности, арки было впервые рассмотрено как проблемная область обучения Уинстоном [169] в его ранней работе по машинному обучению. Эта область рассматривается в данной главе для иллюстрации некоторых важных механизмов, касающихся обучения. Применяемая здесь трактовка в основном (хотя и не полностью) опирается на программу Уинстона, называемую ARCHES.

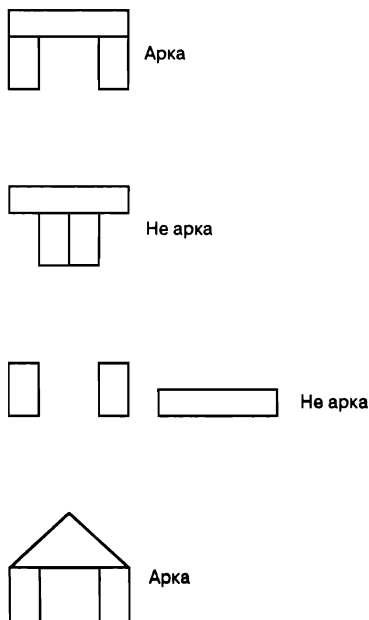


*Рис. 18.4. Последовательность примеров и контрпримеров для изучения понятия арки*

Программу ARCHES можно использовать для изучения понятия арки на примерах (рис. 18.4). Приведенные здесь примеры обрабатываются последовательно, и ученик постепенно обновляет текущую гипотезу, касающуюся понятия арки. В случае, показанном на рис. 18.4, гипотеза (т.е. окончательное понимание учеником, что такое арка), сформулированная после обработки учеником всех четырех примеров, может неформально выглядеть примерно так, как описано ниже.

1. Арка состоит из трех частей; предположим, что они обозначены как `post1` (стойка 1), `post2` (стойка 2) и `lintel` (перекладина).
2. Стойки `post1` и `post2` имеют прямоугольную форму, а перекладина `lintel` может иметь более сложную форму, например многоугольную (этот вывод может быть сделан на основании примеров 1 и 4 на рис. 18.4).

3. Стойки `post1` и `post2` не должны соприкасаться (к этому выводу можно прийти на основании отрицательного примера 2).
4. Стойки `post1` и `post2` должны поддерживать перекладину `lintel` (это следует из отрицательного примера 3).

В целом при изучении понятия путем последовательной обработки учебных примеров процесс обучения проходит через последовательность гипотез об изучаемом понятии,  $H_1$ ,  $H_2$  и т.д. Каждая гипотеза в этой последовательности представляет собой аппроксимацию целевого понятия и результат изучения предыдущих примеров. После обработки следующего примера текущая гипотеза обновляется, что приводит к получению следующей гипотезы. Этот процесс может быть сформулирован в виде описанного ниже алгоритма.

Для изучения понятия  $C$  с использованием заданной последовательности примеров  $E_1, E_2, \dots, E_n$  (где  $E_1$  должен быть положительным примером  $C$ ) необходимо выполнить перечисленные ниже действия.

1. Принять  $E_1$  в качестве начальной гипотезы  $H_1$  о понятии  $C$ .
2. Обработать все оставшиеся примеры, при этом для каждого примера  $E_i$  ( $i = 2, 3, \dots$ ) выполнить следующее.
  - 2.1. Сопоставить текущую гипотезу  $H_{i-1}$  с  $E_i$ ; допустим, что результатом сопоставления является некоторое описание  $D$  различий между  $H_{i-1}$  и  $E_i$ .
  - 2.2. Доработать гипотезу  $H_{i-1}$  с учетом описания  $D$ , а также того, является ли  $E_i$  положительным или отрицательным примером  $C$ ; результатом этого становится уточненная гипотеза  $H_i$ , касающаяся  $C$ .

Окончательным результатом этой процедуры является гипотеза  $H_n$ , которая выражает степень освоения системой понятия  $C$ , изученного на указанных примерах. В фактической реализации шаги 2.1 и 2.2 требуют определенного усовершенствования. Они являются сложными и в разных обучающихся системах реализованы по-разному. Для иллюстрации некоторых идей и возможных сложностей рассмотрим более подробно случай изучения понятия арки на примерах арок, приведенных на рис. 18.4.

Прежде всего необходимо конкретизировать это представление. В программе ARCHES для представления и учебных примеров, и описаний понятий используются семантические сети. Примеры таких семантических сетей приведены на рис. 18.5. Они представляют собой графы, узлы которых соответствуют объектам, а связи обозначают отношения между объектами.

Первый пример, представленный с помощью семантической сети, становится текущей гипотезой о том, что собой представляет арка (см. граф  $H_1$  на рис. 18.5).

Второй пример (граф  $E_2$  на рис. 18.5) представляет собой отрицательный пример арки. Сопоставить  $E_2$  с  $H_1$  несложно. Поскольку обе сети очень похожи, можно легко установить соответствие между узлами и связями в  $H_1$  и  $E_2$ . Результат сопоставления показывает различие  $D$  между  $H_1$  и  $E_2$ . Различие состоит в том, что в  $E_2$  есть еще одно отношение, `touch`. Поскольку в этом состоит единственное различие, система приходит к выводу, что именно по этой причине  $E_2$  не является аркой. После этого система обновляет текущую гипотезу  $H_1$ , применяя следующий общий эвристический принцип обучения:

если

пример – отрицательный и  
содержит отношение  $R$ , который не принадлежит к текущей гипотезе  $H$ ,

то

запретить применение отношения  $R$  в гипотезе  $H$  (ввести в гипотезу  $H$  отношение `must_not_R`).

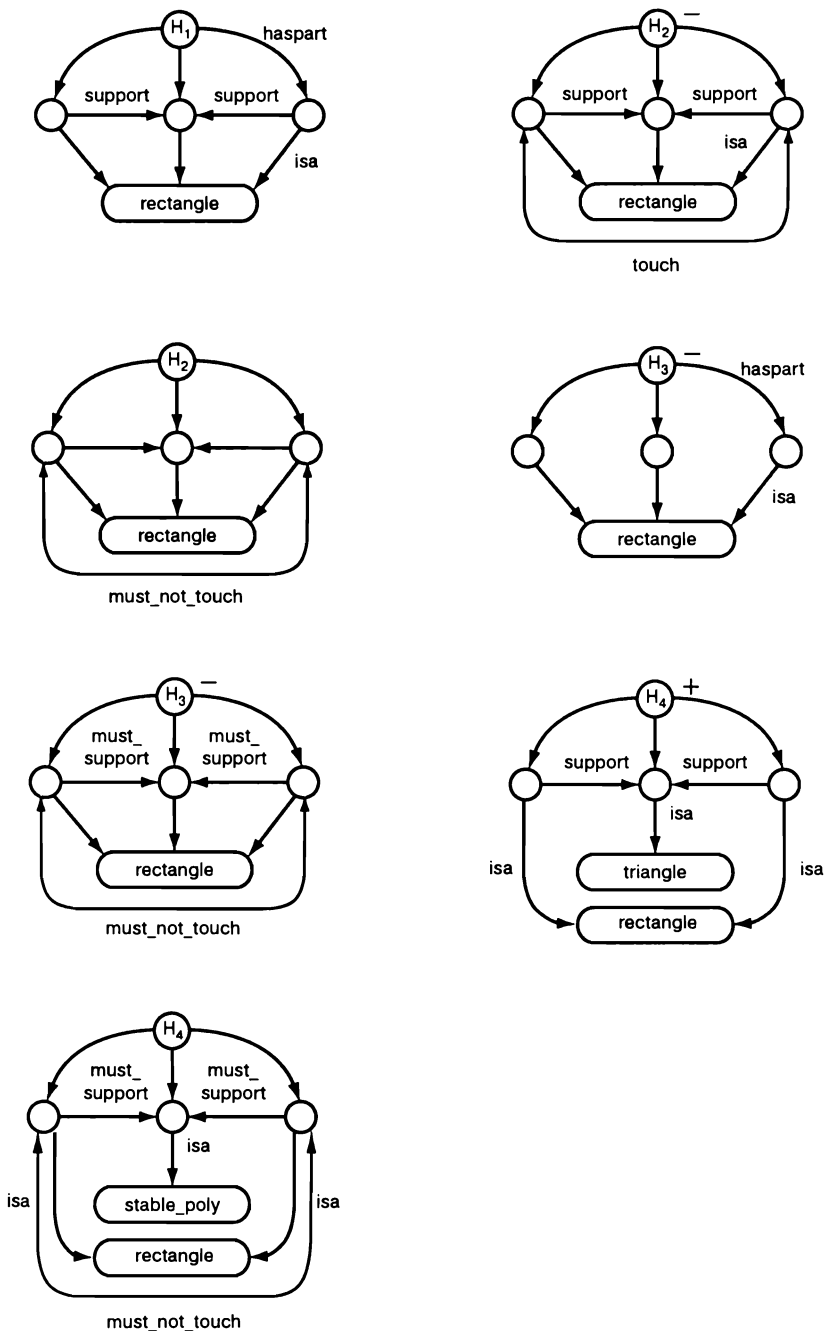


Рис. 18.5. Развитие гипотез об арке; на каждом этапе текущая гипотеза  $H$  сравнивается со следующим примером  $E_{i+1}$  и вырабатывается очередная, уточненная гипотеза  $H_{i+1}$

Результатом применения этого правила к гипотезе  $H_1$  становится новая гипотеза  $H_2$  (см. рис. 18.5). Обратите внимание на то, что в новой гипотезе имеется еще одна связь, `must_not_touch` (не должны соприкасаться), которая налагает дополнительное ограничение на конструкцию, рассматриваемую как арка. Поэтому можно утверждать, что новая гипотеза  $H_2$  является более конкретной, чем  $H_1$ .

Следующий отрицательный пример, приведенный на рис. 18.4, представлен семантической сетью  $E_3$  на рис. 18.5. Его сопоставление с текущей гипотезой  $H_2$  обнаруживает два различия: две связи `support`, присутствующие в  $H_2$ , отсутствуют в  $E_3$ . После этого ученик должен выбрать наиболее приемлемое среди следующих трех возможных объяснений.

1. Пример  $E_3$  — не арка, поскольку отсутствует левая стойка.
2. Пример  $E_3$  — не арка, поскольку отсутствует правая стойка.
3. Пример  $E_3$  — не арка, поскольку отсутствуют обе стойки.

В соответствии с этим ученик должен выбрать один из трех возможных способов обновления текущей гипотезы. Предположим, что ученик скорее склонен действовать решительно, чем осторожно, поэтому ему больше нравится объяснение 3. Таким образом, ученик принимает предположение, что нужны обе связи `support`, и поэтому преобразует две связи `support` гипотезы  $H_2$  в связи `must_support` новой гипотезы  $H_3$  (см. рис. 18.5). Эту ситуацию отсутствия связей можно обработать с помощью следующего правила типа “условие–действие”, которое представляет собой еще одно общее эвристическое правило, касающееся обучения:

если

пример — отрицательный и

не содержит отношения  $R$ , который присутствует в текущей гипотезе  $H$ ,  
то

следует включить отношение  $R$  в новую гипотезу (добавить в гипотезу  $H$  отношение `must_R`).

Еще раз отметим, что в результате обработки отрицательного примера текущая гипотеза стала еще более конкретной, поскольку были введены дополнительные необходимые условия — две связи `must_support`. Обратите также внимание на то, что ученик мог бы выбрать более осмотрительный способ действий — ввести только одну связь `must_support` вместо двух. Поэтому очевидно, что рассматриваемый подход позволяет моделировать индивидуальный стиль обучения с помощью множества правил “условие–действие”, применяемых учеником для обновления текущей гипотезы. Изменяя набор этих правил, можно менять стиль обучения, переходя от осторожного и осмотрительного к радикальному и решительному.

Последний пример в этой обучающей последовательности,  $E_4$ , снова является положительным. Сопоставление соответствующих семантических сетей,  $E_4$  и  $H_3$ , позволяет обнаружить различие: верхняя часть в  $E_4$  является треугольной, а в  $H_3$  — прямоугольной. Теперь ученик может перенаправить соответствующую связь `isa` в данной гипотезе с прямоугольника на новый класс объектов — `rectangle_or_triangle` (прямоугольник или треугольник). Но альтернативная (и более часто применяемая) реакция в обучающейся программе основана на заранее определенной иерархии понятий. Предположим, что ученик имеет возможность воспользоваться классификацией понятий, подобной приведенной на рис. 18.6, в качестве априорных знаний о конкретной проблемной области. Обнаружив, что согласно этой классификации и прямоугольник, и треугольник относятся к типу `stable_poly` (устойчивый многоугольник), ученик может обновить текущую гипотезу, чтобы получить гипотезу  $H_4$  (см. рис. 18.5).

Обратите внимание на то, что на этот раз обработка положительного примера привела к созданию более общей новой гипотезы (о том, что вместо прямоугольного блока можно применять устойчивый многоугольный блок). В этом случае принято считать, что текущая гипотеза была обобщена. Теперь новая гипотеза позволяет использовать трапециевидный блок в качестве верхней части арки, хотя ученику еще не был предъявлен пример арки с трапециевидной перекладиной. Если после этого

системе будет показана конструкция, приведенная на рис. 18.7, и задан вопрос, к какому классу она относится, то система назовет ее аркой, поскольку эта конструкция полностью соответствует окончательному представлению системы об арке, иными словами, гипотезе  $H_4$ .

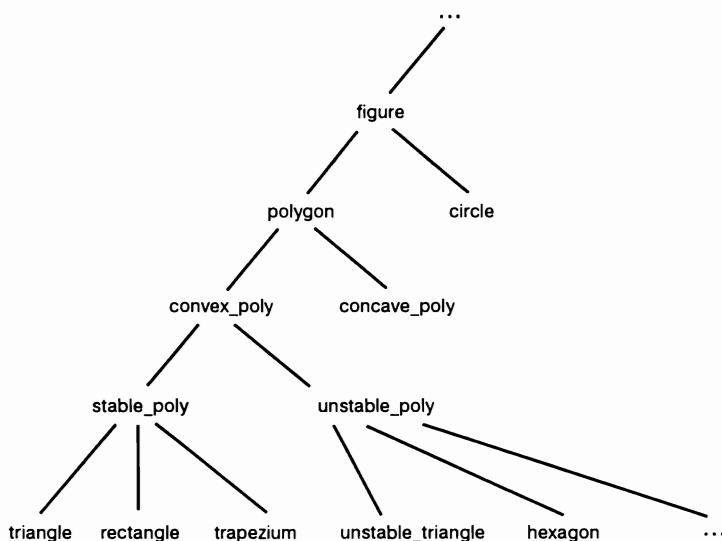


Рис. 18.6. Иерархия понятий

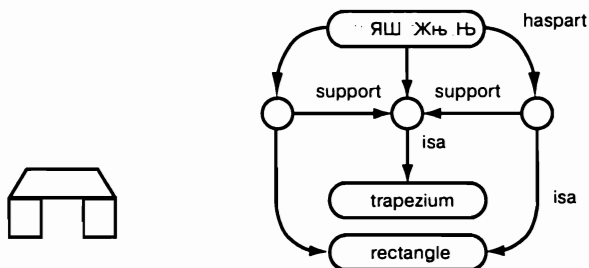


Рис. 18.7. Слева показан новый объект, справа — его представление. Этот объект соответствует определению понятия арки  $H_4$ , показанному на рис. 18.5, при условии, что используется иерархия понятий, показанная на рис. 18.6

Ниже приведены некоторые важные выводы, которые можно продемонстрировать на предыдущем примере.

- Процедура сопоставления объекта с гипотезой зависит от обучающейся системы. Процесс сопоставления часто является сложным и может требовать больших вычислительных ресурсов. Например, если в качестве языка гипотез используются семантические сети, то может потребоваться проверить все возможные соответствия между рассматриваемыми узлами в двух сопоставляемых сетях для определения наилучшего совпадения.
- Если имеются две гипотезы, то одна из них может быть более общей или более конкретной, чем другая, или они могут быть несравнимыми по признаку общности или конкретности.

- Модификация гипотезы в процессе обучения приводит к изменению в лучшую сторону одного из следующих признаков: общности гипотезы, что позволяет сопоставить гипотезу с указанным положительным примером, или конкретности гипотезы, что способствует предотвращению возможности сопоставления гипотезы с отрицательным примером.
- Принципы модификации понятий для определенной обучающейся системы могут быть представлены в виде правил “условие–действие”. Такие правила позволяют моделировать “поведение” обучающейся системы, придавая ей различные стили поведения, начиная от предельно осторожного и заканчивая крайне решительным.

В предыдущее издание этой книги [12] была включена реализация на языке Prolog программы обучения с помощью реляционных описаний, организованная по принципам, которые рассматриваются в данном разделе. В настоящее издание указанная реализация не включена, хотя хорошей иллюстрацией к фундаментальным понятиям машинного обучения может служить программа ARCHES. Дело в том, что некоторые алгоритмы обучения показали себя как намного более эффективные инструментальные средства обучения для практических приложений. К ним относятся средства обучения с помощью правил вывода (раздел 18.4) и деревьев решения (раздел 18.5). Мы вернемся к теме обучения с помощью реляционных описаний в следующей главе, где рассматривается инфраструктура индуктивного логического программирования.

## 18.4. Обучение с помощью простых правил вывода

### 18.4.1. Описание объектов и понятий с использованием атрибутов

В этом разделе рассматривается способ обучения, при котором примеры и гипотезы определяются в терминах множеств атрибутов. В принципе атрибуты могут относиться к различным типам, в зависимости от их возможных значений. Поэтому любой атрибут может быть числовым или нечисловым. Кроме того, если атрибут является нечисловым, то множество его значений рассматривается как упорядоченное или неупорядоченное. Ограничимся нечисловыми атрибутами с неупорядоченными множествами значений. Такое множество обычно невелико и содержит лишь несколько значений.

Для описания любого объекта достаточно указать конкретные значения атрибутов этого объекта. Таким образом, подобное описание представляет собой вектор значений атрибутов.

На рис. 18.8 показаны некоторые объекты, применяемые в качестве иллюстрации в данном разделе. Эти объекты относятся к пяти классам: гайка, винт, ключ, карандаш, ножницы. Предположим, что эти объекты предъявляются системе машинного зрения. Их силуэты фиксируются видеокамерой, а затем обрабатываются программой визуального распознавания объектов. После получения данных эта программа определяет значения некоторых атрибутов каждого объекта по изображению, полученному видеокамерой. В данном случае рассматриваются такие атрибуты, как размер, форма и количество отверстий в объекте. Предположим, что возможные значения этих атрибутов таковы:

|                             |                                                   |
|-----------------------------|---------------------------------------------------|
| size: small, large          | % размер: малый, большой                          |
| shape: long, compact, other | % форма: продолговатая, компактная, другая        |
| holes: none, 1, 2, 3, many  | % количество отверстий: ни одного, 1, 2, 3, много |

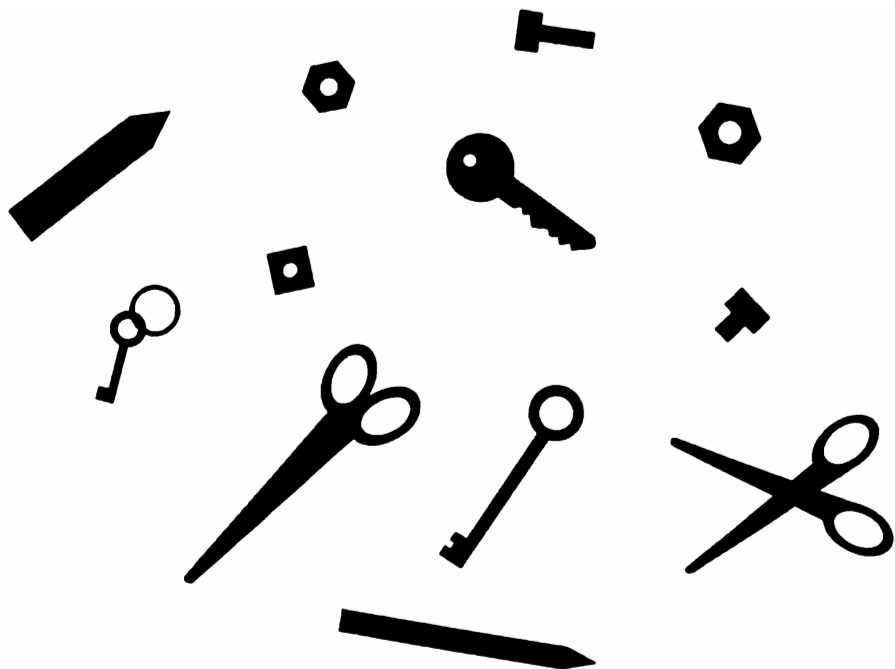


*Рис. 18.8. Изображения некоторых объектов, полученные с помощью видеокамеры*

Допустим, что система машинного зрения определила значения трех атрибутов для каждого объекта. В листинге 18.1 приведены определения атрибутов, а показанные здесь примеры представлены как множество предложений Prolog в следующей форме:

```
example(Class, [Attribute1 = Val1, Attribute2 = Val2, ...]).
```

#### **Листинг 18.1. Определения атрибутов и примеры для обучения распознаванию объектов по их силуэтам (показанным на рис. 18.8)**

```
attribute(size, [small, large]).
attribute(shape, [long, compact, other]).
attribute(holes, [none, 1, 2, 3, many]).

example(nut, [size = small, shape = compact, holes = 1]).
example(screw, [size = small, shape = long, holes = none]).
example(key, [size = small, shape = long, holes = 1]).
example(nut, [size = small, shape = compact, holes = 1]).
example(key, [size = large, shape = long, holes = 1]).
example(screw, [size = small, shape = compact, holes = none]).
example(nut, [size = small, shape = compact, holes = 1]).
example(pen, [size = large, shape = long, holes = none]).
example(scissors, [size = large, shape = long, holes = 2]).
example(pen, [size = large, shape = long, holes = none]).
example(scissors, [size = large, shape = other, holes = 2]).
example(key, [size = small, shape = other, holes = 2]).
```

Теперь предположим, что эти примеры переданы в обучающуюся программу, которая должна научиться распознавать предметы этих пяти классов. Результатом обу-



чения должно быть описание классов в форме правил, которые могут использоваться для классификации новых объектов. В качестве примеров формата этих правил ниже приведены возможные правила для классов nut (гайка) и key (ключ).

```
nut <== [[size = small, holes = 1]]
key <== [[shape = long, holes = 1], [shape = other, holes = 2]]
```

Эти правила можно описать словами, как показано ниже.

Объект является гайкой, если  
он имеет малые размеры и  
одно отверстие.

Объект является ключом, если  
он имеет продолговатую форму и  
одно отверстие  
или  
он имеет другую форму и  
два отверстия.

Правила имеют такую общую форму:

```
Class <== [Conj1, Conj2, ...]
```

Здесь Conj1, Conj2 и т.д. представляют собой списки значений атрибутов в такой форме:

```
[Att1 = Val1, Att2 = Val2, ...]
```

Описание класса [ Conj1, Conj2, ... ] интерпретируется следующим образом.

1. Объект соответствует описанию, если он согласуется, по меньшей мере, с одним из списков значений атрибутов Conj1, Conj2 и т.д.
2. Объект согласуется со списком значений атрибутов Conj, если все значения атрибутов в этом списке являются такими же, как и у самого объекта.

Например, объект, описанный с помощью выражения

```
[size = small, shape = long, holes = 1]
```

соответствует правилу для ключа key, поскольку его атрибуты согласуются с первым списком значений атрибутов в этом правиле. Итак, значения атрибутов в списке Conj связаны между собой конъюнктивной логической зависимостью; это означает, что ни один из них не может отличаться от значения такого же атрибута объекта. С другой стороны, списки Conj1, Conj2 и т.д. связаны между собой дизъюнктивной логической зависимостью; это означает, что значения атрибутов объекта должны удовлетворять хотя бы одному из этих списков.

Процедура сопоставления объекта с описанием понятия может быть сформулирована на языке Prolog следующим образом:

```
match(Object, Description) :-
 member(Conjunction, Description),
 satisfy(Object, Conjunction).

satisfy(Object, Conjunction) :-
 not (
 member(Att = Val, Conjunction), % Значение в описании
 member(Att = ValX, Object), % и значение в объекте
 ValX \== Val). % являются разными
```

Обратите внимание на то, что это определение допускает использование частично заданных объектов, для которых значения некоторых атрибутов могут быть не указаны, иными словами, не включены в список атрибутов и значений. В таком случае предполагается, что незаданное значение удовлетворяет требованиям, которые определены с помощью переменной Conjunction.

## 18.4.2. Логический вывод правил на основании примеров

Теперь рассмотрим, каким образом могут формулироваться правила на основании множества примеров. В отличие от задачи распознавания арок, приведенной в предыдущем разделе, описание класса не формируется путем последовательной обработки отдельных примеров. Вместо этого все примеры обрабатываются одновременно. Поэтому такой процесс называют также *пакетным обучением* (batch learning) в отличие от постепенного, *инкрементного обучения* (incremental learning).

В данном случае основное требование к процедуре обучения состоит в том, чтобы сформированное описание класса точно соответствовало примерам, относящимся к этому классу. Иными словами, это описание должно соответствовать всем положительным примерам данного класса, но ни одному отрицательному примеру.

Если объект соответствует описанию, считается, что описание *охватывает* данный объект. Поэтому необходимо сформировать такое описание для определенного класса, которое охватывает все экземпляры этого класса, но не охватывает ни одного постороннего экземпляра. Такое описание принято называть *полным* и *достоверным*: оно является полным, поскольку охватывает все положительные примеры, и достоверным, поскольку не охватывает ни одного отрицательного примера.

Для формирования непротиворечивой гипотезы в виде множества правил вывода широко используется *алгоритм охвата* (covering algorithm), программа которого приведена в листинге 18.2. Он носит такое название, поскольку постепенно охватывает все положительные примеры изучаемого понятия. Алгоритм охвата начинает свою работу с пустого множества правил, затем обеспечивает итерационный логический вывод одного правила за другим. Ни одно правило не может охватывать какой-либо отрицательный пример, но должно распространяться на некоторые положительные примеры. После логического вывода нового правила это правило добавляется к формулировке гипотезы и из множества примеров удаляются охваченные им положительные примеры. На основании этого нового сокращенного множества примеров осуществляется логический вывод следующего правила, и такая работа продолжается до тех пор, пока не будут охвачены все положительные примеры.

### Листинг 18.2. Алгоритм охвата; процедура InduceOneRule (E) вырабатывает правило, которое охватывает некоторые положительные примеры и ни одного отрицательного

Чтобы логическим путем вывести список правил RULELIST для множества S классифицированных примеров, выполнить следующее:

```
RULELIST := empty;
E:=S;
while E содержит положительные примеры do
begin
 RULE := InduceOneRule(E);
 Добавить правило RULE к списку правил RULELIST;
 Удалить из множества E все примеры, охваченные правилом RULE
end
```

Алгоритм охвата (см. листинг 18.2) допускает введение только непротиворечивых правил. Правила, полученные с помощью логического вывода, должны охватывать все положительные примеры и ни одного отрицательного. На практике используются также менее строгие варианты этого алгоритма охвата, особенно если для обучения применяются зашумленные данные. При использовании таких вариантов работа алгоритма может оканчиваться до того, как будут охвачены все положительные примеры. Кроме того, может быть предусмотрена возможность охватывать в процедуре InduceOneRule некоторые отрицательные примеры, кроме положительных, при условии, что охваченные положительные примеры в достаточной степени превышают по количеству отрицательные.

Реализация этого алгоритма охвата приведена в листинге 18.3. Процедура learn( Examples, Class, Description)

этой программы формирует непротиворечивое описание **Description** для класса **Class** и примеров **Examples**. Ее работу можно описать примерно так, как показано ниже.

Для охвата всех примеров класса **Class**, приведенных с помощью списка примеров **Examples**, необходимо выполнить следующее: если ни один из примеров списка **Examples** не относится к классу **Class**, то **Description** = [] (охвачены все положительные примеры); в противном случае **Description** = [Conj | Conjs], где **Conj** и **Conjs** формируются таким образом.

1. Собрать список **Conj** значений атрибутов, который охватывает хотя бы один положительный пример класса **Class** и не охватывает ни одного примера из любого другого класса.
2. Удалить из списка **Examples** все примеры, охваченные списком **Conj**, затем распространить описание **Conjs** на все оставшиеся и не охваченные объекты.

### Листинг 18.3. Программа логического вывода правил вывода

```
% Обучение на основе простых правил вывода

:- op(300, xfx, <==).

% learn(Class): собрать учебные примеры в список, сформировать и вывести
% описание для класса Class, затем внести в базу данных соответствующее
% правило, касающееся класса Class

learn(Class) :-
 bagof(example(ClassX, Obj), example(ClassX, Obj), Examples), % Собрать
 % примеры
 learn(Examples, Class, Description), % Сформировать правило
 nl, write(Class), write(' <== '), nl, % Вывести правило
 writelist(Description),
 assert(Class <== Description). % Внести правило в базу данных

% learn(Examples, Class, Description):
% описание Description охватывает точно все примеры класса Class
% в списке Examples

learn(Examples, Class, []) :-
 not member(example(Class, _), Examples). % Нет примеров, которые нужно
 % было бы охватить описанием

learn(Examples, Class, [Conj | Conjs]) :-
 learn_conj(Examples, Class, Conj),
 remove(Examples, Conj, RestExamples), % Удалить примеры, которые
 % соответствуют условию Conj
 learn(RestExamples, Class, Conjs). % Охватить описанием
 % остальные примеры

% learn_conj(Examples, Class, Conj):
% Conj - это список значений атрибутов, которому соответствуют некоторые
% примеры класса Class и не соответствует ни один пример какого-то
% иного класса

learn_conj(Examples, Class, []) :-
 not (member(example(ClassX, _), Examples), % Нет примеров какого-то
 ClassX \== Class), !. % иного класса

learn_conj(Examples, Class, [Cond | Conds]) :-
 choose_cond(Examples, Class, Cond), % Выбрать значение атрибута
 filter(Examples, [Cond], Examples1),
 learn_conj(Examples1, Class, Conds).

choose_cond(Examples, Class, AttVal) :-
 findall(AV/Score, score(Examples, Class, AV, Score), AVs),
 best(AVs, AttVal). % Атрибут с наилучшей оценкой
```

```

best([AttVal/_], AttVal).

best([AV0/S0, AV1/S1 | AVSlist], AttVal) :-
 S1 > S0, !, % Атрибут AV1 имеет лучшую оценку, чем AV0
 best([AV1/S1 | AVSlist], AttVal)
 ;
 best([AV0/S0 | AVSlist], AttVal).

% filter(Examples, Condition, Examples1):
% список Examples1 содержит элементы списка Examples, которые соответствуют
% условию Condition

filter(Examples, Cond, Examples1) :-
 findall(example(Class, Obj),
 (member(example(Class, Obj), Examples), satisfy(Obj, Cond)),
 Examples1).

% remove(Examples, Conj, Examples1):
% удаление из списка Examples тех примеров, которые охвачены условием Conj,
% и получение списка Examples1

remove([], _, []).

remove([example(Class, Obj) | Es], Conj, Es1) :-
 satisfy(Obj, Conj), !, % Первый пример соответствует условию Conj
 remove(Es, Conj, Es1). % Удалить его

remove([E | Es], Conj, [E | Es1]) :- % Оставить первый пример в списке
 remove(Es, Conj, Es1).

satisfy(Object, Conj) :-
 not (member(Att = Val, Conj),
 member(Att = ValX, Object),
 ValX \== Val).

score(Examples, Class, AttVal, Score) :-
 candidate(Examples, Class, AttVal), % Подходящее значение атрибута
 filter(Examples, [AttVal], Examples1), % Примеры в списке Examples1
 % соответствуют условию Att = Val
 length(Examples1, N1), % Длина списка
 count_pos(Examples1, Class, NPos1), % Количество положительных примеров
 NPos1 > 0, % По меньшей мере один положительный
 % пример соответствует значению AttVal

 Score is 2 * NPos1 - N1.

candidate(Examples, Class, Att = Val) :-
 attribute(Att, Values), % Атрибут
 member(Val, Values), % Значение
 suitable(Att = Val, Examples, Class).

suitable(AttVal, Examples, Class) :-
 % По меньшей мере один отрицательный пример не должен соответствовать
 % значению AttVal
 member(example(ClassX, ObjX), Examples),
 ClassX \== Class, % Отрицательный пример, который
 not satisfy(ObjX, [AttVal]), !. % не соответствует значению AttVal

% count_pos(Examples, Class, N):
% N - количество положительных примеров класса Class

count_pos([], _, 0).

count_pos([example(ClassX, _) | Examples], Class, N) :-
 count_pos(Examples, Class, N1),
 (ClassX = Class, !, N is N1 + 1; N = N1).

```

```
writelst([]).
```

```
writelst([X | L]) :-
 tab(2), write(X), nl,
 writelst(L).
```

---

Каждый список атрибутов и значений формируется с помощью следующей процедуры:

```
learn_conj(Examples, Class, Conjunction)
```

Список атрибутов и значений Conjunction наращивается постепенно, начиная с пустого списка, к которому последовательно добавляются условия в следующей форме: Attribute = Value

Обратите внимание на то, что в результате список атрибутов и значений становится все более и более конкретным (охватывает все меньше объектов). Список атрибутов и значений является наиболее приемлемым, если он стал настолько конкретным, что охватывает только положительные примеры класса Class.

Процесс формирования подобного конъюнктивного выражения характеризуется высокой комбинаторной сложностью. Каждый раз, когда происходит добавление нового условия, состоящего из атрибута и значения, приходится рассматривать значительное количество возможных вариантов добавления, которое почти столь же велико, как и количество пар атрибутов и значений. При этом невозможно сразу же определить, какой из этих вариантов является наилучшим. Как правило, следует стремиться к тому, чтобы все положительные примеры были охвачены с помощью минимально возможного количества правил, а сами правила были наиболее краткими. Такое обучение может рассматриваться как поиск среди возможных описаний с целью максимального уменьшения длины описания понятия. В связи с высокой комбинаторной сложностью этого поиска обычно приходится прибегать к использованию некоторых эвристических функций. Работа программы, приведенной в листинге 18.3, основана на использовании функции эвристической оценки, которая применяется локально. На каждом этапе к списку добавляется только значение атрибута с наилучшей оценкой, а все остальные варианты сразу же отбрасываются. Поэтому поиск сводится к детерминированной процедуре без какого-либо перебора с возвратами. Такой поиск называют *поглощающим*, или *жадным* (greedy), а также поиском по принципу “подъема к вершине” (hill-climbing). Он рассматривается как поглощающий, поскольку всегда предусматривает выбор наиболее перспективной альтернативы, не оставляя шансов для других вариантов выбора. Но при такой организации поиска существует риск, что не будет обнаружено кратчайшее описание понятия.

Эвристическая оценка является простой и основана на следующем интуитивном наблюдении: полезное условие, заданное в виде атрибута и значения, должно позволять хорошо отличать друг от друга положительные и отрицательные примеры. Поэтому оно должно охватывать максимально возможное количество положительных примеров и минимально возможное количество отрицательных примеров. На рис. 18.9 проиллюстрированы принципы работы такой эвристической функции оценки. Функция, применяемая в рассматриваемой программе, реализована в виде следующей процедуры:

```
score(Examples, Class, AttributeValue, Score)
```

где Score — разница между количеством охваченных положительных и отрицательных примеров.

Программа, приведенная в листинге 18.3, может быть вызвана на выполнение для создания некоторого описания класса для примеров, приведенных в листинге 18.1, с помощью следующего запроса:

```
?- learn(nut), learn(key), learn(scissors).
nut <==
[shape = compact, holes = 1]
key <==
```

```
[shape = other, size = small]
[holes = 1, shape = long]
scissors <==
[holes = 2, size = large]
```

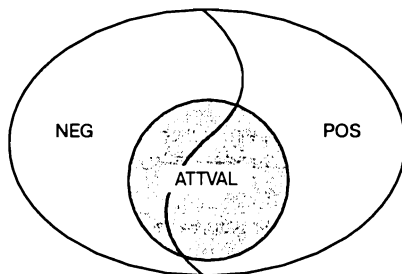


*Рис. 18.9. Эвристическая оценка значения атрибута; где POS — множество положительных примеров изучаемого класса, NEG — множество отрицательных примеров этого класса. Затененная область, ATTVAL, представляет множество объектов, которые удовлетворяют условию, заданному в виде атрибутов и значений. Эвристическая оценка значения атрибута определяется как разница между количеством положительных примеров в области ATTVAL и количеством отрицательных примеров в этой области*

Кроме того, процедура `learn` вносит правила, касающиеся соответствующих классов в программе, в базу данных. Эти правила могут использоваться для классификации новых объектов. Соответствующая процедура распознавания, в которой применяются усвоенные описания, приведена ниже.

```
classify(Object, Class) :-
 Class <== Description, % Правило, касающееся класса Class,
 % полученное путем обучения
 member(Conj, Description), % Конъюнктивное условие
 satisfy(Object, Conj). % Объект Object удовлетворяет условию Conj
```

## 18.5. Логический вывод деревьев решения

### 18.5.1. Основной алгоритм логического вывода дерева

Логический вывод деревьев решения, по-видимому, является наиболее широко применяемым подходом к машинному обучению. В таком случае гипотезы представлены в виде деревьев решения. Процесс логического вывода деревьев является эффективным и удобным для программирования.

На рис. 18.10 показано дерево решения, которое может быть получено путем логического вывода из примеров, приведенных в листинге 18.1 (т.е. объектов, показанных на рис. 18.8). Внутренние узлы этого дерева обозначены именами атрибутов. Листья дерева обозначены именами классов или символом “null”, который указывает, что этому листу не соответствует ни один учебный пример. Ветви в дереве обозначены значениями атрибутов. При классификации объекта по дереву прокладывается путь, начинающийся с корневого узла и оканчивающийся в одном из листьев.

После прохождения каждого внутреннего узла процедура следует по ветви, обозначенной значением атрибута в объекте. Например, объект, описанный следующим образом:  
[ size = small, shape = compact, holes = 1]

согласно этому дереву должен классифицироваться как nut — гайка (после прохождения пути holes = 1, shape = compact). Обратите внимание на то, что в этом случае значение атрибута size = small для классификации объекта не требуется.

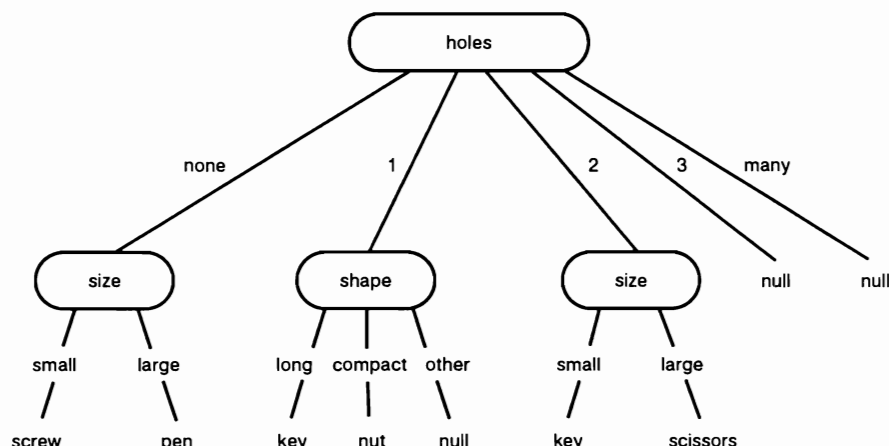


Рис. 18.10. Дерево решения, полученное с помощью логического вывода на основании примеров, приведенных в листинге 18.1 (и изображенных графически на рис. 18.8)

По сравнению с правилами вывода, используемыми для описания классов, которые рассматривались в предыдущем разделе, деревья обладают более ограниченными возможностями представления. Они имеют свои преимущества и недостатки. Представления некоторых понятий с помощью деревьев являются более громоздкими по сравнению с представлениями с помощью правил; хотя в соответствующее дерево решения может быть преобразовано любое описание на основе правил, результирующее дерево часто бывает гораздо сложнее по сравнению с описанием на основе правил. В этом состоит один из очевидных недостатков деревьев.

С другой стороны, тот факт, что деревья как средства представления знаний являются более ограниченными, способствует уменьшению комбинаторной сложности процесса обучения. Это может привести к существенному повышению эффективности обучения. Обучение с помощью деревьев решения является одним из наиболее эффективных форм обучения. Но необходимо отметить, что высокая вычислительная эффективность обучения является лишь одним из многих критериев успешного обучения, как описано ниже в этой главе.

Основной алгоритм логического вывода дерева решения показан на рис. 18.11. Этот алгоритм предназначен для формирования минимально возможного дерева, совместимого с учебными данными. Но поиск среди всех таких деревьев невозможен из-за комбинаторной сложности. Поэтому в процедурах логического вывода деревьев широко применяется эвристический подход, который не гарантирует получение оптимального решения. Алгоритм, показанный на рис. 18.11, является поглощающим, в том смысле, что он всегда выбирает наиболее информативный атрибут и не допускает перебор с возвратами, поэтому не гарантирует, что будет найдено наименьшее дерево. С другой стороны, этот алгоритм работает быстро, и было обнаружено, что он хорошо действует в практических приложениях.

Чтобы сформировать дерево решения  $T$  для обучающего множества  $S$ , необходимо выполнить следующее:

- если все примеры в  $S$  относятся к одному и тому же классу,  $C$ , то результат представляет собой дерево, состоящее из единственного узла, который обозначен как  $C$ ,
- в противном случае
  - выбрать "самый информативный" атрибут,  $A$ , значениями которого являются  $v_1, \dots, v_n$ ;
  - разделить множество  $S$  на подмножества  $S_1, \dots, S_n$  в соответствии со значениями  $A$ ;
  - сформировать (рекурсивно) поддеревья  $T_1, \dots, T_n$  для  $S_1, \dots, S_n$ ;
 конечным результатом становится дерево, корнем которого является  $A$ , поддеревьями -  $T_1, \dots, T_n$ , а связи между  $A$  и поддеревьями обозначены как  $v_1, \dots, v_n$ ; полученное таким образом дерево решения имеет следующий вид:

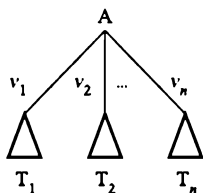


Рис. 18.11. Основной алгоритм логического вывода дерева решения

Даже в своей простейшей реализации этот основной алгоритм требует определенных уточнений, которые описаны ниже.

1. Если множество  $S$  является пустым, то результат представляет собой дерево с одним узлом, обозначенным как "null".
2. После выбора каждого нового атрибута рассматриваются только те атрибуты, которые еще не использовались в верхней части дерева.
3. Если множество  $S$  не пусто, и не все объекты в  $S$  принадлежат к одному и тому же классу, и не осталось атрибутов, доступных для выбора, то результатом становится дерево с одним узлом. Узел такого типа удобно обозначить с помощью списка всех классов, представленных в множестве  $S$  наряду с их относительными частотами в  $S$ . Такое дерево иногда называют *деревом вероятностей классов* (class probability tree), а не деревом решения. В подобном случае для проведения различия между значениями класса некоторых объектов недостаточно иметь множество доступных атрибутов (когда объекты, принадлежащие к разным классам, имеют одинаковые значения атрибутов).
4. Необходимо определить критерий выбора наиболее информативного атрибута. Эта тема рассматривается в следующем разделе.

## 18.5.2. Выбор "наилучшего" атрибута

Значительная часть исследований в области машинного обучения посвящена определению критериев выбора "наилучшего" атрибута. По сути, эти критерии служат для измерения в множестве примеров *количества посторонних включений* (impurity) по отношению к классу. Качественный атрибут должен обеспечивать разбиение примеров на подмножества как можно более безошибочно (охватывая как можно меньше посторонних включений).

В одном из подходов к выбору атрибута используются идеи из области теории информации. Соответствующий критерий может быть разработан следующим образом. Для классификации объекта требуется определенный объем информации. После изучения значения некоторого атрибута объекта для классификации этого объекта иногда достаточно получить еще лишь некоторое небольшое количество информации. Такое дополнительное количество информации будет именоваться *остаточной информацией*. Безусловно, остаточная информация должна быть меньше по сравнению



с первоначальной информацией. Наиболее информативным атрибутом является такой атрибут, который минимизирует остаточную информацию. Количество информации определяется с помощью широко известной формулы энтропии. Для области определения, представленной примерами из обучающего множества  $S$ , среднее количество информации  $I$ , необходимой для классификации объекта, задается с помощью следующей формулы энтропии:

$$I = - \sum_c p(c) \log_2 p(c)$$

где  $c$  обозначает классы, а  $p(c)$  — вероятность того, что некоторый объект из множества  $S$  принадлежит к классу  $c$ . Эта формула полностью соответствует интуитивному представлению о посторонних включениях. Если множество полностью очищено от посторонних включений, то вероятность распознавания одного из его классов равна 1, а для всех других классов равна 0. Для такого множества информация  $I = 0$ . С другой стороны, информация  $I$  является максимальной в том случае, если вероятности всех классов равны.

После применения атрибута  $A$  множество  $S$  разделяется на подмножества в соответствии со значениями  $A$ . Поэтому остаточная информация  $I_{\text{res}}$  равна взвешенной сумме объемов информации для подмножеств:

$$I_{\text{res}}(A) = - \sum_v p(v) \sum_c p(c | v) \log_2 p(c | v)$$

где  $v$  — значения атрибута  $A$ ,  $p(v)$  — вероятность значения  $v$  в множестве  $S$ ,  $p(c | v)$  — условная вероятность класса  $c$ , при условии, что атрибут  $A$  имеет значение  $v$ . Вероятности  $p(v)$  и  $p(c | v)$  обычно аппроксимируются с помощью статистических данных о множестве  $S$ .

В приведенное выше определение критерия с помощью теории информации могут быть внесены дополнительные уточнения. Одним из его недостатков является то, что он способствует выбору атрибутов со многими значениями. Подобные атрибуты, как правило, разбивают множество  $S$  на целый ряд небольших подмножеств. А если эти подмножества очень малы и включают лишь немного примеров, они и так не содержат посторонних включений, независимо от исходной корреляции между атрибутом и классом. Поэтому приведенная выше прямолинейная оценка  $I_{\text{res}}$  придает такому атрибуту слишком высокое значение. Один из способов устранения этого недостатка состоит в использовании *коэффициента приращения информации*. Этот коэффициент позволяет учитывать количество информации  $I(A)$ , необходимое для определения значения любого атрибута  $A$ , как показано ниже.

$$I(A) = - \sum_v p(v) \log_2 (p(v))$$

Обычно значение  $I(A)$  является более высоким для атрибутов с большим количеством значений. Коэффициент приращения информации определяется следующим образом:

$$\text{GainRatio}(A) = \frac{\text{Gain}(A)}{I(A)} = \frac{I - I_{\text{res}}(A)}{I(A)}$$

В процессе обучения должен выбираться такой атрибут, который имеет наибольший коэффициент приращения.

Еще один способ устранения недостатков, связанных с наличием многозначных атрибутов, состоит в использовании *метода раздваивания*, или *дихотомизации атрибутов* (binarization of attributes). Раздваивание атрибута осуществляется путем разбиения множества его значений на два подмножества. В результате этого разбиения приводит к созданию нового (двухзначного) атрибута, два значения которого соответствуют двум подмножествам. Если такое подмножество содержит больше одного

значения, его можно разбить еще на два подмножества, что приведет к созданию еще одного двухзначного атрибута, и т.д. Критерием выбора качественного разбиения обычно является максимальное увеличение приращения информации, приобретаемой с помощью полученного таким образом двухзначного атрибута. После преобразования всех атрибутов в двухзначные проблема сравнения многозначных атрибутов с немногочисленными исчезает. При этом даже приведенный выше простой критерий определения остаточной информации обеспечивает достоверное сравнение атрибутов (двухзначных).

Применяются также другие научно обоснованные критерии определения количества посторонних включений, такие как индекс Gini, определяемый следующим образом:

$$Gini = \sum_{i \neq j} p(i) p(j)$$

где  $i$  и  $j$  обозначают классы. После применения атрибута  $A$  формула определения результирующего индекса Gini принимает следующий вид:

$$Gini(A) = \sum_v p(v) \sum_{i \neq j} p(i | v) p(j | v)$$

где  $v$  — значения атрибута  $A$ ,  $p(i | v)$  — условная вероятность класса  $i$ , если дано, что атрибут  $A$  имеет значение  $v$ .

Следует отметить, что критерии определения количества посторонних включений, используемые в этом разделе, служат для оценки влияния одного атрибута. Поэтому критерий наибольшей информативности является локальным в том смысле, что он не позволяет надежно предсказывать совокупный эффект совместного использования нескольких атрибутов. Основной алгоритм логического вывода дерева основан на такой локальной минимизации количества посторонних включений. Как было описано выше, для глобальной оптимизации может потребоваться гораздо больше вычислительных ресурсов.

## Упражнения

- 18.1. Рассмотрите задачу обучения на силуэтах объектов (см. листинг 18.1). Рассчитайте энтропию для всего множества примеров по отношению к классу, остаточную информацию для атрибутов “size” и “holes”, соответствующие приращения информации и коэффициенты приращения. Оцените вероятности, необходимые для этих вычислений, с помощью относительных частот, например  $p(\text{nut}) = 3/12$  или  $p(\text{nut} | \text{holes}=1) = 3/5$ .
- 18.2. Заболевание  $D$  возникает в 25% из всех случаев заболеваний. Симптом  $S$  наблюдается у 75% пациентов, страдающих от заболевания  $D$ , и только в одном из шести прочих случаев. Предположим, что вы формируете дерево решения для диагностики заболевания  $D$ , состоящее только из двух классов,  $D$  (некоторое лицо страдает от заболевания  $D$ ) и  $\sim D$  (некоторое лицо не имеет заболевания  $D$ ). Симптом  $S$  является одним из соответствующих атрибутов. Какowymi являются приращение информации и коэффициент приращения для атрибута  $S$ ?

## 18.5.3. Реализация процедуры обучения с помощью дерева решения

Наметим структуру процедуры логического вывода деревьев решения следующим образом:

```
induce_tree(Attributes, Examples, Tree)
```

где Tree — дерево решения, полученное путем логического вывода из примеров Examples с помощью атрибутов в списке Attributes. Если рассматриваются примеры и атрибуты, приведенные в листинге 18.1, то можно собрать все примеры и доступные атрибуты в списки, применяемые в качестве параметров для рассматриваемой процедуры логического вывода, следующим образом:

```
induce_tree(Tree) :-
 findall(example(Class, Obj), example(Class, Obj), Examples),
 findall(Att, attribute(Att, _), Attributes),
 induce_tree(Attributes, Examples, Tree).
```

Форма дерева определяется в соответствии с описанными ниже тремя случаями.

1. Tree = null, если множество примеров является пустым.
2. Tree = leaf( Class), если все примеры относятся к одному и тому же классу Class.
3. Tree = tree( Attribute, [ Val1 : SubTree1, Val2 : SubTree2, ... ]), если примеры относятся больше чем к одному классу, Attribute является корнем дерева, Val1, Val2, ... представляют собой значения атрибута Attribute, а SubTree1, SubTree2, ... — соответствующие поддеревья решения.

Эти три случая учитываются с помощью следующих трех предложений:

```
% induce_tree(Attributes, Examples, Tree)
induce_tree(_, [], null) :- !.

induce_tree(_, [example(Class, _) | Examples], leaf(Class)) :-
 not (member(example(ClassX, _), Examples), % Других примеров
 ClassX \= Class), !. % иного класса нет

induce_tree(Attributes, Examples, tree(Attribute, SubTrees)) :-
 choose_attribute(Attributes, Examples, Attribute),
 delete(Attribute, Attributes, RestAtts), % Удалить атрибут Attribute
 attribute(Attribute, Values),
 induce_trees(Attribute, Values, RestAtts, Examples, SubTrees).
```

Процедура induce\_trees осуществляет логический вывод решения SubTrees для подмножеств множества Examples в соответствии со значениями Values атрибута Attribute следующим образом:

```
% induce_trees(Att, Vals, RestAtts, Examples, SubTrees)
induce_trees(_, [], _, _, []). % Нет ни атрибутов, ни поддеревьев

induce_trees(Att, [Val1 | Vals], RestAtts, Exs, [Val1 : Tree1 | Trees]) :-
 attval_subset(Att = Val1, Exs, ExampleSubset),
 induce_tree(RestAtts, ExampleSubset, Tree1),
 induce_trees(Att, Vals, RestAtts, Exs, Trees).
```

Выражение attval\_subset( Attribute = Value, Examples, Subset) является истинным, если Subset — это подмножество примеров в множестве Examples, которые удовлетворяют условию Attribute = Value следующим образом:

```
attval_subset(Attribute Value, Examples, ExampleSubset) :-
 findall(example(Class, Obj),
 (member(example(Class, Obj), Examples),
 satisfy(Obj, [AttributeValue]))), ExampleSubset).
```

Определение предиката satisfy( Object, Description) приведено в листинге 18.3. Предикат choose\_attribute позволяет выбрать атрибут, который обеспечивает наилучшее распознавание отдельных классов. Для этого требуется использовать критерий определения количества посторонних включений. Следующее предложение позволяет свести к минимуму значение выбранного критерия оценки количества посторонних включений с помощью предиката setof. Этот предикат упорядочивает доступные атрибуты по возрастанию количества посторонних включений, как показано ниже.

```
choose_attribute(Atts, Examples, BestAtt) :-
 setof(Impurity/Att,
 (member(Att, Atts), impurity1(Examples, Att, Impurity)),
 [MinImpurity/BestAtt | _]).
```

### Процедура

```
impurity1(Examples, Attribute, Impurity)
```

реализует выбранный критерий определения количества посторонних включений; где Impurity — это результирующее значение количества посторонних включений для подмножеств примеров после разделения списка Examples в соответствии со значениями атрибута Attribute.

## Упражнения

**18.3.** Реализуйте выбранный критерий определения количества посторонних включений, разработав для этого предикат impurity1. В качестве такого критерия может, например, использоваться остаточное информационное наполнение или индекс Gini, как описано выше в данном разделе. Для примеров, приведенных в листинге 18.1, и атрибута size применение индекса Gini в качестве критерия количества посторонних включений позволяет получить следующие результаты:

```
?- Examples = ... % Примеры, приведенные в листинге 18.1
 impurity1(Examples, size, Impurity).
 Impurity = 0.647619
```

А если вероятности аппроксимируются с помощью относительных частот, то значение Impurity вычисляется так:

```
Impurity = Gini(size)
 = p(small)*(p(nut | small)*p(screw | small)+...)+p(large)*(...)
 = 7/12 * (3/7 * 2/7 + ...) + 5/12 * (...)
 = 7/12 * 0.653061 + 5/12 * 0.64
 = 0.647619
```

**18.4.** Завершите разработку программы логического вывода дерева, приведенной в этом разделе, и проверьте ее работу на некоторой задаче обучения, например, показанной в листинге 18.1. Обратите внимание на то, что процедура choose\_attribute, в которой используется предикат setof, является очень неэффективной и должна быть усовершенствована. Введите также в действие процедуру

```
show(DecisionTree)
```

для отображения деревьев решения в форме, удобной для чтения. Например, для дерева, показанного на рис. 18.10, подходящая форма выглядит следующим образом:

```
holes
 none
 size
 small ==> screw
 large ==> pen
 1
 shape
 long ==> key
 compact ==> nut
 other ==> null
 2
 size
 small ==> key
 large ==> scissors
 3 ==> null
many ==> null
```

## 18.6. Обучение по зашумленным данным и отсечение частей деревьев

Во многих приложениях данные, применяемые для обучения, далеки от идеальных. Одна из распространенных проблем состоит в наличии ошибок в значениях атрибутов и определениях классов. В подобных случаях принято считать, что данные *зашумлены*. Безусловно, что при наличии шума задача обучения усложняется, поэтому требует использования специальных механизмов. Обычно в случае зашумленных данных не применяется требование единообразия, согласно которому гипотезы, полученные с помощью логического вывода, должны повторно классифицировать все учебные примеры правильно. Поэтому допускается, что гипотезы, сформированные в результате обучения, могут неправильно классифицировать некоторые из изучаемых объектов. Такое отступление от жесткой линии оправдано наличием возможных ошибок в данных. При этом остается надежда, что неправильно классифицированные изучаемые объекты относятся к числу именно тех объектов, которые содержат ошибки. Неправильная классификация таких объектов показывает лишь то, что ошибочные данные действительно были успешно проигнорированы.

При использовании простого алгоритма логического вывода дерева для формирования деревьев решения на основе зашумленных данных возникают две проблемы: во-первых, деревья, сформированные путем логического вывода, не обеспечивают надежной классификации новых объектов и, во-вторых, сформированные таким образом деревья обычно становятся слишком разветвленными и поэтому сложными для понимания. Можно показать, что в определенной степени усложнение подобного дерева является результатом наличия шума в обучающих данных. Алгоритм обучения не только обнаруживает фундаментальные закономерности в проблемной области, но и отслеживает весь шум в данных.

В качестве примера рассмотрим ситуацию, в которой необходимо сформировать поддерево дерева решения и текущим подмножеством объектов для обучения является  $S$ . Предположим, что в множестве  $S$  имеется 100 объектов, причем 99 из них относятся к классу  $C_1$ , а один — к классу  $C_2$ . Если известно, что в учебных данных присутствует шум и что все объекты, выбранные до сих пор в дереве решения, имеют одни и те же значения атрибутов, то можно с полной уверенностью предположить, что появление объекта класса  $C_2$  в множестве  $S$  является лишь результатом ошибки в данных. Если это действительно так, то лучше всего проигнорировать такой объект и просто возвратить лист дерева решения, обозначенный именем класса  $C_1$ . Поскольку в этой ситуации основной алгоритм логического вывода дерева должен был бы дальше развертывать дерево решения, то, остановившись в этой точке, мы фактически отсекаем одно из поддеревьев полного дерева решения.

*Отсечение частей деревьев* — это основной способ борьбы с шумом в программах логического вывода дерева. Такая программа может эффективно отсекают части деревьев решения с использованием некоторого критерия, который указывает, нужно ли в данный момент останавливать развертывание дерева или нет. Критерий остановки обычно учитывает количество примеров, соответствующих узлу, степень доминирования наиболее многочисленного класса в этом узле, степень влияния выбора дополнительного атрибута в данном узле на количество посторонних включений в этом множестве примеров и т.д.

Отсечение такого рода, осуществляемое путем прекращения развертывания дерева, называется *предварительным отсечением* (forward pruning), в отличие от другого вида отсечения, называемого *последующим отсечением* (post-pruning). Последующее отсечение выполняется после того, как обучающаяся программа сформирует полное дерево решения. Затем те части дерева, которые кажутся ненадежными, отсекаются. Такой процесс схематически показан на рис. 18.12. После удаления нижних частей дерева точность распознавания новых данных с помощью этого дерева может повыситься. Такое утверждение на первый взгляд кажется парадоксальным, ведь при отсечении фактически отбрасывается часть информации. Почему же в результате может повыситься точность?

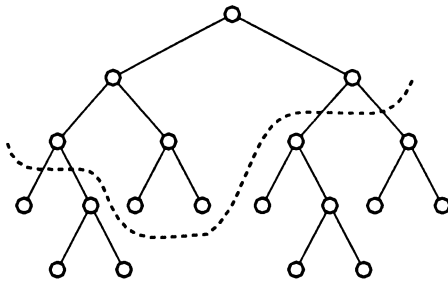


Рис. 18.12. Отсечение частей дерева решения. После отсечения точность распознавания может повыситься

Это связано с тем, что фактически осуществляется отсечение ненадежных частей дерева, т.е. тех частей, которые вносят наибольший вклад в возникновение ошибок из-за неправильной классификации с помощью этого дерева. Это — те части дерева, которые в основном отслеживают ошибки в данных, а не фундаментальные закономерности в данной области машинного обучения. Интуитивно можно легко понять, почему наименее надежными являются нижние части дерева. В рассматриваемом алгоритме нисходящего логического вывода дерева при построении верхней части дерева учитываются все обучающие данные, а после перехода на нижние уровни дерева обучающие данные фрагментарно распределяются по поддеревьям. Поэтому логический вывод расположенных ниже частей дерева осуществляется на основе меньшего объема данных. Чем меньше объем данных, тем выше опасность того, что он подвергнется существенному влиянию шума. Именно поэтому нижние части дерева обычно менее надежны.

Из двух методов отсечения (предварительного отсечения и последующего отсечения) последний считается лучшим, поскольку в нем используется информация, представленная всем деревом. При предварительном отсечении, с другой стороны, используется только информация из верхней части дерева.

Но остается нерешенной важная задача — как точно определить, какие поддерева нужно отсекают, а какие — нет. Если будет выполнено слишком обширное отсечение, то может быть отброшена также полезная информация и поэтому точность уменьшится. Итак, как определить, охватывает ли выбранный способ отсечения слишком большую или слишком малую часть дерева? Это — сложный вопрос, и существует несколько методов, которые позволяют найти на него разные, более или менее удовлетворительные ответы. Рассмотрим один из методов последующего отсечения, известный под названием *отсечение с минимальной ошибкой* (minimal error pruning).

Наиболее важное решение при использовании метода последующего отсечения состоит в том, нужно ли отсекают поддерева ниже указанного узла или нет. Такая ситуация иллюстрируется на рис. 18.13, где  $T$  — дерево решения, корнем которого является узел  $s$ ,  $T_1, T_2, \dots$  — поддерева дерева  $T$ ,  $p_i$  — вероятности того, что случайный объект будет передан из корня  $s$  в поддерево  $T_i$ . Дерево  $T$ , в свою очередь, может оказаться поддеревом более крупного дерева решения. Задача состоит в том, чтобы определить необходимость отсечения поддеревьев ниже узла  $s$  (т.е. удаления поддеревьев  $T_1, \dots$ ). Сформулируем некоторый критерий, основанный на принципе минимизации ожидаемой ошибки классификации. Предполагается, что в поддеревьях  $T_1, \dots$  уже было выполнено оптимальное отсечение их поддеревьев на основании того же критерия.

*Точностью классификации* дерева решения  $T$  является вероятность того, что  $T$  правильно классифицирует случайно выбранный новый объект. *Ошибкой классификации*  $T$  называется показатель с противоположным значением, т.е. вероятность неправильной классификации. Проанализируем эту ошибку для двух случаев, описанных ниже.

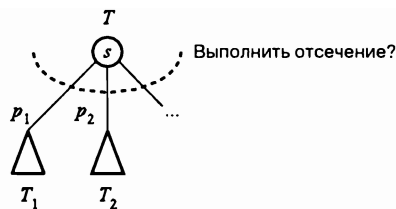


Рис. 18.13. Принцип принятия решения об отсечении частей дерева решения

1. Если отсечение поддеревьев дерева  $T$  осуществляется непосредственно ниже узла  $s$ , то  $s$  становится листом. В таком случае лист  $s$  обозначается именем наиболее вероятного класса  $C$  в листе  $s$  и все, что находится в этом листе, классифицируется как относящееся к классу  $C$ . Ошибка классификации в листе  $s$  представляет собой вероятность того, что случайно выбранный объект, который относится к  $s$ , принадлежит классу, отличному от  $C$ . Такая ошибка называется *статической ошибкой* (static error) в листе  $s$  и определяется по следующей формуле:

$$e(s) = p(\text{class} \neq C \mid s)$$

2. Если отсечение дерева не выполнено непосредственно ниже узла  $s$ , то ошибка в этом узле представляет собой взвешенную сумму ошибок  $E(T_1)$ ,  $E(T_2)$ , ... для поддеревьев  $T_1$ ,  $T_2$ , ..., отсечение которых выполнено оптимальным образом. Эта ошибка определяется по следующей формуле:

$$p_1 E(T_1) + p_2 E(T_2) + \dots$$

Такая ошибка называется *зафиксированной ошибкой* (backed-up error).

Таким образом, правило принятия решения об отсечении ниже узла  $s$  состоит в следующем: если статическая ошибка меньше или равна зафиксированной ошибке, то выполнять отсечение, в противном случае не выполнять. В соответствии с этим можно определить ошибку дерева  $T$ , в котором отсечение выполнено оптимальным образом, с помощью формулы

$$E(T) = \min(e(s), \sum_i p_i E(T_i))$$

Безусловно, если  $T$  не имеет поддеревьев, то эта формула принимает вид  $E(T) = e(s)$ .

Остается определить, как найти оценку статической ошибки  $e(s)$ , которая сводится к вероятности появления в узле  $s$  класса  $C$ , который чаще всего встречается в этом узле. Фактические данные, которые можно использовать для такой оценки, представляют собой множество примеров, относящихся к узлу  $s$ . Предположим, что это множество обозначено как  $S$ , общее количество примеров в  $S$  равно  $N$ , а количество экземпляров класса  $C$  равно  $n$ . Теперь наиболее сложная задача фактически сводится к определению вероятности появления экземпляра класса  $C$  в узле  $s$ . Большинство людей, которые сталкиваются с этой задачей, сразу же решают, что достаточно взять для этого отношение  $n/N$  (относительную частоту) для экземпляров класса  $C$  в узле  $s$ . Такое предложение является резонным, если количество экземпляров в узле  $s$  достаточно велико, но, безусловно, становится сомнительным, если это количество мало. Например, предположим, что к узлу  $s$  относится только один экземпляр класса  $C$ . В таком случае процентная доля наиболее вероятного класса составляет  $1/1 \cdot 100 = 100\%$ , а оценка ошибки равна  $0/1 = 0$ . Но если в узле  $s$  имеется только один экземпляр класса, то такая оценка становится статистически пол-

ностью недостоверной. Допустим, что мы смогли получить еще один учебный пример для узла  $s$  и этот пример относится к другому классу. В таком случае единственный дополнительный пример резко снизит вероятность оценки — до  $1/2 * 100 = 50\%$ !

Еще одной наглядной иллюстрацией к тому, насколько сложна задача оценки вероятностей, могут служить результаты подбрасывания определенной монеты. Предположим, что в первом эксперименте с монетой выпал орел. Итак, в соответствии с предложением об использовании относительной частоты мы должны оценить вероятность выпадения орла как 1. Это полностью противоречит здравому смыслу, поскольку априорно следовало ожидать, что эта вероятность равна 0,5. Даже если это — не совсем “идеальная” монета, вероятность выпадения орла все равно должна быть близка к 0,5, а оценка 1, безусловно, является неприемлемой. Этот пример также показывает, что оценка вероятности должна зависеть не только от результатов экспериментов, но также и от априорных ожиданий в отношении этой вероятности.

Очевидно, что требуется более обоснованная оценка, чем относительная частота. В этом разделе будет представлена одна из таких оценок, называемая *m-оценкой*, которая имеет глубокое математическое обоснование. В соответствии с *m-оценкой* ожидаемая вероятность события  $C$  состоит в следующем:

$$p = \frac{n + p_a \cdot m}{N + m}$$

где  $p_a$  — априорная вероятность  $C$ ,  $m$  — параметр оценки. Эта формула выведена с использованием байесовского подхода к оценке вероятности. В общем в байесовской процедуре предполагается, что имеется определенное, возможно очень приблизительное априорное представление о вероятности события  $C$ . Эти априорные знания рассматриваются как предварительное распределение вероятностей для события  $C$ . Затем проводятся эксперименты, дающие дополнительную информацию о вероятности  $C$ . Предварительное распределение вероятностей обновляется с использованием этой новой, экспериментальной информации; при этом применяется формула Байеса для условной вероятности. Приведенная выше формула *m-оценки* позволяет получить ожидаемое значение  $p$  этого распределения. Поэтому данная формула дает возможность учитывать предварительные ожидания в отношении вероятности  $C$ , а это удобно, если кроме заданных примеров имеются также некоторые ранее приобретенные знания о событии  $C$ . Такие предварительные ожидания в формуле *m-оценки* выражаются с помощью переменных  $p_a$  и  $m$ , как описано ниже.

Формула *m-оценки* может быть представлена также следующим образом:

$$p = p_a * \frac{m}{N+m} + \frac{n}{N} * \frac{N}{N+m}$$

Эта формула представляет собой удобную интерпретацию *m-оценки*: вероятность  $p$  равна априорной вероятности  $p_a$ , исправленной с учетом тех данных, которые содержатся в  $N$  дополнительных примерах. Если примеров больше нет, то  $N = 0$  и  $p = p_a$ . Если количество примеров велико ( $N$  — очень большое число), то  $p \approx n/N$ . В противном случае  $p$  находится между этими двумя значениями. Значимость предварительной оценки вероятности изменяется в зависимости от параметра  $m$  ( $m \geq 0$ ) — чем больше  $m$ , тем больше относительная значимость предварительной оценки вероятности.

Параметр  $m$  имеет особенно удобную интерпретацию при использовании зашумленных данных. Если специалист в рассматриваемой проблемной области считает, что данные очень зашумлены и следует больше доверять ранее полученным знаниям, то он может присвоить параметру  $m$  высокое значение (например,  $m = 100$ ) и придать тем самым больше веса предварительной оценке вероятности. Если, с другой стороны, заслуживают доверия обучающие данные, а предварительная оценка вероятности менее надежна, то можно установить низкое значение  $m$  (например,  $m = 0.2$ ) и таким образом придать больший вес данным. На практике, чтобы устранить неопределенность в отношении выбора наиболее подходящего значения параметра  $m$ , можно опробовать целый ряд его значений. Это позволяет получить последовательность деревьев с различной степенью отсекания поддеревьев, каждое из которых яв-



ляется оптимальным применительно к определенному значению  $m$ . Затем такая последовательность деревьев может быть изучена специалистом в проблемной области, который сможет выбрать наиболее подходящие деревья.

Остается нерешенным еще один важный вопрос о том, как определить предварительную оценку вероятности  $p_a$ . Если в распоряжении исследователя имеются экспертные знания, то значение  $p_a$  должно быть определено на их основе. В ином случае чаще всего используется метод определения предварительных оценок вероятностей по статистическим сведениям, относящимся ко всем обучающим данным (а не только к их фрагменту в узле  $s$ ), с использованием оценки относительной частоты на полном, крупном множестве. Альтернативный (чаще всего худший по сравнению с этим) способ состоит в том, что все классы априорно рассматриваются как равновероятные и поэтому имеющие единообразное предварительное распределение вероятностей. Это предположение приводит к частному случаю  $m$ -оценки, известному как *лапласовская оценка*. Если имеется всего  $k$  возможных классов, то для этого частного случая справедлива формула

$$p_a = 1/k, \quad m = k$$

Поэтому лапласовская оценка вероятности принимает следующий вид:

$$P = \frac{n + 1}{N + k}$$

Эта формула является удобной, поскольку для вычислений по ней не требуются параметры  $p_a$  и  $m$ . С другой стороны, она основана на том предположении, что все классы априорно являются равновероятными, которое обычно не соответствует действительности. Кроме того, эта формула не позволяет пользователю принять в расчет оценку степени зашумленности данных.

На рис. 18.14 показан процесс отсечения поддеревьев дерева решения по методу отсечения с минимальной ошибкой с использованием лапласовской оценки. На этом рисунке самый левый лист неотсеченного поддерева характеризуется частотами классов [3, 2]. Это означает, что данному листу соответствуют три объекта класса  $C1$  и два объекта класса  $C2$ . Статическая оценка ошибки для частот этих классов, полученная с использованием формулы лапласовской оценки вероятности, может быть представлена следующим образом:

$$e(b_{\text{left}}) = 1 - \frac{n+1}{N+k} = 1 - \frac{3+1}{5+2} = 0.429$$

Для правого дочернего узла  $b$  оценка вероятности ошибки составляет  $e(b_{\text{right}}) = 0.333$ . Для узла  $b$  статическая оценка ошибки представляет собой следующее:

$$e(b) = 0.375$$

Резервированная оценка ошибки для узла  $b$  может быть представлена таким образом:

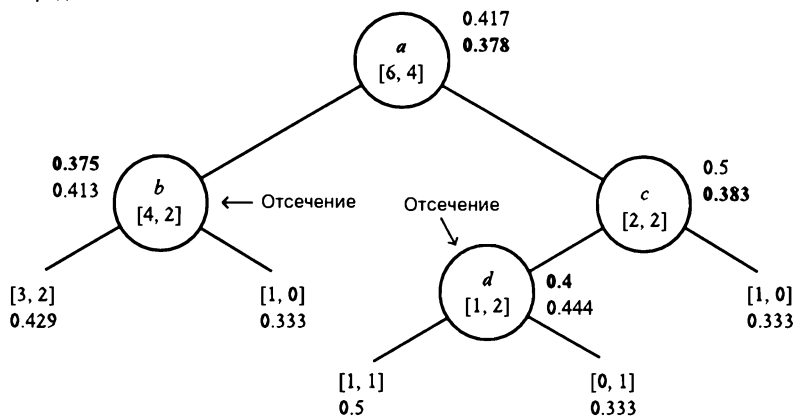
$$\text{BackUpError}(b) = 5/6 * 0.429 + 1/6 * 0.333 == 0.413$$

Поскольку резервированная оценка выше по сравнению со статической оценкой, поддеревья узла  $b$  отсекаются и ошибка в узле  $b$  после отсечения представляет собой следующее:

$$E(b) = 0.375$$

Метод использования  $m$ -оценки для оценивания ошибок в узлах дерева решения может быть подвергнут критике по следующим причинам. В формуле  $m$ -оценки предполагается, что доступные данные представляют собой случайно выбранный образец. Но это не совсем справедливо применительно к подмножествам данных в узлах дерева решения. В процессе формирования дерева подмножества данных в узлах дерева были выбраны из обучающих данных на основании критерия выбора атрибута среди возможных вариантов распределения, в соответствии со значениями атрибутов. Но несмотря на эти теоретические возражения, эксперименты показали, что метод отсечения с минимальной ошибкой при использовании  $m$ -оценки хорошо действует на практике. Особенно удобной является возможность проведения экспериментов по отсечению с применением различных значений параметра  $m$  во время исследования данных с помощью деревьев решения.

Перед отсечением:



После отсечения:

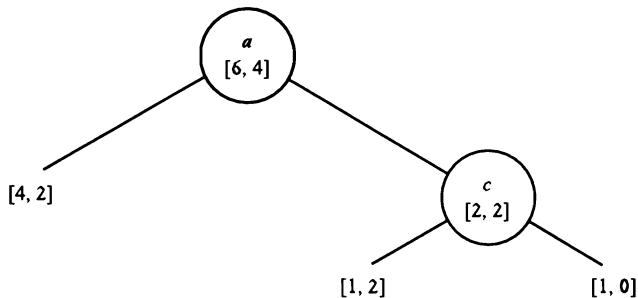


Рис. 18.14. Процесс отсечения поддеревьев дерева решения по методу отсечения с минимальной ошибкой. Пары чисел в квадратных скобках обозначают количество примеров классов  $C_1$  и  $C_2$ , которые попадают в соответствующие узлы. Другие числа, стоящие рядом с узлами, представляют собой оценки ошибок. Для внутренних узлов первое число — это статическая оценка ошибки, а второе — резервированная оценка. Наименьшее из этих двух чисел (обозначенное полужирным шрифтом) передается на верхний уровень

В методе отсечения с минимальной ошибкой в принципе может использоваться любой подход к оценке ошибок. В любом случае существует возможность свести к минимуму ожидаемую ошибку дерева с отсеченными поддеревьями. Безусловно, полученное таким образом усеченное дерево является оптимальным только по отношению к конкретным оценкам ошибок. Один из альтернативных подходов к оценке ошибок состоит в разделении доступных обучающих данных  $S$  на два подмножества — *растущее множество* (growing set) и *отсекаемое множество* (pruning set). Растущее множество используется для формирования полного дерева решения, совместимого с “растущими” данными. Отсекаемое множество применяется только для измерения ошибки в узлах дерева, а затем отсечения, выполняемого таким образом, чтобы эта ошибка стала минимальной. Поскольку отсекаемое множество является независимым от растущего множества, появляется возможность классифицировать примеры в отсекаемом множестве с помощью дерева решения и подсчитывать коли-

чество ошибок дерева. Это позволяет получить оценку прогнозируемой точности распознавания новых данных. Но подход, предусматривающий использование отсекаемого множества, является приемлемым, только если количество обучающих данных достаточно велико. А если обучающие данные имеются лишь в ограниченном количестве, обнаруживаются значительные недостатки этого метода. В частности, передача части данных в отсекаемое множество приводит к тому, что количество данных, применимых для наращивания дерева, становится еще меньше.

## Упражнения

**18.5.** Некоторый вулкан выбрасывает лаву полностью случайным образом. Статистические данные, накопленные за продолжительное время, показывают, что этот вулкан был в среднем активным в течение одних суток из десяти, в остальные сутки — неактивным. За последние 30 суток он был активным в течение 25 суток. Такая активность, проявляющаяся в последнее время, была учтена экспертами в прогнозе на следующие сутки. В новостях, переданных по радио, было сказано, что вероятность активности вулкана на следующие сутки составляет не меньше 30%, а в прогнозе, переданном телевизионной станцией, указано, что эта вероятность составляет 50–60%. Известно, что оба эксперта, работающих на радио и телевидении, используют метод *m*-оценки. Чем можно объяснить разницу в их оценках?

**18.6.** Напишите процедуру

```
prunetree(Tree, PrunedTree)
```

которая отсекает поддерева дерева решения *Tree* в соответствии с методом отсечения с минимальной ошибкой, описанным в данном разделе. Листья дерева *Tree* содержат информацию о частоте классов, представленную в виде списков целых чисел. Используйте в этой программе лапласовскую оценку. Допустим, что информация о количестве классов хранится в программе в виде факта `number_of_classes( K)`.

## 18.7. Успех обучения

В начале этой главы рассматривалась задача изучения реляционных описаний в стиле программы ARCHES. Программа ARCHES является хорошей иллюстрацией основных принципов и идей в области машинного обучения. Кроме того, принятый в ней последовательный стиль обработки примеров, вероятно, ближе к используемому людьми стилю обучения по сравнению с обучением за один сеанс, принятым в других двух подходах, описанных в данной главе. С другой стороны, процесс обучения с помощью описаний атрибутов и значений, представленных в форме правил или деревьев, является более простым и легким для понимания по сравнению с обучением на основе реляционных описаний. Поэтому до сих пор эти более простые подходы привлекали большее внимание исследователей и позволили добиться значительного успеха в создании практических приложений.

Было доказано, что обучение по своей сути представляет собой комбинаторный процесс, который связан с поиском среди возможных гипотез. Для управления этим поиском могут применяться эвристические методы. Кроме того, сам поиск может быть в значительной степени ограничен рамками “лингвистических возможностей”; иными словами, он может испытывать влияние языка гипотез, позволяющего формировать только гипотезы, имеющие определенные формы. Оба описанных выше алгоритма обучения с помощью правил вывода и деревьев решения были строго управляемыми с помощью эвристических методов, например, с помощью критерия оценки количества посторонних включений. Поэтому данные алгоритмы являются эффективными, даже несмотря на то, что в реализации на языке Prolog (см. листинг 18.3) сама оценка такой эвристики не была осуществлена эффективным образом. Для этих

эвристических функций требуется вычисление некоторых статистических данных, но язык Prolog не очень приспособлен для подобных целей.

В целом успех обучения измеряется с помощью нескольких критериев. В следующих разделах рассматриваются некоторые полезные критерии и обсуждаются преимущества усечения деревьев с точки зрения этих критериев.

### 18.7.1. Критерии успеха обучения

Ниже приведены некоторые критерии, широко применяемые для определения того, насколько успешными являются результаты работы обучающейся системы.

- **Точность классификации.** Этот показатель обычно определяется как процентная доля объектов, которые были правильно классифицированы с помощью гипотезы  $H$ , полученной с использованием логического вывода. Мы проводим различия между двумя типами точности классификации, как описано ниже.
  1. Точность классификации новых объектов, иными словами, объектов, не содержащихся в обучающем множестве  $S$ .
  2. Точность классификации объектов в множестве  $S$ . (Безусловно, этот показатель может представлять интерес, только если допускаются несовместимые гипотезы.)
- **Постижимость (понятность) гипотезы  $H$ , полученной путем логического вывода.** Часто очень важно, чтобы сформированное описание было понятным, поскольку лишь в этом случае пользователь может узнать что-то интересное о рассматриваемой проблемной области. Понятное описание может также использоваться непосредственно самими людьми, без помощи машины, для повышения собственного уровня знаний. В этом смысле машинное обучение является одним из подходов к выработке новых знаний с помощью компьютера. Одним из первых эту идею высказал Дональд Мичи [100]. Критерий понятности также очень важен, если сформированные с помощью логического вывода описания используются в экспертной системе, поведение которой должно быть вполне постижимым.
- **Вычислительная сложность.** Этот показатель позволяет оценить необходимый объем компьютерных ресурсов с точки зрения времени и пространства. Как правило, принято различать две области, в которых расходуются вычислительные ресурсы.
  1. Сложность формирования (ресурсы, необходимые для логического вывода описания понятия на основании примеров).
  2. Сложность выполнения (затраты ресурсов на классификацию объекта с помощью логически сформированного описания).

### 18.7.2. Оценка точности гипотез, полученных путем обучения

После завершения процесса обучения обычно возникает вопрос о том, насколько успешно гипотеза, полученная в результате обучения, позволяет предсказать, к какому классу относятся вновь поступившие данные. Безусловно, как только становятся доступными новые данные, эту точность можно просто измерить, проводя классификацию новых объектов и сравнивая их истинное значение класса с классом, предсказанным с помощью гипотезы. Но проблема состоит в том, что хотелось бы оценить эту точность еще до того, как станут доступными новые данные.

Обычный подход к оценке точности по новым данным состоит в разбиении случайным образом всех имеющихся данных на два множества: учебное и испытательное. Затем обучающаяся программа вызывается на выполнение на учебном множестве, а логически выведенные гипотезы проверяются на испытательном множестве так, как если бы данные этого множества представляли собой новые, будущие данные.

Такой подход является простым и не создает каких-либо проблем, если количество данных велико. Но наиболее распространенной ситуацией является недостаток данных. Предположим, что программа обучается составлению диагноза в определенной области медицины. В нашем распоряжении имеются только данные о последних пациентах, а количество этих данных не может быть увеличено. Если объем обучающих данных невелик, он может оказаться недостаточным для успешного обучения. В таком случае нехватка данных еще больше усугубляется тем, что часть данных необходимо выделить для использования в качестве испытательного множества.

Если количество обучающих примеров невелико, то результаты обучения и испытания подвержены значительным статистическим колебаниям. Они могут зависеть от того, как именно было выполнено разбиение данных на обучающее и испытательное множество. Для устранения такой статистической неопределенности процесс обучения и испытания повторяется несколько раз (как правило, десять) с использованием различных случайных разбиений. Затем результаты измерения точности усредняются, а их дисперсия позволяет получить представление о стабильности оценки.

Дальнейшим развитием метода проведения повторных испытаний такого типа является *k-кратная перекрестная проверка* (*k-fold cross-validation*). При использовании этого метода все обучающее множество случайным образом разбивается на  $k$  подмножеств. После этого обучение и испытание повторяются для каждого из этих подмножеств следующим образом:  $i$ -е подмножество удаляется из данных, остальная часть данных используется в качестве обучающего множества, после чего  $i$ -е подмножество применяется для испытания логически выведенной гипотезы. Полученные таким образом  $k$  результатов оценки точности усредняются, и вычисляется их дисперсия. Конкретного метода выбора значения  $k$  не существует, но в экспериментах по машинному обучению чаще всего применяется значение  $k = 10$ .

Особый случай перекрестной проверки возникает, если подмножества содержат только по одному элементу. В каждой итерации обучение осуществляется по всем данным, кроме одного примера, а логически выведенные гипотезы проверяются на оставшемся примере. Такая форма перекрестной проверки называется *исключением одного примера* (*leave-one-out*). Она является оправданной, если количество доступных данных особенно мало.

### 18.7.3. Влияние отсечения частей на точность и наглядность деревьев решения

Методы отсечения поддеревьев дерева решения являются исключительно важными, поскольку они оказывают благоприятное воздействие на процесс обучения при обработке зашумленных данных. Отсечение частей дерева решения оказывает свое воздействие на два критерия успеха обучения. Во-первых, оно способствует повышению точности классификации новых объектов с помощью дерева решения, а во-вторых, позволяет сделать дерево решения более наглядным. Рассмотрим более подробно оба эти результата отсечения.

Понятность описания зависит от его структуры и размеров. Удачно структурированное дерево решения проще понять, чем полностью неструктурированное. С другой стороны, если дерево решения невелико (состоит только из десяти или примерно такого небольшого количества узлов), то его можно легко понять независимо от структуры. Поскольку отсечение поддеревьев приводит к уменьшению размеров дерева, оно способствует лучшему пониманию дерева решения. Как было доказано экспериментально во многих проблемных областях, характеризующихся использованием зашумленных данных (таких как медицинская диагностика), сокращение размеров дерева может быть весьма значительным. В усеченном дереве количество узлов иногда составляет всего лишь десять процентов от первоначального количества, притом что сохраняется, по меньшей мере, такая же точность классификации.

Отсечение частей дерева позволяет также повысить точность классификации с помощью дерева. Такой результат отсечения поддеревьев может на первый взгляд

показаться противоречащим здравому смыслу, поскольку, отсекая часть поддерева, мы отбрасываем некоторую информацию, и может создаться впечатление, что в результате должна быть в некоторой степени потеряна точность. Но в случае обучения с использованием зашумленных данных отсечение некоторых поддеревьев (в какой-то приемлемой степени) обычно приводит к повышению точности. Этот феномен можно объяснить на основе теории статистики. С точки зрения статистики отсечение поддеревьев выполняет функции своего рода подавления шума. В результате отсечения мы устраняем ошибки в обучающих данных, возникшие под воздействием шума, а не отбрасываем полезную информацию.

## Проект

Осуществите типичный исследовательский проект в области машинного обучения. Он состоит в реализации алгоритма обучения и испытании его точности на множествах экспериментальных данных с использованием 10-кратной перекрестной проверки. Изучите, как влияет усечение дерева на точность классификации новых данных. Исследуйте влияние отсечения с минимальной ошибкой, варьируя значение параметра  $m$  в  $m$ -оценке. Большое количество наборов обучающих данных, взятых из практики, приведено в электронном виде для использования в подобных экспериментах в широко известном репозитории данных для машинного обучения UCI Repository for Machine Learning (Калифорнийский университет, г. Ирвин; <http://www.ics.uci.edu/~mllearn/MLRepository.html>).

## Резюме

- К основным формам обучения относятся *обучение путем сообщения необходимых знаний*, *обучение в результате открытия* и *обучение на примерах*. Освоение понятий на примерах называют также *индуктивным обучением*. Последняя форма обучения позволила добиться наиболее значительных успехов в создании практических приложений.
- Для обучения на примерах требуются следующие *информационные компоненты*:
  - объекты и понятия, представленные в виде множеств;
  - положительные и отрицательные примеры изучаемых понятий;
  - гипотезы о целевом понятии;
  - язык гипотез.
- *Задача обучения на примерах* состоит в формировании гипотезы, которая достаточно хорошо “объясняет” предъявленные примеры. При этом можно надеяться, что такая гипотеза позволит также точно классифицировать и будущие примеры. Гипотеза является совместимой с обучающими примерами, если она классифицирует все учебные данные таким же образом, как указано в этих примерах.
- *Процесс индуктивного обучения* предусматривает поиск среди возможных гипотез. Такая задача по своей сути является комбинаторной. Для уменьшения комбинаторной сложности этот процесс поиска обычно управляется с помощью эвристических методов.
- В процессе ее формирования гипотеза может быть *обобщена* или *конкретизирована*. Как правило, окончательная гипотеза представляет собой обобщение положительных примеров.
- В настоящей главе представлены следующие программы:
  - программа, которая в результате обучения формирует правила вывода на основании примеров, сформулированных в виде векторов атрибутов и значений;

- программа, которая в результате обучения формирует деревья решения на основании примеров, сформулированных в виде векторов атрибутов и значений.
- *Отсечение поддеревьев дерева решения* — это мощный подход к организации обучения с использованием зашумленных данных. В главе подробно описан метод отсечения с минимальной ошибкой.
- Показана сложность *оценки вероятностей* на основе малых выборок и сформулировано понятие *m-оценки*.
- *Критерии оценки* того, насколько успешным явилось применение некоторого метода обучения на примерах, включают следующие:
  - точность логически выведенных гипотез;
  - постижимость формулировок понятий, составленных в результате обучения;
  - вычислительная эффективность, во-первых, логического вывода гипотез на основании данных, а во-вторых, классификации новых объектов с помощью логически выведенных гипотез.
- *Ожидаемая точность гипотез*, сформированных в результате обучения, при обработке новых данных обычно оценивается с помощью перекрестной проверки. Чаще всего используется 10-кратная перекрестная проверка. Особой формой перекрестной проверки является метод с исключением одного примера.
- В этой главе рассматриваются следующие понятия:
  - машинное обучение;
  - изучение понятий на примерах, индуктивное обучение;
  - языки гипотез;
  - реляционные описания;
  - описания атрибутов и значений;
  - общность и конкретность гипотез;
  - обобщение и конкретизация описаний;
  - формирование путем обучения реляционных описаний по такому же принципу, как в программе ARCHES;
  - формирование путем обучения правил вывода;
  - нисходящий логический вывод деревьев решения;
  - обучение с использованием зашумленных данных;
  - отсечение частей дерева, последующее отсечение, отсечение с минимальной ошибкой;
  - оценка вероятностей;
  - перекрестная проверка.

## Дополнительные источники информации

Превосходное общее введение в проблематику машинного обучения приведено в [105]. В [97] описаны программы машинного обучения для самых различных областей, начиная от инженерного проектирования и заканчивая медициной, биологией и музыкой. Результаты современных исследований в области машинного обучения публикуются в литературе по искусственному интеллекту; наиболее известными из этих источников являются журналы *Machine Learning* (Boston: Kluwer) и *Artificial Intelligence* (Amsterdam: North-Holland), а также сборники материалов ежегодных конференций ICML (International Conference on Machine Learning — международная

конференция по машинному обучению), ECML (European Conference on Machine Learning — европейская конференция по машинному обучению) и COLT (Computational Learning Theory — теория компьютерного обучения). Значительное количество классических статей по машинному обучению включено в [75], [98] и [99]. В [59] приведены результаты изучения важности технических достижений в области машинного обучения и перспектив применения этого направления искусственного интеллекта для разрешения традиционного противоречия в философии науки.

Большое количество наборов данных для проведения экспериментов с новыми методами машинного обучения приведено в электронном виде в репозитории UCI (Калифорнийский университет, г. Ирвин; <http://www.ics.uci.edu/~mllearn/MLRepository.html>).

Приведенный в данной главе пример реляционных описаний, сформированных с помощью обучения (раздел 18.3), основан на идеях, которые воплощены в одной из первых обучающихся программ ARCHES [169]. Программа, приведенная в разделе 18.4, которая формирует описания на основе атрибутов, представляет собой упрощенную версию алгоритмов обучения типа AQ. Семейство AQ обучающихся программ было разработано Михальски (Michalski) и его сотрудниками (см., например, [96]). Широко известной программой для формирования правил вывода в результате обучения является CN2 [31].

Одним из наиболее широко применяемых подходов к обучению является логический вывод деревьев решения; этот метод известен также под названием TDIDT (Top-Down Induction of Decision Trees — нисходящий логический вывод деревьев решения), предложенный в [128]. На разработку метода обучения TDIDT большое влияние оказала одна из первых программ Куинлана (Quinlan) ID3 — Iterative Dichotomizer 3 [127], поэтому алгоритм логического вывода дерева решения часто называют просто ID3. Тема логического вывода деревьев изучалась также вне проблематики искусственного интеллекта Брейманом (Breiman) и его сотрудниками [23], которые независимо открыли несколько относящихся к этой теме механизмов, включая отсечение поддеревьев дерева решения. Метод отсечения дерева с минимальной ошибкой был предложен в [114] и усовершенствован в результате введения  $m$ -оценки в [27]. Формула  $m$ -оценки была выведена в [26] с использованием байесовской процедуры оценки вероятности. В [47] приведены результаты подробного экспериментального сопоставления различных методов усечения дерева. Кроме отсечения, в основной алгоритм логического вывода дерева решения необходимо внести другие усовершенствования для того, чтобы он мог применяться в качестве практического инструментального средства в сложных приложениях, связанных с обработкой реальных данных. Несколько подобных усовершенствований наряду с результатами первых экспериментов, в которых качество знаний, полученных в результате машинного обучения, превзошло качество знаний экспертов-людей, представлены в [28] и [128]. Результаты исследования в области структуризации деревьев решения для улучшения их наглядности приведены в [145].

Необходимо также упомянуть некоторые из наиболее интересных исследований и подходов к машинному обучению, которые не описаны в этой главе.

Одной из первых попыток реализации идей машинного обучения является обучающаяся программа, которая играет в шашки, описанная в [137]. Эта программа постоянно совершенствовала применяемую функцию оценки позиции благодаря опыту, полученному в сыгранных ею партиях.

В [104] предложена идея пространства версий, которая представляет собой попытку повысить эффективность поиска среди альтернативных описаний понятия.

В процессе активного обучения ученик исследует свою среду, выполняя действия и получая поощрения и наказания из этой среды [155]. В ходе обучения таким действиям, которые приводят к максимальному увеличению количества поощрений, сложность состоит в том, что поощрение и наказание могут запаздывать, поэтому не всегда ясно, какие именно действия послужили причиной успеха или неудачи.



При обучении на основе конкретных ситуаций, которое относится к обучению по принципу формирования рассуждений об определенных случаях [76], ученик рассуждает об указанном случае, сравнивая его с аналогичными предыдущими случаями.

Одним из важных направлений исследований в области обучения являются нейронные сети (представленные также в [105]), которые позволяют в определенной степени имитировать обучение биологических объектов. Обучение происходит в результате корректировки числовых весовых коэффициентов, связанных с искусственными нейронами сети. Нейронные сети никогда не были в фокусе исследований по искусственному интеллекту, поскольку они не допускают явного символического представления полученных результатов обучения. Гипотезы, сформированные в ходе нейронного обучения, могут обеспечивать очень качественное прогнозирование, но людям их трудно понимать и интерпретировать.

Одним из интересных подходов к обучению является обучение на основе объяснения (например, [106]); этот подход известен также под названием *аналитического обучения*. В данном случае обучающаяся система использует исходные знания для “объяснения” предъявленного примера с помощью дедуктивного процесса. В результате исходные знания преобразуются в форму, допускающую более эффективное использование. Один из методов обобщения на основе объяснения реализован в виде программы в главе 23 этой книги в качестве своего рода упражнения по метапрограммированию.

Некоторые обучающиеся программы обучаются по принципу самостоятельного совершения открытий. Они открывают новые понятия в результате исследований, проводя свои собственные эксперименты. Знаменитым примером программы такого рода является АМ (Automatic Mathematician — автоматический математик) [88]. Например, программа АМ начала свою работу с понятий множества и мультимножества (множества, допускающего вхождение одинаковых элементов), открыла понятия числа, сложения, умножения, простого числа и т.д. В [149] приведен сборник статей по использованию машинного обучения для научных открытий.

Средства формальной логики используются в качестве языка гипотез в одном из подходов к обучению, получившим название индуктивного логического программирования (Inductive Logic Programming — ILP), который рассматривается в следующей главе этой книги.

Математическая теория обучения известна также под названием COLT (COmputational Learning Theory — теория компьютерного обучения) [71]. Этот подход направлен на получение ответов на следующие вопросы: “Каким должно быть количество примеров для того, чтобы они позволили добиться некоторой заданной точности логически выведенных гипотез? Каковыми являются различные теоретические показатели измерения сложности обучения для разных классов языков гипотез?”

# Индуктивное логическое программирование

*В этой главе...*

|                                                |     |
|------------------------------------------------|-----|
| 19.1. Введение                                 | 446 |
| 19.2. Формирование программ Prolog на примерах | 449 |
| 19.3. Программа HYPER                          | 459 |

Индуктивное логическое программирование (Inductive Logic Programming — ILP) — это один из подходов к машинному обучению, в котором определения отношений формируются на основании примеров. В методе ILP в качестве языка гипотез используется логика предикатов, и результатом обучения обычно становится программа Prolog. Таким образом, программы Prolog создаются автоматически на основании примеров. Например, пользователь предоставляет некоторые примеры того, как можно и как нельзя сортировать списки, а программа сортировки списков создается автоматически. По сравнению с другими подходами к машинному обучению в методе ILP предусмотрен более общий способ определения *фоновых знаний*, т.е. знаний, известных ученику до начала обучения. Обычно за такую гибкость метода ILP приходится платить значительным повышением вычислительной сложности. В данной главе описывается разработка программы ILP под названием HYPER (HYPOthesis refinER), которая способна решать типичные задачи обучения в постановке ILP.

## 19.1. Введение

Индуктивное логическое программирование — это один из подходов к машинному обучению. Оно представляет собой метод изучения отношений на примерах. В методе ILP в качестве языка определения гипотез используется логика предикатов. Поэтому результатом обучения становится формула логики предикатов, обычно программа Prolog.

При таком подходе машинное обучение подобно автоматическому программированию, при котором пользователь (разработчик программ) не занимается сам написанием программ. Вместо этого он показывает на примерах, для выполнения каких действий предназначена создаваемая программа. Одни примеры (называемые *положительными*) указывают, что должна делать будущая программа, а другие примеры (называемые *отрицательными*) позволяют определить, что эта будущая программа не должна делать. Кроме того, пользователь задает некоторые *фоновые* предикаты, которые могут использоваться при написании намеченной новой программы.

Ниже приведен пример использования такого метода автоматического программирования на языке Prolog. Предположим, что уже имеются предикаты `parent(X, Y)`, `male(X)` и `female(X)`, определяющие некоторые семейные отношения

(см. рис. 1.1 в главе 1). Теперь необходимо найти определение нового предиката `has_daughter(X)`. Допустим, что даны некоторые примеры, относящиеся к этому новому предикату. В частности, ниже приведены два положительных примера.

```
has_daughter(tom), has_daughter(bob)
```

Двумя отрицательными примерами являются следующие:

```
has_daughter(pam), has_daughter(jim)
```

Задача обучения состоит в том, что нужно найти определение предиката `has_daughter(X)` в терминах предикатов `parent`, `male` и `female` таким образом, чтобы этот предикат принимал истинное значение для всех заданных положительных примеров и ложное значение — для всех заданных отрицательных примеров. Программа ILP может выработать следующую гипотезу в отношении предиката `has_daughter`:

```
has_daughter(X) :-
 parent(X, Y),
 female(Y).
```

Эта гипотеза объясняет все заданные примеры.

На приведенном выше примере можно отметить типичное отличие такого реляционного (основанного на отношениях) обучения от обучения на основе атрибутов и значений. В этом примере свойство `has_daughter` объекта `X` определено не только в терминах атрибутов `X`. Вместо этого для определения наличия у объекта `X` свойства `has_daughter` рассматривается другой объект, `Y`, связанный с `X`, и проверяются свойства объекта `Y`.

Теперь необходимо определить некоторые технические термины, применяемые в методе ILP. В приведенном выше примере предикат, который должен быть получен в результате обучения, `has_daughter`, называется *целевым предикатом*. Заданные предикаты `parent`, `male` и `female` называются *фоновыми знаниями* (Background Knowledge — ВК), или *фоновыми предикатами*. Это — знания, известные ученику до начала обучения. Итак, фоновые предикаты фактически определяют язык, на котором ученик может выражать гипотезы, относящиеся к целевому предикату.

Теперь мы можем обычным образом сформулировать задачу ILP, как описано ниже. Дано:

1. множество положительных примеров `E`. и множество отрицательных примеров `E-`;
2. фоновые знания ВК, заданные как множество логических формул, такое, что примеры `E`. невозможно вывести логически из ВК.

Найти:

гипотезу `H`, заданную как множество логических формул, такое, что

1. все положительные примеры в `E`. можно вывести логически из ВК и `H`;
2. ни одного отрицательного примера в `E-` нельзя вывести логически из ВК и `H`.

Принято также использовать такую формулировку, что `H` вместе с ВК должны охватывать все положительные примеры и не должны охватывать ни одного из отрицательных примеров. Подобная гипотеза `H` называется *полной* (охватывающей все положительные примеры) и *достоверной* (совместимой только с положительными примерами, т.е. не охватывающей ни одного отрицательного примера).

Обычно и ВК, и `H` представляют собой множества предложений Prolog, иными словами, программы Prolog. Поэтому с точки зрения автоматического программирования на языке Prolog приведенная выше формулировка задачи обучения соответствует следующей. Предположим, что целевым предикатом является `p(X)` и кроме прочих примеров имеются некоторые положительные пример `p(a)` и отрицательный пример `p(b)`. Возможный диалог с программой ВК может состоять в следующем:

```
?- p(a). % Положительный пример
no % Не может быть выведен логически из ВК
?- p(b). % Отрицательный пример
no % Не может быть выведен логически из ВК
```

Теперь предположим, что вызвана на выполнение система ILP, которая автоматически осуществляет логический вывод дополнительного множества предложений  $H$  и добавляет их к программе ВК. После этого возможный диалог с расширенной таким образом программой, состоящей из предложений ВК и  $H$ , может представлять собой следующее:

|          |                                               |
|----------|-----------------------------------------------|
| ?- p(a). | % Положительный пример                        |
| yes      | % Может быть выведен логически из ВК и $H$    |
| ?- p(b). | % Отрицательный пример                        |
| no       | % Не может быть выведен логически из ВК и $H$ |

Широко известными задачами в области ILP является автоматическое формирование программ Prolog для конкатенации или сортировки списков. Такие программы создаются на основании положительных примеров того, как должна осуществляться конкатенация или сортировка списков, а также отрицательных примеров того, как эти действия не должны выполняться. В настоящей главе разрабатывается программа ILP под названием HYPER и применяется для решения таких задач.

Рассмотрим, в чем состоят отличия метода ILP от других, нелогических подходов к машинному обучению, которые, по сути, представляют собой своего рода обучение на основе атрибутов и значений. Преимущество ILP состоит в том, что в этом методе используются мощный язык гипотез и перспективный способ включения фоновых знаний в процесс обучения. Язык гипотез позволяет применять реляционные определения, которые могут даже предусматривать рекурсию. Обычно такая возможность не обеспечивается при использовании других подходов к машинному обучению.

В методе ILP в качестве фоновых знаний, по сути, может применяться любая программа Prolog. Это дает возможность пользователю подготавливать для использования в обучении конкретные знания о проблемной области и представлять их наиболее естественным способом. Применение фоновых знаний позволяет пользователю подготавливать качественное представление задачи и задавать в процессе обучения конкретные ограничения, относящиеся к рассматриваемой задаче. В отличие от этого, в процессе обучения на основе атрибутов и значений фоновые знания обычно могут быть представлены только в весьма ограниченных формах, например в форме допустимых новых атрибутов. С другой стороны, метод ILP является гораздо более гибким. Например, если задача состоит в изучении свойств химических веществ, то в качестве фоновых знаний могут задаваться молекулярные структуры, которые определены в виде атомов и связей между ними. А если задача обучения состоит в автоматическом формировании модели физической системы по результатам наблюдений за ее действиями, то в состав фоновых знаний может быть включен весь математический аппарат, который рассматривается как приемлемый для описания моделируемой проблемной области. Если же речь идет о процессах, происходящих в физическом мире, то в состав фоновых знаний могут быть включены аксиомы, позволяющие формировать рассуждения о времени и пространстве. Как правило, при разработке приложений ILP основные усилия направлены на подготовку качественного представления примеров наряду с соответствующими фоновыми знаниями. После этого для осуществления логического вывода применяется система ILP общего назначения.

За возможность применения мощного языка гипотез и разнообразных фоновых знаний в методе ILP приходится платить. Чем больше вариантов рассматривается в задаче обучения, тем выше становится комбинаторная сложность. Поэтому методы обучения на основе атрибутов и значений, например с использованием деревьев решения, являются намного более эффективными по сравнению с методом ILP. В связи с этим по соображениям эффективности для решения таких задач обучения, для которых являются приемлемыми представления в виде атрибутов и значений, рекомендуется использовать методы обучения на основе атрибутов и значений.

В данной главе разрабатывается программа ILP, называемая HYPER (HYPOthesis refinER), которая формирует программы Prolog путем постепенного усовершенствования некоторых начальных гипотез. Для иллюстрации основных идей вначале будет создана простая и малоэффективная версия этой программы, называемая MINHYPER. Затем на ее основе будет разработана программа HYPER.

## 19.2. Формирование программ Prolog на примерах

### 19.2.1. Постановка задачи обучения

Рассмотрим еще раз пример с семейными отношениями для ознакомления с тем, каким образом можно автоматически формировать гипотезы на основании примеров. Предположим, что используются фоновые знания и примеры, приведенные в листинге 19.1. Соответствующий граф семейных отношений аналогичен приведенному на рис. 1.1 (см. главу 1) с тем дополнением, что Пэт имеет дочь Еву. Предполагается, что в программе, разрабатываемой в этом разделе, используются соглашения по представлению примеров, показанные в листинге 19.1. Положительные примеры представляются с помощью предиката `ex(Example)`, например, следующим образом:

```
ex(has_daughter(tom)). % Том имеет дочь
```

#### Листинг 19.1. Определение задачи изучения предиката `has_daughter`

```
% Изучение предиката по фактам, определяющим семейные отношения

% Фоновые знания

backliteral(parent(X,Y), [X,Y]). % Фоновый литерал с переменными [X,Y]

backliteral(male(X), [X]).

backliteral(female(X), [X]).

prolog_predicate(parent(_,_)). % Цель parent(_,_) выполняется
 % непосредственно интерпретатором Prolog
prolog_predicate(male(_)).
prolog_predicate(female(_)).

parent(pam, bob).
parent(tom, bob).
parent(tom, liz).
parent(bob, ann).
parent(bob, pat).
parent(pat, jim).
parent(pat, eve).

female(pam).
male(tom).
male(bob).
female(liz).
female(ann).
female(pat).
male(jim).
female(eve).

% Положительные примеры

ex(has_daughter(tom)). % Том имеет дочь
ex(has_daughter(bob)).
ex(has_daughter(pat)).

% Отрицательные примеры

nex(has_daughter(pam)). % Пэм не имеет дочери
nex(has_daughter(jim)).

start_hyp([[has_daughter(X)] / [X]]). % Начальная гипотеза
```

Отрицательные примеры представлены с помощью предиката `nex(Example)`, например, как показано ниже.

```
nex(has_daughter(pam)). % Пэм не имеет дочери
```

Предикат `backliteral` (сокращение от `background literal` — фоновый литерал) задает форму литералов, которые могут использоваться в программе ILP в качестве целей при формировании предложений Prolog. В частности, предикат `backliteral( parent(X,Y), [X,Y])`.

указывает, что в состав языка гипотез входят литералы в форме `parent(X,Y)`, в которых переменные `X` и `Y` могут быть переименованы. Вторым параметром предиката `backliteral` является список переменных в этом литерале. Такие *фондовые литералы* могут добавляться в качестве целей в тела формируемых предложений. Фондовые литералы представляют собой вызовы фоновых предикатов, которые могут быть заданы непосредственно на языке Prolog. Предикат `prolog_predicate` определяет цели, которые должны быть непосредственно выполнены интерпретатором Prolog. Например, предикат

```
prolog_predicate(parent(X,Y)).
```

указывает, что цели, которые согласуются с предикатом `parent(X,Y)`, обрабатываются интерпретатором Prolog непосредственно, путем выполнения фонового предиката `parent`. Другие цели, такие как `has_daughter(tom)`, могут быть выполнены с помощью интерпретатора, подобного Prolog, но с особыми свойствами, который будет реализован специально для использования в методе ILP.

## 19.2.2. Граф усовершенствования

Теперь рассмотрим, каким образом может быть сформирована полная и совместимая гипотеза для задачи обучения, приведенной в листинге 19.1. Мы можем начать с некоторой слишком общей гипотезы, которая является полной (охватывает все положительные примеры), но несовместимой (она является несовместимой в том смысле, что не охватывает лишь положительные примеры, т.е. охватывает также отрицательные примеры). Подобная гипотеза должна быть усовершенствована таким образом, чтобы она сохранила свою полноту и стала совместимой. Этого можно достичь путем поиска в пространстве возможных гипотез и их усовершенствований. При каждом усовершенствовании берется гипотеза  $H_1$  и вырабатывается более конкретная гипотеза  $H_2$ , такая, что  $H_2$  охватывает подмножество случаев, охватываемых гипотезой  $H_1$ .

Подобное пространство гипотез и их усовершенствований называется *графом усовершенствования*. На рис. 19.1 показана часть такого графа усовершенствования для задачи обучения, приведенной в листинге 19.1. Узлы в этом графе соответствуют гипотезам, а дуги между гипотезами соответствуют усовершенствованиям. Между гипотезами  $H_1$  и  $H_2$  имеется ориентированная дуга, если  $H_2$  является усовершенствованием  $H_1$ .

После определения графа усовершенствования задача обучения сводится к поиску в этом графе. Начальным узлом поиска является некоторая слишком общая гипотеза. Целевым узлом поиска становится совместимая и полная гипотеза. В примере, приведенном на рис. 19.1, достаточно, чтобы все гипотезы представляли собой отдельные предложения, но в общем гипотезы могут состоять из нескольких предложений.

Для реализации описанного подхода необходимо разработать два компонента.

1. *Оператор усовершенствования*, который будет вырабатывать усовершенствования гипотез (такой оператор определяет граф усовершенствования).
2. *Процедура поиска* для выполнения поиска.

В графе, приведенном на рис. 19.1, имеются усовершенствования двух типов. Усовершенствование любого предложения может быть получено одним из следующих способов.

1. Согласование двух переменных в предложении.
2. Добавление фонового литерала к телу предложения.

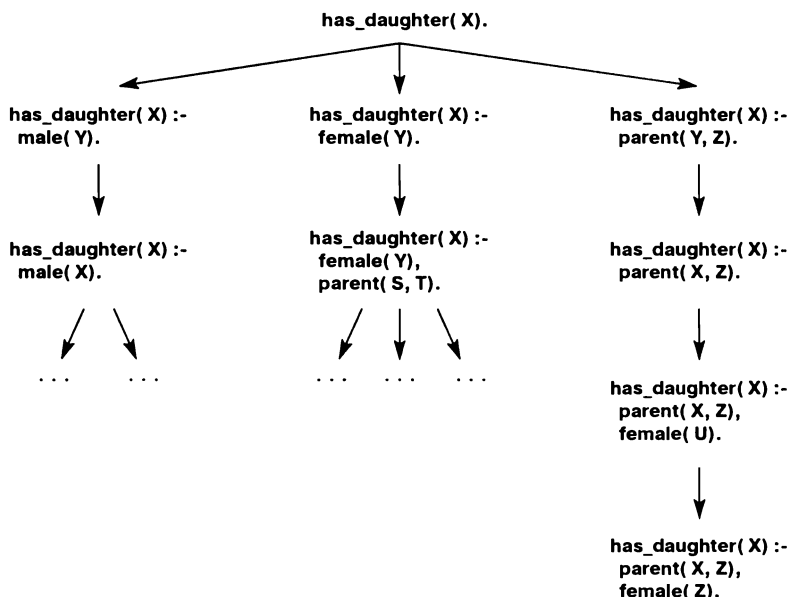


Рис. 19.1. Часть графа усовершенствования для задачи обучения, приведенной в листинге 19.1. На этой схеме не показано много других возможных усовершенствований

Примером усовершенствования первого типа является следующее предложение:

```
has_daughter(X) :- parent(Y, Z).
```

Это предложение путем усовершенствования преобразуется в предложение

```
has_daughter(X) :- parent(X, Z).
```

в котором выполнено согласование  $X=Y$ . Примером усовершенствования второго типа является операция, в результате которой предложение

```
has_daughter(X).
```

путем усовершенствования преобразуется в следующее предложение:

```
has_daughter(X) :- parent(Y, Z).
```

Необходимо отметить несколько важных моментов. Во-первых, каждое усовершенствование представляет собой *уточнение*. Иными словами, преемник любой гипотезы в графе усовершенствования охватывает лишь некоторое подмножество тех случаев, которые охвачены предшественником этой гипотезы. Поэтому в процессе поиска достаточно рассматривать только полные гипотезы (которые охватывают все положительные примеры). Неполная гипотеза в процессе усовершенствования не может стать полной.

Во-вторых, оператор усовершенствования должен осуществлять достаточно “малые” этапы усовершенствования. В противном случае оператор усовершенствования может не позволить выработать целевую гипотезу. Оператор, допускающий слишком крупные этапы усовершенствования, может перейти от полной и несовместимой гипотезы прямо к неполной и совместимой, минуя находящуюся между ними совместимую и полную гипотезу.

Может также рассматриваться еще один тип усовершенствования, который предусматривает *замену переменных структурированными термами*. Например, предложение

```
member(X1, L1) :- member(X1, L3).
```

может быть преобразовано в процессе усовершенствования в следующее предложение:

```
member(X1, [X2 | L2]) :- member(X1, L3).
```

В результате усовершенствования переменная `L1` преобразована в структуру `[X2 | L2]`. Для простоты в программе `MINIHYPER`, разрабатываемой в этом разделе, не предусмотрена возможность решать задачи, для которых требуются структурированные термы. Отложим введение этого средства до следующего раздела, в котором разрабатывается более сложная программа `HYPER`.

На основании рис. 19.1 можно прийти еще к одному очевидному выводу, что графы усовершенствования отличаются высокой комбинаторной сложностью и поэтому требуют больших затрат ресурсов в процессе поиска. В программе `MINIHYPER` это соображение также игнорируется и используется неуправляемый поиск с итеративным углублением. Программа `HYPER` будет улучшена и в этом отношении благодаря использованию поиска по заданному критерию.

### 19.2.3. Программа `MINIHYPER`

Теперь мы можем приступить к написанию нашей первой программы `ILP`. Гипотезы удобнее всего представлять в виде списка предложений следующим образом:

```
Hypothesis = [Clause1, Clause2, ...]
```

Каждое предложение представляется в виде списка литералов (головы предложения, за которым следуют литералы тела) и списка переменных в предложении, как показано ниже.

```
Clause = [Head, BodyLiteral1, BodyLiteral2, ...] / [Var1, Var2, ...]
```

Например, в соответствии с этим соглашением гипотеза

```
pred(X,Y) :- parent(X,Y).
pred(X,Z) :- parent(X,Y), pred(Y,Z).
```

должна быть представлена следующим образом:

```
[[pred(X1,Y1), parent(X1,Y1)] / [X1,Y1], [pred(X2,Z2), parent(X2,Y2),
 pred(Y2,Z2)] / [X2,Y2,Z2]]
```

Несмотря на то что нет особой необходимости явно представлять список переменных в предложении, это удобно с точки зрения реализации способа усовершенствования гипотез путем согласования переменных. Следует отметить, что для переменных каждого предложения в гипотезе должны быть предусмотрены различные имена, поскольку они фактически являются разными переменными.

Для того чтобы проверить, охватывает ли гипотеза некоторый пример, требуется интерпретатор типа `Prolog` для гипотез, представленных, как описано выше. Для этого определим следующий предикат:

```
prove(Goal, Hypothesis, Answer)
```

Этот предикат для заданной цели `Goal` и гипотезы `Hypothesis` находит ответ `Answer`, указывающий, можно ли вывести цель `Goal` логически из гипотезы `Hypothesis`. По сути, данный предикат должен предпринимать попытки доказать истинность `Goal` на основании гипотезы `Hypothesis` по принципу, аналогичному самому интерпретатору `Prolog`. Задача разработки такой программы относится к области метапрограммирования, которое рассматривается более подробно в главе 23. В данном случае особая сложность состоит в том, что необходимо избежать опасности возникновения бесконечных циклов. Применяемый оператор усовершенствования вполне может вырабатывать рекурсивные предложения, подобные следующему:

```
[p(X), p(X)]
```

Такое предложение соответствует предложению `p(X) :- p(X)`. Это может привести к бесконечному циклу. Необходимо добиться того, чтобы предикат `prove` не был восприимчивым к таким циклам. Проще всего этого можно достичь, ограничив длину доказательств. Если в процессе доказательства цели `Goal` количество вызовов предикатов достигает заданного предела, то процедура `prove` прекращает свою работу, не сформировав окончательного ответа. Таким образом, параметр `Answer` предиката `prove` может иметь перечисленные ниже возможные значения.



- Answer = yes. Цель Goal была логически выведена из гипотезы Hypothesis в пределах длины доказательства.
- Answer = no. Очевидно, что цель Goal невозможно вывести логически из гипотезы Hypothesis, даже если данный предел будет увеличен.
- Answer = maybe. Поиск доказательства был прекращен в связи с тем, что достигнуто значение максимальной длины доказательства D.

Случай "Answer = maybe" может соответствовать любому из перечисленных ниже трех вариантов, если цель Goal выполнялась с помощью стандартного интерпретатора.

1. Стандартный интерпретатор Prolog (без ограничения длины доказательства) вошел в бесконечный цикл.
2. Стандартный интерпретатор Prolog мог бы в конечном итоге найти доказательство, длина которого превышает предел D.
3. Стандартный интерпретатор Prolog мог бы определить на каком-то этапе доказательства, длина которого превышает D, что этот вариант логического вывода оканчивается неудачей. Поэтому он должен был бы перейти по методу перебора с возвратами к другому варианту и в этом случае либо найти доказательство (длина которого может оказаться меньше чем D), либо не достичь цели, либо войти в бесконечный цикл.

Код предиката prove приведен в листинге 19.2. Предел длины доказательства задается с помощью следующего предиката:

```
max_proof_length(D)
```

Значение D по умолчанию установлено равным 6, но может быть откорректировано в зависимости от конкретной задачи обучения. Вызовы фоновых предикатов (объявленных с помощью предиката prolog\_predicate) передаются стандартному интерпретатору Prolog и поэтому не вносят свой вклад в увеличение длины доказательства. Таким образом, учитываются только вызовы целевых предикатов, которые определены в гипотезе.

## Листинг 19.2. Интерпретатор гипотез, позволяющий избежать циклов

```
% Интерпретатор гипотез
% prove(Goal, Hypo, Ans):
% Ans = yes, если цель Goal можно логически вывести из гипотезы Hypo
% не больше, чем за D шагов
% Ans = no, если цель Goal нельзя вывести логическим путем
% Ans = maybe, если поиск заканчивается безрезультатно после D шагов

prove(Goal, Hypo, Answer) :-
 max_proof_length(D),
 prove(Goal, Hypo, D, RestD),
 (RestD >= 0, Answer = yes % Доказано
 ;
 RestD < 0, !, Answer = maybe % Возможно, что доказательство существует,
 % но пока ситуация напоминает бесконечный цикл
).

prove(Goal, _, no). % В противном случае цель Goal определено
 % нельзя доказать

% prove(Goal, Hyp, MaxD, RestD):
% MaxD - допустимая длина доказательства, RestD - длина, "оставшаяся
% неиспользованной" после доказательства; учитываются только шаги
% доказательства, в которых используется гипотеза Hyp

prove(G, H, D, D) :-
 D < 0, !. % Длина доказательства превышена
```

```

prove([], _, D, D) :- !.

prove([G1 | Gs], Hypo, D0, D) :- !,
 prove(G1, Hypo, D0, D1),
 prove(Gs, Hypo, D1, D).

prove(G, _, D, D) :-
 prolog_predicate(G), % Фоновый предикат на языке Prolog?
 call(G). % Вызвать фоновый предикат

prove(G, Hyp, D0, D) :-
 D0 =< 0, !, D is D0-1 % Доказательство - слишком длинное
 ;
 D1 is D0-1, % Оставшаяся длина доказательства
 member(Clause/Vars, Hyp), % Одно из предложений в гипотезе Hyp
 copy_term(Clause, [Head | Body]), % Переименовать переменные в предложении
 G = Head, % Согласовать голову предложения с целью
 prove(Body, Hyp, D1, D). % Доказать G с помощью предложения Clause

```

---

Эта обучающаяся программа реагирует на случаи “maybe”, как говорится, “осторожно” (так принято характеризовать данный способ организации работы программы), т.е. осуществляется *осторожная интерпретация*, поскольку в ней используется описанный ниже подход.

1. Если проводится доказательство положительного примера, то случай “maybe” трактуется как утверждение, что “данный пример не охвачен”.
2. Если проводится доказательство отрицательного примера, то случай “maybe” трактуется как отрицание утверждения, что этот отрицательный пример “не охвачен”. Иными словами, считается, что он все еще охвачен.

В пользу подобной осторожной интерпретации случая “maybe” свидетельствует то, что среди всех возможных полных гипотез наиболее предпочтительными являются гипотезы с высокой вычислительной эффективностью. Но ответ “maybe” в лучшем случае указывает, что гипотеза характеризуется низкой вычислительной эффективностью, а в худшем случае — что она является неполной.

Остальная часть программы MINIHYPHER приведена в листинге 19.3. В этой программе определены предикаты, описанные ниже.

### Листинг 19.3. Простая программа ILP — MINIHYPHER

---

```

% Программа MINIHYPHER

% induce(Hyp):
% осуществляет логический вывод совместимой и полной гипотезы Hyp путем
% постепенного усовершенствования начальных гипотез

induce(Hyp) :-
 iter_deep(Hyp, 0). % Итеративное углубление, начиная
 % с максимальной глубины 0

iter_deep(Hyp, MaxD) :-
 write('MaxD = '), write(MaxD), nl,
 start_hyp(Hyp0),
 complete(Hyp0), % Гипотеза Hyp0 охватывает все положительные примеры
 depth_first(Hyp0, Hyp, MaxD) % Поиск в глубину с ограничением глубины
 ;
 NewMaxD is MaxD + 1,
 iter_deep(Hyp, NewMaxD).

% depth_first(Hyp0, Hyp, MaxD):
% преобразует гипотезу Hyp0 путем усовершенствования в совместимую
% и полную гипотезу Hyp не больше чем за MaxD шагов

```

```

depth_first(Hyp, Hyp, _) :-
 consistent(Hyp).

depth_first(Hyp0, Hyp, MaxD0) :-
 MaxD0 > 0,
 MaxD1 is MaxD0 - 1,
 refine_hyp(Hyp0, Hyp1),
 complete(Hyp1), % Гипотеза Hyp1 охватывает все положительные примеры
 depth_first(Hyp1, Hyp, MaxD1).

complete(Hyp) :- % Гипотеза Hyp охватывает все положительные примеры
 not (ex(E), % Положительный пример
 once(prove(E, Hyp, Answer)), % Доказать его с помощью гипотезы Hyp
 Answer \== yes). % Возможно, что доказательство не найдено

consistent(Hyp) :- % Возможно, что гипотеза не охватывает ни одного
 % отрицательного примера
 not (nex(E), % Отрицательный пример
 once(prove(E, Hyp, Answer)), % Доказать его с помощью гипотезы Hyp
 Answer \== no). % Возможно, доказательство существует

% refine_hyp(Hyp0, Hyp):
% предикат, который позволяет усовершенствовать гипотезу Hyp0
% и получить гипотезу Hyp

refine_hyp(Hyp0, Hyp) :-
 conc(Clauses1, [Clause0/Vars0 | Clauses2], Hyp0), % Выбрать предложение
 % Clause0 из гипотезы Hyp0
 conc(Clauses1, [Clause/Vars | Clauses2], Hyp), % Новая гипотеза
 refine(Clause0, Vars0, Clause, Vars). % Усовершенствовать предложение Clause

% refine(Clause, Args, NewClause, NewArgs):
% предикат, который позволяет усовершенствовать предложение Clause
% с параметрами Args и получить предложение NewClause с параметрами NewArgs

% Усовершенствовать по методу согласования параметров

refine(Clause, Args, Clause, NewArgs) :-
 conc(Args1, [A | Args2], Args), % Выбрать переменную A
 member(A, Args2), % Согласовать ее с другой переменной
 conc(Args1, Args2, NewArgs).

% Усовершенствовать по методу добавления литерала

refine(Clause, Args, NewClause, NewArgs) :-
 length(Clause, L),
 max_clause_length(MaxL),
 L < MaxL,
 backliteral(Lit, Vars), % Литерал с определением фоновых знаний
 conc(Clause, [Lit], NewClause), % Добавить литерал к телу предложения
 conc(Args, Vars, NewArgs). % Добавить переменные литерала

% Значения параметров, заданные по умолчанию

max_proof_length(6). % Общая длина доказательства с учетом вызовов
 % предикатов, отличных от предикатов Prolog

max_clause_length(3). % Максимальное количество литералов в теле предложения

```

- `refine( Clause, Vars, NewClause, NewVars)`. Позволяет усовершенствовать заданное предложение `Clause` с помощью переменных `Vars` и выработать усовершенствованное предложение `NewClause` с новыми переменными `NewVars`. Усовершенствованное предложение формируется путем согласования двух переменных в списке `Vars` или добавления нового фонового литерала к

предложению Clause. Новые литералы добавляются лишь до тех пор, пока не будет достигнута определяемая пользователем максимальная длина предложения (заданная с помощью предиката `max_clause_length(MaxL)`).

- `refine_hyp( Hyp, NewHyp)`. Позволяет усовершенствовать гипотезу `Hyp` и выработать гипотезу `NewHyp`, выбирая недетерминированным образом одно из предложений в `Hyp` и применяя один из способов усовершенствования этого предложения.
- `induce( Hyp)`. Осуществляет логический вывод совместимой и полной гипотезы `Hyp` для указанной задачи обучения путем поиска с итеративным углублением в графе усовершенствования, начиная от начальной гипотезы (которая задана с помощью предиката `start_hyp(StartHyp)`).
- `iter_deep( Hyp, MaxD)`. Находит полную и совместимую гипотезу `Hyp` по методу поиска с итеративным углублением, начиная с предела глубины `MaxD` и увеличивая этот предел до тех пор, пока не будет найдена гипотеза `Hyp`. Полный поиск с итеративным углублением осуществляется с помощью вызова `iter_deep(Hyp, 0)`.
- `depth_first( Hyp0, Hyp, MaxD)`. Выполняет поиск в глубину, ограниченный глубиной `MaxD`, начиная с гипотезы `Hyp0`. Если поиск завершается успехом в пределах `MaxD`, то `Hyp` представляет собой полную и совместимую гипотезу, полученную в результате не больше чем `MaxD` последовательных усовершенствований гипотезы `Hyp0`.
- `complete( Hyp)`. Принимает истинное значение, если гипотеза `Hyp` охватывает все заданные положительные примеры (при условии осторожной интерпретации ответа `Answer` в предикате `prove/3`).
- `consistent( Hyp)`. Принимает истинное значение, если гипотеза `Hyp` не охватывает ни одного из заданных отрицательных примеров (при условии осторожной интерпретации ответа `Answer` в предикате `prove/3`).

Для использования программы `MINIHYPER` необходимо загрузить в систему Prolog код, приведенный в листингах 19.2 и 19.3, наряду с часто применяемыми предикатами `member/2`, `conc/3`, `not/1`, `once/1` и `copy_term/2` и с определением задачи обучения (подобным приведенному в листинге 19.1). Безусловно, что параметры `max_proof_length` и `max_clause_length` могут быть переопределены в соответствии с требованиями конкретной задачи обучения.

Например, процесс изучения предиката `has_daughter(X)` в соответствии с условиями, заданными в листинге 19.1, выглядит следующим образом:

```
?- induce(H).
MaxD = 0
MaxD = 1
MaxD = 2
MaxD = 3
MaxD = 4
H = [[has_daughter(A),parent(A,B),female(B)],[A,B]]
```

Возрастающие пределы глубины при поиске с итеративным углублением отображаются как `MaxD = Limit`. Совместимая и полная гипотеза обнаруживается на глубине усовершенствования 4 (см. рис. 19.1). Если бы в эту программу были введены счетчики гипотез, они показали бы, что количество всех выработанных гипотез составляло 105, а 25 из них были усовершенствованы. Результирующая гипотеза `H`, приведенная выше, после перевода на обычный синтаксис Prolog принимает следующий вид:

```
has_daughter(A) :- parent(A,B), female(B).
```

Это соответствует нашему целевому предикату.

## Упражнение

**19.1.** Проведите эксперименты с программой MINIHYPHER, применяя модифицированные множества примеров, касающихся отношения `has_daughter`. Как эти модификации влияют на результаты?

Теперь рассмотрим немного более сложную задачу обучения: изучение отношения `predecessor` (предок) с использованием тех же фоновых знаний, которые приведены в листинге 19.1. Эта задача является более сложной потому, что для нее необходимо сформировать рекурсивное определение, но именно для этого и предназначен метод ILP. Мы можем определить некоторые положительные и отрицательные примеры следующим образом:

```
ex(predecessor(pam, bob)).
ex(predecessor(pam, jim)).
ex(predecessor(tom, ann)).
ex(predecessor(tom, jim)).
ex(predecessor(tom, liz)).
nex(predecessor(liz, bob)).
nex(predecessor(pat, bob)).
nex(predecessor(pam, liz)).
nex(predecessor(liz, jim)).
nex(predecessor(liz, liz)).
```

“Догадавшись”, что целевая гипотеза состоит из двух предложений (а фактически используя свои знания), можно определить начальную гипотезу следующим образом:

```
start_hyp([[predecessor(X1,Y1)] / [X1,Y1],
[predecessor(X2,Y2)] / [X2,Y2]]).
```

Соответствующие фоновые предикаты приведены ниже.

```
backliteral(parent(X,Y), [X,Y]).
backliteral(predecessor(X,Y), [X,Y]).
prolog_predicate(parent(X,Y)).
```

## Упражнение

**19.2.** Определите, сколько этапов усовершенствования требуется для получения целевой гипотезы из заданной выше начальной гипотезы.

Теперь можно попытаться вызвать на выполнение программу MINIHYPHER с этим определением задачи, но оказывается, что данная программа является для этого слишком неэффективной. Пространство поиска вплоть до требуемой глубины усовершенствования для нее слишком велико. Кроме того, процедура поиска повторно вырабатывает одинаковые гипотезы, достижимые с помощью разных путей усовершенствования. Это приводит к повторному формированию больших подпространств.

Доработка программы MINIHYPHER с целью создания программы HYPHER будет выполнена в следующем разделе. Тем не менее можно достаточно просто обеспечить успешное применение программы MINIHYPHER для решения задачи изучения отношения `predecessor` с помощью следующего приема. Для управления сложностью поиска (не без привлечения своей “догадки” о том, каким будет результат обучения) мы можем ввести такое ограничение, что фоновые литералы `parent(X,Y)` и `predecessor(X,Y)` должны вызываться с конкретизированным первым параметром `X`. Этого можно добиться, потребовав, чтобы `X` был атомом, т.е. применив следующее модифицированное определение фоновых знаний:

```
backliteral([atom(X), parent(X,Y)], [X,Y]).
backliteral([atom(X), predecessor(X,Y)], [X,Y]).
prolog_predicate(parent(X,Y)).
prolog_predicate(atom(X)).
```

Это означает, что при каждом усовершенствовании предложения к нему добавляется пара литералов и первым из них является литерал `atom(X)`. При проверке полноты такой гипотезы попытка достижения цели в теле предложения оканчивается

неудачей, если параметр  $X$  не конкретизирован (эта проверка осуществляется с помощью предиката  $\text{atom}(X)$ ). Таким образом, многие бесполезные гипотезы станут неполными и поэтому будут немедленно исключены из процесса поиска. Теперь задача поиска становится проще, хотя и может все еще потребовать много времени, которое составляет целые минуты или десятки минут, в зависимости от компьютера и реализации Prolog. В конечном итоге выполнение цели  $\text{induce}(H)$  приводит к получению следующего результата:

```
H = [[predecessor(A,B), [atom(A), parent(A,C)], [atom(C), predecessor(C,B)]]
 / [A,C,B], [predecessor(D,E), [atom(D), parent(D,E)]] / [D,E]]
```

Безусловно, что этот результат представляет собой искомое определение отношения  $\text{predecessor}$ .

Вполне очевидно, что программа MINIHYPHER вскоре окажется полностью непригодной, столкнувшись с более сложными задачами обучения. Поэтому в следующем разделе в нее будет введено несколько полезных дополнений.

## Упражнение

**19.3.** Вызовите на выполнение программу MINIHYPHER для решения задачи изучения предиката  $\text{predecessor}$  и измерьте время выполнения. Сколько гипотез выработано и сколько усовершенствовано? Описанный выше способ введения “защитного” литерала  $\text{atom}(X)$  в фоновые литералы является допустимым, но сам становится источником проблем. В тот момент, когда происходит добавление пары литералов наподобие  $[\text{atom}(X), \text{parent}(X, Y)]$ , процесс достижения всех целей, предшествующих им, оканчивается неудачей (поскольку  $X$  — это новая переменная, еще не согласованная с чем-либо иным). Поэтому обработка подобного литерала может быть выполнена успешно только после дальнейшего усовершенствования, при котором  $X$  будет согласована с существующей переменной. Таким образом, после усовершенствования по методу согласования такая гипотеза может стать более общей. В этом состоит отличие от обычного варианта, при котором в результате усовершенствований обрабатываются более конкретные гипотезы. Указанная аномалия в случае применения литерала  $\text{atom}(X)$  приводит к тому, что процесс усовершенствования становится немотонным: гипотезы вдоль пути усовершенствования не обязательно становятся все более и более конкретными. В том случае, когда неполная гипотеза может стать полной после усовершенствования, нарушается главная предпосылка, на которой основан поиск в пространстве гипотез! Но в данном конкретном случае поиск все еще осуществляется успешно. Найдите такую последовательность усовершенствований полных гипотез в графе усовершенствования, которая несмотря на эту аномалию приводит к целевой гипотезе.

## 19.2.4. Обобщение, уточнение и тэта-классификация

Как обычно принято в области машинного обучения, пространство возможных гипотез в методе ILP является частично упорядоченным с помощью отношений обобщения “более общий, чем” или “более конкретный, чем”. Гипотеза  $H_1$  является более общей, чем  $H_2$ , если  $H_1$  охватывает, по меньшей мере, все случаи, охваченные с помощью  $H_2$ . Применяемый оператор усовершенствования соответствует подобному отношению обобщения между гипотезами. Это отношение обобщения между гипотезами может быть определено синтаксически — усовершенствования представляют собой просто синтаксические операции над гипотезами.

В методе ILP часто используется еще одно отношение обобщения, называемое *тэта-классификацией*. Несмотря на то что в программах этой главы тэта-классификация непосредственно не применяется, рассмотрим здесь эту тему для полноты изложения.

Вначале определим понятие подстановки  $\theta$  (тэта). Подстановка  $\theta = \{\text{Var1}/\text{Term1}, \text{Var2}/\text{Term2}, \dots\}$  представляет собой отображение переменных  $\text{Var1}$ ,  $\text{Var2}$  и т.д. на термы  $\text{Term1}$ ,  $\text{Term2}$  и т.д. Подстановка  $\theta$  применяется к предложению  $C$  путем замены переменных предложения термами в соответствии с отображением, заданным в  $\theta$ . Результат применения подстановки  $\theta$  к предложению  $C$  записывается в виде  $C\theta$ , например, как показано ниже.

```
C = has_daughter(X) :- parent(X,Y), female(Y).
 $\theta = \{ X/\text{tom}, Y/\text{liz} \}$
 $C\theta = \text{has\_daughter}(\text{tom}) \text{ :- parent}(\text{tom},\text{liz}), \text{female}(\text{liz}).$
```

Теперь мы можем определить понятие *тэта-классификации*. Оно представляет собой отношение обобщения между предложениями. Предложение  $C_1$  тэта-классифицирует предложение  $C_2$ , если существует подстановка  $\theta$ , такая, что каждый литерал в  $C_1\theta$  встречается в  $C_2$ . Например, предложение

```
parent(X,Y).
```

может служить для тэта-классификации предложения

```
parent(X,liz).
```

где  $\theta = \{ Y/\text{liz} \}$ , а предложение

```
has_daughter(X) :- parent(X,Y).
```

может служить для тэта-классификации предложения

```
has_daughter(X) :- parent(X,Y), female(Y).
```

где  $\theta = \{ \}$ . Понятие тэта-классификации может стать основой способа синтаксической проверки существования отношения обобщения между предложениями. Если предложение  $C_1$  тэта-классифицирует предложение  $C_2$ , то  $C_2$  логически следует из  $C_1$ , поэтому  $C_1$  является более общим, чем  $C_2$ . Таким образом,  $C_1$  наряду с остальными гипотезами позволяет *истолковать* (или, как принято называть эту ситуацию, *охватить*), по меньшей мере, все примеры, охваченные предложением  $C_2$  и остальными гипотезами. Между применяемым оператором усовершенствования и тэта-классификацией существует простая связь. Оператор усовершенствования принимает предложение  $C_1$  и вырабатывает предложение  $C_2$ , такое, что  $C_1$  тэта-классифицирует  $C_2$ .

## Упражнение

19.4. Предположим, что задан следующий факт  $C_0$ :

```
num(0).
```

Допустим, что предложение  $C_1$  имеет форму

```
num(s(X)) :- num(X).
```

А предложение  $C_2$  — такую форму:

```
num(s(s(X))) :- num(X).
```

Является ли гипотеза  $\{C_0, C_1\}$  более общей, чем гипотеза  $\{C_0, C_2\}$ ? Обеспечивает ли предложение  $C_1$  тэта-классификацию предложения  $C_2$ ?

## 19.3. Программа HYPER

В данном разделе будут введены перечисленные ниже дополнения в программу ILP, приведенную в листинге 19.3.

1. Для предотвращения ненужных усовершенствований с помощью неприемлемых согласований переменные в предложениях должны быть типизированы. В усовершенствованиях предложений должно быть разрешено согласование только переменных одного и того же типа.

2. Метод обработки структурированных термов изменяется следующим образом: оператор усовершенствования позволяет также преобразовывать переменные в термы.
3. Проводится различие между входными и выходными параметрами литералов так, что оператор усовершенствования обеспечивает немедленную конкретизацию входных параметров.
4. Вместо поиска с итеративным углублением в графе усовершенствования применяется поиск по заданному критерию.

В следующих разделах указанные выше дополнения рассматриваются более подробно.

### 19.3.1. Оператор усовершенствования

Применяется такой же оператор усовершенствования, как и в программе MINHYPER, но дополненный с учетом типов параметров и различий между входными и выходными параметрами. Для ознакомления с некоторыми синтаксическими соглашениями рассмотрим определение задачи изучения предиката `member(X,L)` (листинг 19.4). В предложении

```
backliteral(member(X,L), [L:list], [X:item]).
```

указано, что `member(X,L)` — это фоновый литерал (допускающий рекурсивные вызовы), где `L` — входная переменная типа списка, а `X` — выходная переменная типа элемента. Общая форма определения фоновых литералов приведена ниже.

```
backliteral(Literal, InArgs, OutArgs)
```

В этом выражении `InArgs` и `OutArgs` представляют собой списки входных и выходных параметров:

```
InArgs = [In1:TypeI1, In2:TypeI2, ...]
```

```
OutArgs = [Out1:TypeO1, Out2: TypeO2, ...]
```

где `In1`, `In2`, ... — имена входных переменных, а `TypeI1`, `TypeI2`, ... — их типы. Аналогичным образом, список `OutArgs` определяет выходные переменные и их типы.

#### Листинг 19.4. Формулировка задачи для изучения предиката определения принадлежности к списку

```
% Формулировка задачи для изучения предиката member(X,L)

backliteral(member(X,L), [L:list], [X:item]). % Фоновый литерал

% Усовершенствование термов

term(list, [X|L], [X:item, L:list]).
term(list, [], []).

prolog_predicate(fail). % Фоновый литерал отсутствует в программе
 % на языке Prolog

start_clause([member(X,L)] / [X:item, L:list]).

% Положительные и отрицательные примеры

ex(member(a, [a])).
ex(member(b, [a,b])).
ex(member(d, [a,b,c,d,e])).

nex(member(b, [a])).
nex(member(d, [a,b])).
nex(member(f, [a,b,c,d,e])).
```



Назначение входных и выходных параметров состоит в следующем. Если какой-либо параметр литерала является входным, то предполагается, что он должен конкретизироваться при каждом выполнении этого литерала. Это означает, что если при усовершенствовании предложения такой литерал добавляется к телу предложения, то каждая из его входных переменных должна быть немедленно согласована с некоторой существующей переменной в предложении. В приведенном выше примере должна быть немедленно согласована с существующей переменной переменная *L* (также относящаяся к типу "список") при каждом добавлении литерала `member(X,L)` к предложению. Подобное согласование должно обеспечивать, чтобы входной параметр был конкретизирован ко времени выполнения данного литерала. С другой стороны, выходные параметры могут быть согласованы с другими переменными позднее. Поэтому при усовершенствовании предложения выходные переменные обрабатываются по такому же принципу, как и все переменные в программе MINIHYPHER. Но ограничения на типы исключают возможность согласования переменных разных типов. Поэтому переменная *X* типа *item* (элемент) не может быть согласована с переменной *L* типа *list* (список).

Возможные усовершенствования переменных с преобразованием в структурированные термы определяют следующим предикатом:

```
term(Type, Term, Vars)
```

Он указывает, что переменная типа *Type* в предложении может быть заменена термом *Term*; *Vars* — это список переменных и их типов, которые встречаются в терме *Term*. Поэтому в листинге 19.4 предложения

```
term(list, [X|L], [X:item, L:list]).
term(list, [], []).
```

указывают, что переменная типа *list* может быть преобразована в процессе усовершенствования в терм `[X|L]`, где *X* относится к типу *item*, а *L* — к типу *list*, или что эта переменная может быть заменена константой `[]` (без переменных). Во всей данной главе предполагается, что символ ":" введен как инфиксный оператор. В листинге 19.4 предложение

```
start_clause([member(X,L)] / [X:item, L:list]).
```

объявляет форму предложений в начальных гипотезах графа усовершенствования. Каждая начальная гипотеза представляет собой список начальных предложений, количество которых не может превышать некоторого максимального количества (копий). Список начальных гипотез формируется автоматически. Максимальное количество предложений в гипотезе определяется предикатом `max_clauses`. Он может быть задан пользователем соответствующим образом с учетом специфики задачи обучения.

Теперь определим оператор усовершенствования программы HYPHER в соответствии с приведенным выше описанием. Для усовершенствования любого предложения необходимо выполнить одно из перечисленных ниже действий.

1. Согласовать две переменные в предложении, например  $X_1 = X_2$ . Могут быть согласованы только переменные одного и того же типа.
2. Преобразовать в процессе усовершенствования переменную предложения в фоновый терм. При этом могут использоваться только термы, определенные предикатом `term/3`, а тип переменной и тип терма должны соответствовать друг другу.
3. Добавить фоновый литерал к предложению. Все входные параметры литерала должны быть согласованы (недетерминированно) с существующими переменными предложения (такого же типа).

Как и в программе MINIHYPHER, для усовершенствования гипотезы *H* в программе HYPHER выбирается одно из предложений *C*<sub>0</sub> в гипотезе *H*<sub>0</sub>, предложение *C*<sub>0</sub> преобразуется в процессе усовершенствования в предложение *C* и формируется новая гипотеза *H* путем замены предложения *C*<sub>0</sub> в гипотезе *H*<sub>0</sub> предложением *C*. В листин-

ге 19.5 показана последовательность усовершенствований при изучении предиката `member`.

**Листинг 19.5. Последовательность усовершенствований от начальной гипотезы до целевой гипотезы. Обратите внимание на четвертый шаг, в котором добавлен литерал и его входной параметр немедленно согласован с существующей переменной того же типа**

```
member(X1,L1).
member(X2,L2).
 ↓ Усовершенствовать терм L1 = [X3|L3]
member(X1,[X3|L3]).
member(X2,L2).
 ↓ Согласовать X1 = X3
member(X1,[X1|L3]).
member(X2,L2).
 ↓ Усовершенствовать терм L2 = [X4|L4]
member(X1,[X1|L3]).
member(X2,[X4|L4]).
 ↓ Ввести литерал member(X5,L5) и согласовать входные параметры L5 = L4
member(X1,[X1|L3]).
member(X2,[X4|L4]) :- member(X5,L4).
 ↓ Согласовать X2 = X5
member(X1,[X1|L3]).
member(X2,[X4|L4]) :- member(X2,L4).
```

В программе `HYPER` к этим операциям добавлены некоторые полезные эвристические функции, которые часто позволяют намного уменьшить вычислительную сложность. Первая эвристическая функция предусматривает, что если в гипотезе  $H_0$  имеется лишь одно предложение, которое охватывает отрицательный пример, то вырабатываются только усовершенствования, вытекающие из данного предложения. Причина этого состоит в том, что для получения совместимой гипотезы должно быть обязательно усовершенствовано именно такое предложение. Вторая эвристическая функция состоит в том, что отбрасываются “избыточные” предложения (которые содержат несколько копий одного и того же литерала). А третья эвристическая функция обеспечивает исключение *неудовлетворяемых предложений*. Предложение является неудовлетворяемым, если его тело невозможно логически вывести с помощью предиката `prove` из текущей гипотезы.

Такой оператор усовершенствования предназначен для выработки *наименее конкретных уточнений* (Least Specific Specialization — LSS). Уточнение  $H$  гипотезы  $H_0$  называется наименее конкретным, если не существует других уточнений гипотезы  $H_0$ , более общих, чем  $H$ . Но в действительности рассматриваемый оператор усовершенствования позволяет лишь приблизиться к LSS. Этот оператор усовершенствования позволяет достичь LSS только в условиях того ограничения, что количество предложений в гипотезе после усовершенствования остается тем же самым. Без этого ограничения оператор LSS должен был бы увеличивать количество предложений в гипотезе. Это может привести к созданию оператора усовершенствования, не совсем приемлемого с точки зрения практики из-за его вычислительной сложности, поскольку при его использовании количество предложений в усовершенствованной гипотезе может стать очень большим. Но ограничение, предусмотренное в данной программе, которое требует сохранения неизменного количества предложений в гипотезе после ее усовершенствования, является не слишком жестким. Если для решения требуется формирование гипотезы с большим количеством предложений, то такая гипотеза может быть выработана из другой начальной гипотезы, которая имеет достаточное количество предложений.

## 19.3.2. Поиск

Поиск начинается с множества начальных гипотез. Оно представляет собой множество всех возможных мультимножеств определяемых пользователем начальных предложений, вплоть до некоторого максимального количества предложений в гипотезе. Как правило, в начальной гипотезе появляется несколько копий одного и того же начального предложения. Типичное начальное предложение представляет собой нечто довольно общее и нейтральное, такое как `conc( L1, L2, L3)`. В процессе поиска граф усовершенствования рассматривается как дерево (если к одной и той же гипотезе ведет несколько путей, то в дереве появляется несколько копий этой гипотезы). Поиск начинается с нескольких начальных гипотез, которые становятся корнями несвязанных друг с другом деревьев поиска. Поэтому, строго говоря, пространство поиска представляет собой не дерево, а *лес*.

Программа **HYPER** выполняет поиск по заданному критерию с использованием функции оценки `Cost(Hypothesis)`, в которой учитывается размер гипотезы (как описано ниже), а также ее точность по отношению к заданным примерам. Стоимость гипотезы *H* определяется с помощью следующей простой формулы:

$$\text{Cost}(H) = w_1 * \text{Size}(H) + w_2 * \text{NegCover}(H)$$

где `NegCover(H)` — количество отрицательных примеров, охватываемых гипотезой *H*. Определение “гипотеза *H* охватывает пример *E*” трактуется в рамках осторожной интерпретации ответа `Answer` в предикате `prove`. В этой формуле  $w_1$  и  $w_2$  обозначают весовые коэффициенты. Размер гипотезы определяется как взвешенная сумма количества литералов и количества переменных в гипотезе следующим образом:

$$\text{Size}(H) = k_1 * \text{количество\_литералов}(H) + k_2 * \text{количество\_переменных}(H)$$

В коде программы **HYPER**, приведенном в этой главе, фактически используются следующие значения весовых коэффициентов:  $w_1 = 1$ ,  $w_2 = 10$ ,  $k_1 = 10$ ,  $k_2 = 1$ . Это соответствует приведенной ниже формуле.

$$\text{Cost}(H) = \text{количество\_переменных}(H) + 10 * \text{количество\_литералов}(H) + 10 * \text{NegCover}(H)$$

Эти значения весовых коэффициентов выбраны произвольным образом, но их относительные величины были интуитивно обоснованы на основании следующих рассуждений. При увеличении количества переменных в гипотезе вычислительная сложность ее обработки возрастает, поэтому количество переменных должно учитываться. Но вычислительная сложность в большей степени зависит от количества литералов, поэтому такое количество должно рассматриваться как составляющая стоимости с большим весовым коэффициентом. Любой охваченный отрицательный пример вносит такой же вклад в увеличение стоимости гипотезы, как и литерал. Это соответствует интуитивному пониманию того, что каждый дополнительный литерал должен исключать возможность охвата гипотезой, по меньшей мере, одного отрицательного примера. Проведенные эксперименты принесли довольно неожиданные результаты, согласно которым эти весовые коэффициенты можно изменять в значительных пределах без существенного воздействия на производительность поиска [18].

## 19.3.3. Программа HYPER

В листинге 19.6 показана реализация описанного выше проекта в виде программы **HYPER**. Основные предикаты этой программы описаны ниже.

### Листинг 19.6. Программа HYPER. Процедура `prove/3` приведена в листинге 19.2

```
% Программа HYPER (HYPothesis refInER) для решения задач обучения путем
% логического вывода

:- op(500, xfx, :).

% induce(Hyp):
```

```

% осуществляет логический вывод совместимой и полной гипотезы Нур путем
% постепенного усовершенствования начальных гипотез

induce(Hyp) :-
 init_counts, !, % Инициализировать счетчики гипотез
 start_hyps(Hyps), % Получить начальные гипотезы
 best_search(Hyps, _:Hyp). % Специализированный предикат поиска
 % по заданному критерию

% best_search(CandidateHyps, FinalHypothesis)

best_search([Hyp | Hyps], Hyp) :-
 show_counts, % Показать содержимое счетчиков гипотез
 Hyp = 0:H, % Стоимость Cost = 0; это означает, что гипотеза Н
 % не охватывает ни одного отрицательного примера
 complete(H). % Гипотеза Н охватывает все положительные примеры

best_search([C0:H0 | Hyps0], H) :-
 write('Refining hypo with cost '), write(C0),
 write(:), nl, show_hyp(H0), nl,
 all_refinements(H0, NewHs), % Все усовершенствования гипотезы H0
 add_hyps(NewHs, Hyps0, Hyps), !,
 addl(refined), % Подсчитать количество усовершенствованных гипотез
 best_search(Hyps, H).

all_refinements(H0, Hyps) :-
 findall(C:H,
 (refine_hyp(H0,H), % Н - новая гипотеза
 once((addl(generated), % Подсчитать количество выработанных
 complete(H), % гипотез
 complete(H), % Гипотеза Н охватывает все положительные
 % примеры
 addl(complete), % Подсчитать количество полных гипотез
 eval(H,C) % C - стоимость гипотезы Н
)),
 Hyps).

% add_hyps(Hyps1, Hyps2, Hyps):
% выполнить слияние гипотез Hyps1 и Hyps2 в порядке стоимостей
% и получить гипотезу Hyps

add_hyps(Hyps1, Hyps2, Hyps) :-
 mergesort(Hyps1, OrderedHyps1),
 merge(Hyps2, OrderedHyps1, Hyps).

complete(Hyp) :- % Нур охватывает все положительные примеры
 not (ex(P), % Положительный пример
 once(prove(P, Hyp, Answ)), % Доказать его с помощью гипотезы Нур
 Answ \== yes). % Возможно, доказательство не найдено

% eval(Hypothesis, Cost):
% стоимость гипотезы Cost = Size + 10
% * количество_охваченных_отрицательных_примеров,
% где Size - размер

eval(Hyp, Cost) :-
 size(Hyp, S), % Размер гипотезы
 covers_neg(Hyp, N), % Количество охваченных отрицательных примеров
 (N = 0, !, Cost is 0; % Охваченных отрицательных примеров нет
 Cost is S + 10*N).

% size(Hyp, Size):
% размер Size = k1 * количество_литералов + k2
% * количество_переменных_в_гипотезе;
% значения коэффициентов: k1=10, k2=1

```

```

size([], 0).

size([Cs0/Vs0 | RestHyp], Size) :-
 length(Cs0, L0),
 length(Vs0, N0),
 size(RestHyp, SizeRest),
 Size is 10*L0 + N0 + SizeRest.

% covers_neg(H, N):
% N - количество отрицательных примеров, которые, возможно, охвачены
% гипотезой H. Возможность того, что примеры охвачены гипотезой,
% существует, если предикат prove/3 возвращает значение 'yes' или 'maybe'

covers_neg(Hyp, N) :- % Гипотеза Hyp охватывает N отрицательных примеров
 findall(1, (nex(E), once(prove(E,Hyp,Answ)), Answ \== no), L),
 length(L, N).

% unsatisfiable(Clause, Hyp):
% предложение Clause нельзя ни при каких условиях использовать в каком-либо
% доказательстве; это означает, что тело предложения Clause не может быть
% доказано на основании гипотезы Hyp

unsatisfiable([Head | Body], Hyp) :-
 once(prove(Body, Hyp, Answ)), Answ = no.

start_hyps(Hyps) :- % Множество начальных гипотез
 max_clauses(M),
 setof(C:H,
 (start_hyp(H,M), add1(generated),
 complete(H), add1(complete), eval(H,C)),
 Hyps).

% start_hyp(Hyp, MaxClauses):
% Hyp - начальная гипотеза, количество предложений в которой
% не превышает MaxClauses

start_hyp([], _).

start_hyp([C | Cs], M) :-
 M > 0, M1 is M-1,
 start_clause(C), % Определяемое пользователем начальное предложение
 start_hyp(Cs, M1).

% refine_hyp(Hyp0, Hyp):
% предикат, преобразующий гипотезу Hyp0 в гипотезу Hyp
% путем усовершенствования

refine_hyp(Hyp0, Hyp) :-
 choose_clause(Hyp0, Clause0/Vars0, Clauses1, Clauses2), % Выбрать предложение
 conc(Clauses1, [Clause0/Vars | Clauses2], Hyp), % Новая гипотеза
 refine(Clause0, Vars0, Clause, Vars), % Усовершенствовать выбранное
 % предложение
 non_redundant(Clause), % Предложение Clause не является избыточным
 not unsatisfiable(Clause, Hyp). % Предложение Clause не является
 % удовлетворяемым

choose_clause(Hyp, Clause, Clauses1, Clauses2) :- % Предикат, который выбирает
 % предложение Clause из состава гипотезы Hyp
 conc(Clauses1, [Clause | Clauses2], Hyp), % Выбрать предложение
 nex(E), % Отрицательный пример E
 prove(E, [Clause], yes), % Пример E охватывает само предложение Clause
 !, % Предложение должно быть усовершенствовано
 ;
 conc(Clauses1, [Clause | Clauses2], Hyp). % В противном случае выбрать
 % любое предложение

```

```

% refine(Clause, Args, NewClause, NewArgs):
% предикат, который позволяет усовершенствовать предложение Clause
% с параметрами Args и получить предложение NewClause с параметрами NewArgs

% Усовершенствовать по методу согласования параметров

refine(Clause, Args, Clause, NewArgs) :-
 conc(Args1, [A | Args2], Args), % Выбрать переменную A
 member(A, Args2), % Согласовать ее с другой переменной
 conc(Args1, Args2, NewArgs).

% Усовершенствовать переменную по методу преобразования в терм

refine(Clause, Args0, Clause, Args) :-
 del(Var:Type, Args0, Args1), % Удалить переменную Var:Type
 % из множества параметров Args0
 term(Type, Var, Vars), % Переменная Var становится термом типа Type
 conc(Args1, Vars, Args). % Ввести переменные в новый терм

% Усовершенствовать предложение путем добавления литерала

refine(Clause, Args, NewClause, NewArgs) :-
 length(Clause, L),
 max_clause_length(MaxL),
 L < MaxL,
 backliteral(Lit, InArgs, RestArgs), % Литерал с определением фоновых знаний
 conc(Clause, [Lit], NewClause), % Добавить литерал к телу предложения
 connect_inputs(Args, InArgs), % Соединить входные параметры литерала
 % с другими параметрами
 conc(Args, RestArgs, NewArgs). % Добавить остальные параметры литерала

% non_redundant(Clause): очевидно, что предложение Clause
% не имеет избыточных литералов

non_redundant([]). % Предложение с одним литералом

non_redundant([Lit1 | Lits]) :-
 not literal_member(Lit1, Lits),
 non_redundant(Lits).

literal_member(X, [X1 | Xs]) :- % Переменная X в полном смысле слова
 % равна элементу списка
 X == X1, !
;
 literal_member(X, Xs).

% show_hyp(Hypothesis):
% предикат, который выводит гипотезу Hypothesis в удобной для чтения форме,
% с именами переменных A, B, ...

show_hyp([]) :- nl.

show_hyp([C/Vars | Cs]) :- nl,
 copy_term(C/Vars, C1/Vars1),
 name_vars(Vars1, ['A','B','C','D','E','F','G','H','I','J','K','L','M','N']),
 show_clause(C1),
 show_hyp(Cs), !.

show_clause([Head | Body]) :-
 write(Head),
 (Body = [] ; write(':-'), nl),
 write_body(Body).

write_body([]) :-
 write('.'), !.

```

```

write_body([G | Gs]) :- !,
 tab(2), write(G),
 (Gs = [], !, write('.') , nl
 ;
 write(',') , nl,
 write_body(Gs)
).

name_vars([], _).

name_vars([Name:Type | Xs], [Name | Names]) :-
 name_vars(Xs, Names).

% connect_inputs(Vars, Inputs):
% предикат, который согласует каждую переменную в списке Inputs с переменной
% в списке Vars

connect_inputs(_, []).

connect_inputs(S, [X | Xs]) :-
 member(X, S),
 connect_inputs(S, Xs).

% merge(L1, L2, L3): предикат слияния, в котором все параметры,
% заданные в виде списков, отсортированы

merge([], L, L) :- !.

merge(L, [], L) :- !.

merge([X1|L1], [X2|L2], [X1|L3]) :-
 X1 @=< X2, !, % X1 "лексикографически предшествует" X2 (встроенный предикат)
 merge(L1, [X2|L2], L3).

merge(L1, [X2|L2], [X2|L3]) :-
 merge(L1, L2, L3).

% mergesort(L1, L2): предикат, который сортирует список L1, формируя список L2

mergesort([], []) :- !.

mergesort([X], [X]) :- !.

mergesort(L, S) :-
 split(L, L1, L2),
 mergesort(L1, S1),
 mergesort(L2, S2),
 merge(S1, S2, S).

% split(L, L1, L2): предикат, который разбивает список L на два списка
% примерно равной длины

split([], [], []).

split([X], [X], []).

split([X1,X2 | L], [X1|L1], [X2|L2]) :-
 split(L, L1, L2).

% Счетчики сформированных, полных и усовершенствованных гипотез

init_counts :-
 retract(counter(_, _)), fail % Удалить прежние значения счетчиков
 ;
 assert(counter(generated, 0)), % Инициализировать счетчик сформированных
 % гипотез

```

```

assert(counter(complete, 0)), % Инициализировать счетчик полных гипотез
assert(counter(refined, 0)). % Инициализировать счетчик
 % усовершенствованных гипотез

add1(Counter) :-
 retract(counter(Counter, N)), !, N1 is N+1,
 assert(counter(Counter, N1)).

show_counts :-
 counter(generated, NG), counter(refined, NR), counter(complete, NC),
 nl, write('Hypotheses generated: '), write(NG),
 nl, write('Hypotheses refined: '), write(NR),
 ToBeRefined is NC - NR,
 nl, write('To be refined: '), write(ToBeRefined), nl.

% Значения параметров

max_proof_length(6). % Максимальная длина доказательства с учетом
 % вызовов предикатов в гипотезах
max_clauses(4). % Максимальное количество предложений в гипотезах
max_clause_length(5). % Максимальное количество литералов в теле предложении

```

- **refine\_hyp( Hyp0, Hyp).** Предикат, который в процессе усовершенствования преобразует гипотезу Hyp0 в Hyp, усовершенствовав одно из предложений в Hyp0.
- **refine( Clause, Vars, NewClause, NewVars).** Предикат, преобразующий в процессе усовершенствования данное предложение Clause с переменными Vars и вырабатывающий усовершенствованное предложение NewClause с переменными NewVars. Это усовершенствованное предложение формируется путем согласования двух переменных одного и того же типа в списке Vars, или путем усовершенствования переменной в списке Vars с преобразованием в терм, или путем добавления нового фонового литерала к предложению Clause.
- **induce\_hyp( Hyp).** Предикат, который логически выводит совместимую и полную гипотезу Hyp для данной задачи обучения. В этом предикате осуществляется поиск по заданному критерию путем вызова предиката `best_search/2`.
- **best\_search( Hyps, Hyp).** Процедура, которая начинает работу с множества начальных гипотез Hyps, выработанных предикатом `start_hyps/1`, и выполняет поиск по заданному критерию в лесу усовершенствования до тех пор, пока не будет найдена совместимая и полная гипотеза Hyp. В этой процедуре для управления поиском в качестве функции оценки используется стоимость гипотез. Каждая рассматриваемая гипотеза применяется в сочетании с ее стоимостью в виде термина `Cost: Hypothesis`. Во время сортировки списка таких термов (по методу сортировки и слияния) гипотезы сортируются по возрастанию стоимости.
- **prove( Goal, Hyp, Answer).** Интерпретатор с ограничением длины доказательства, который определен в листинге 19.2.
- **eval( Hyp, Cost).** Функция оценки гипотез. В параметре Cost учитываются размер гипотезы Hyp и количество отрицательных примеров, охваченных гипотезой Hyp. Если Hyp не охватывает ни одного отрицательного примера, то `Cost = 0`.
- **start\_hyps( Hyps).** Предикат, который вырабатывает множество Hyps начальных гипотез для поиска. Каждая начальная гипотеза представляет собой список, состоящий из начальных предложений в количестве вплоть до `MaxClauses`. Значение `MaxClauses` определяется пользователем с помощью



предиката `max_clauses`. Начальные предложения определяются пользователем с использованием предиката `start_clause`.

- `show_hyp( Hyp)`. Предикат, отображающий гипотезу `Hyp` в обычном формате Prolog.
- `init_counts, show_counts, add1(Counter)`. Предикаты, которые инициализируют, отображают и обновляют счетчики гипотез. Отдельно подсчитываются гипотезы трех типов: выработанные (количество всех выработанных гипотез), полные (количество выработанных гипотез, которые охватывают все положительные примеры) и усовершенствованные (количество всех усовершенствованных гипотез).
- `start_clause( Clause)`. Определяемые пользователем начальные предложения (наподобие следующего), которые обычно представляют собой нечто очень общее:  

```
start_clause([member(X,L)] / [X:item, L:list]).
```
- `max_proof_length(D), max_clauses(MaxClauses), max_clause_length(MaxLength)`. Предикаты, определяющие следующие параметры: максимальная длина доказательства, максимальное количество предложений в гипотезе и максимальное количество литералов в предложении. Например, при изучении предиката `member/2` или `conc/3` достаточно задать `MaxClauses=2`. По умолчанию в программе, приведенной в листинге 19.6, заданы следующие значения:  

```
max_proof_length(6). max_clauses(4). max_clause_length(5).
```

Для программы в листинге 19.6 требуются также такие часто используемые предикаты, как `not/1`, `once/1`, `member/2`, `conc/3`, `del/3`, `length/2`, `copy_term/2`.

В качестве иллюстрации вызовем на выполнение программу `HYPER` для решения задачи изучения предиката `member(X,L)`. Кроме самой программы `HYPER`, в систему Prolog необходимо загрузить определение задачи, приведенное в листинге 19.4. После этого программе можно задать следующий вопрос:

```
?- induce(H), show_hyp(H).
```

Во время выполнения программа `HYPER` последовательно отображает текущее количество гипотез (выработанных, усовершенствованных и ожидающих усовершенствования), а также показывает гипотезу, которая в настоящее время проходит процесс усовершенствования. Эта программа вырабатывает следующие окончательные результаты:

```
Hypotheses generated: 105 % Количество выработанных гипотез
Hypotheses refined: 26 % Количество усовершенствованных гипотез
To be refined: 15 % Количество гипотез, подлежащих усовершенствованию
```

```
member(A, [A|B]).
member(C, [A|B]) :-
member(C, B).
```

Логически выведена именно такая гипотеза, как и следовало ожидать. До того как была найдена эта гипотеза, всего было выработано 105 гипотез, 26 из них были усовершенствованы, а 15 все еще оставались в списке кандидатов на усовершенствование. Разность  $105 - 26 - 15 = 51$  показывает, сколько найденных неполных гипотез было немедленно отброшено. Необходимая глубина усовершенствования для этой задачи обучения равна 5 (см. листинг 19.5). Общее количество возможных гипотез в пространстве усовершенствования, которое определено с помощью данного (ограниченного) оператора усовершенствования программы `HYPER`, составляет несколько тысяч. Программа `HYPER` выполнила поиск только в очень ограниченной части этого пространства (составляющей меньше 10%). Эксперименты показали, что при решении более сложных задач обучения (конкатенация списков, поиск маршрутов) это соотношение становится еще меньше.

## Упражнение

19.5. Определите задачу обучения для изучения предиката конкатенации списков `conc(L1,L2,L3)` в соответствии с соглашениями, показанными в листинге 19.4, и вызовите на выполнение программу **HYPER** с этим определением. Определите глубину усвоения целевой гипотезы и оцените размер дерева усвоения для этой глубины при начальной гипотезе, состоящей из двух предложений. Сравните этот размер с количеством гипотез, выработанных и усовершенствованных программой **HYPER**.

### 19.3.4. Эксперименты с программой **HYPER**

Поскольку в программе **HYPER** применяются ограничения процесса усвоения и эвристический поиск, она является намного более эффективной по сравнению с **MINIHYPER**. Но и программа **HYPER** сталкивается с затруднениями при решении многих логических задач обучения, которые обычно характеризуются высокой вычислительной сложностью. Было бы интересно узнать, где проходит граница между задачами обучения, которые могут и не могут быть решены с помощью программы **HYPER**. Но отметим, что эту границу можно намного расширить за счет творческой постановки задачи обучения (умелого использования фоновых знаний, входных и выходных переменных в фоновых литералах, множеств примеров). В этом разделе рассматриваются некоторые весьма наглядные примеры решения задач обучения с помощью программы **HYPER**.

#### Одновременное изучение предикатов `odd(L)` и `even(L)`

Программа **HYPER** может применяться без дополнительной обработки для *мультипредикатного обучения* (multi-predicate learning). Так называется одновременное изучение нескольких предикатов, при котором один предикат может быть определен в терминах другого. При этом может даже использоваться взаимная рекурсия, когда изучаемые предикаты вызывают друг друга. В этом разделе такой вариант будет продемонстрирован на примере изучения предикатов `odd(List)` и `even(List)` (которые принимают истинное значение при обработке списков нечетной и четной длины). Определение этой задачи обучения приведено в листинге 19.7, а результат обучения приведен ниже.

```
Hypotheses generated: 85
Hypotheses refined: 16
To be refined: 29
```

```
even({}).
even([A,B|C]) :-
 even(C).
```

```
odd([A|B]) :-
 even(B).
```

#### Листинг 19.7. Одновременное изучение предикатов, позволяющих различать списки нечетной и четной длины

```
% Логический вывод предикатов, позволяющих различать списки
% нечетной (odd) и четной (even) длины
```

```
backliteral(even(L), [L:list], []).
backliteral(odd(L), [L:list], []).
```

```
term(list, [X|L], [X:item, L:list]).
term(list, [], []).
```

```
prolog_predicate(fail).
```

```

start_clause([odd(L)] / [L:list]).
start_clause([even(L)] / [L:list]).

ex(even([])).
ex(even([a,b])).
ex(odd([a])).
ex(odd([b,c,d])).
ex(odd([a,b,c,d,e])).
ex(even([a,b,c,d])).

nex(even([a])).
nex(even([a,b,c])).
nex(odd([])).
nex(odd([a,b])).
nex(odd([a,b,c,d])).

```

---

Полученный результат соответствует целевому понятию. Но программа HYPER нашла определение, которое не является взаимно рекурсивным. Достаточно только потребовать выработки еще одного решения (введя, как обычно, точку с запятой), и программа HYPER продолжает поиск, после чего находит следующее взаимно рекурсивное определение:

```

Hypotheses generated: 115
Hypotheses refined: 26
To be refined: 32
even([]).
odd([A|B]) :-
 even(B).
even([A|B]) :-
 odd(B).

```

Выработку первого, не взаимно рекурсивного определения, можно предотвратить с помощью более ограничительной регламентации процедуры усовершенствования термов. Такое более ограничительное определение должно было допускать усовершенствование списков только на глубину 1. Этого можно достичь, заменяя тип `list` типом `list(D)` и применяя вместо первого предложения, касающегося усовершенствования списков, следующее предложение:

```

term(list(D), [X|L], [X:item, L:list(1)]) :- var(D).

```

Это определение исключает возможность продолжать дальнейшее усовершенствование переменной типа `list(1)`. Поэтому в программе больше не могут вырабатываться термы наподобие `[X,Y|L]`. Безусловно, существует возможность модифицировать соответствующим образом другие предложения, касающиеся предикатов `term` и `start_clause`.

## Изучение предиката `path( StartNode, GoalNode, Path )`

В листинге 19.8 показана область определения для изучения предиката `path` в ориентированном графе (который задан с помощью фонового предиката `link/2`). Процесс обучения проходит бесперебойно и приводит к следующим результатам:

```

Hypotheses generated: 401
Hypotheses refined: 35
To be refined: 109

path(A,A,[A]).
path(A,C,[A,B|E]) :-
 link(A,B),
 path(B,C,[D|E]).

```

## Листинг 19.8. Изучение предиката `path`, предназначенного для поиска пути в графе

```
% Изучение предиката path(StartNode,GoalNode,Path), предназначенного
% для поиска пути в графе

% Ориентированный граф

link(a,b).
link(a,c).
link(b,c).
link(b,d).
link(d,e).

backliteral(link(X,Y), [X:item], [Y:item]).
backliteral(path(X,Y,L), [X:item], [Y:item, L:list]).

term(list, [X|L], [X:item, L:list]).
term(list, [], []).

prolog_predicate(link(X,Y)).

start_clause([path(X,Y,L)] / [X:item,Y:item,L:list]).

% Примеры

ex(path(a, a, [a])).
ex(path(b, b, [b])).
ex(path(e, e, [e])).
ex(path(f, f, [f])).
ex(path(a, c, [a,c])).
ex(path(b, e, [b,d,e])).
ex(path(a, e, [a,b,d,e])).

nex(path(a, a, [])).
nex(path(a, a, [b])).
nex(path(a, a, [b,b])).
nex(path(e, d, [e,d])).
nex(path(a, d, [a,b,c])).
nex(path(a, c, [a])).
nex(path(a, c, [a,c,a,c])).
nex(path(a, d, [a,d])).
```

Последняя строка в логически выведенном определении может на первый взгляд показаться неожиданной, но в данном контексте она фактически эквивалентна ожидаемому значению `path(B,C,[B|E])`. Для получения последнего варианта требуется, чтобы количество этапов усовершенствования было на единицу меньше. Тот факт, что в этом процессе поиска было усовершенствовано только 35 гипотез, может показаться довольно удивительным, если принять во внимание следующие факты. Глубина усовершенствования приведенной выше гипотезы `path`, обнаруженной в процессе поиска, равна 12, а результаты оценки показывают, что размер дерева усовершенствования на этой глубине превышает  $10^{17}$  гипотез! Но фактически выполнен поиск лишь в очень небольшой части этого пространства. Такие результаты можно объяснить тем, что в данном случае требования к полноте гипотезы ограничивают поиск особенно эффективно.

## Изучение предиката, предназначенного для сортировки по методу вставки

Определение этой задачи обучения приведено в листинге 19.9. Данное определение нельзя назвать бесспорным, поскольку в нем фоновые знания весьма конкретно направлены на логический вывод процедуры сортировки по методу вставки. Направляется замечание, что при таком определении фоновых знаний мы практически уже были обязаны знать решение. Но такая постанова может служить иллюстраци-

ей типичной постановки задачи в области машинного обучения. Для того чтобы обучение было наиболее эффективным, мы должны передать программе обучения настолько качественное представление задачи, насколько это возможно, включая фоновые знания. Это неизбежно требует от пользователя размышлений по поводу возможных решений. В данном случае, в котором рассматривается поиск алгоритма сортировки, задача обучения была бы очень сложной без подобного полезного определения фоновых знаний. Но даже в этом случае оказывается, что рассматриваемая задача является наиболее сложной среди всех проведенных до сих пор экспериментов. Код, показанный в листинге 19.9, требует дополнительных комментариев. Дело в том, что в терме `sort(L1,L2)` параметр `L1`, согласно принятому предположению, должен быть конкретизирован, тогда как `L2` — это выходной параметр, который конкретизируется после вызова на выполнение процедуры `sort`. Для того чтобы логически выведенная процедура `sort` могла действовать при неконкретизированном параметре `L2`, один из примеров задан следующим образом:

```
ex([sort([c,a,b], L), L = [a,b,c]]).
```

### Листинг 19.9. Изучение процедуры сортировки по методу вставки

% Изучение процедуры сортировки

```
backliteral(sort(L, S), [L:list], [S:list]).
backliteral(insert_sorted(X, L1, L2), [X:item, L1:list], [L2:list]).
```

```
term(list, [X | L], [X:item, L:list]).
term(list, [], []).
```

```
prolog_predicate(insert_sorted(X, L0, L)).
prolog_predicate(X=Y).
```

```
start_clause([sort(L1,L2)] / [L1:list, L2:list]).
```

```
ex(sort([], [])).
ex(sort([a], [a])).
ex([sort([c,a,b], L), L = [a,b,c]]). % Второй параметр процедуры sort
 % неконкретизирован!
```

```
ex(sort([b,a,c], [a,b,c])).
ex(sort([c,d,b,e,a], [a,b,c,d,e])).
ex(sort([a,d,c,b], [a,b,c,d])).
```

```
nex(sort([], [a])).
nex(sort([a,b], [a])).
nex(sort([a,c], [b,c])).
nex(sort([b,a,d,c], [b,a,d,c])).
nex(sort([a,c,b], [a,c,b])).
nex(sort([], [b,c,d])).
```

```
insert_sorted(X, L, _) :- % "Защитное" предложение: проверка конкретизации
 % параметров
 var(X), !, fail
 ;
 var(L), !, fail
 ;
 L = [Y|_], var(Y), !, fail.
```

```
insert_sorted(X, [], [X]) :- !.
```

```
insert_sorted(X, [Y | L], [X,Y | L]) :-
 X @< Y, !. % Терм X "лексикографически предшествует" терму Y
```

```
insert_sorted(X, [Y | L], [Y | L1]) :-
 insert_sorted(X, L, L1).
```

Это позволяет обеспечить возможность вызова процедуры `sort` с неконкретизированным значением `L`, так, чтобы `sort` сформировала (а не просто распознала!) результат `L` и только после этого проверила его правильность. Необходимо также соблюдать осторожность при определении фонового предиката `insert_sorted(X, L1, L2)`, чтобы можно было обеспечить конкретизацию параметров в соответствии с ожидаемым (например, `X` не является переменной). Полученные результаты приведены ниже.

Hypotheses generated: 3708

Hypotheses refined: 284

To be refined: 448

```
sort([], []).
sort([A|B], D) :-
 sort(B, C),
 insert_sorted(A, C, D).
```

## Изучение предиката, позволяющего распознать конструкцию арки

Эта задача обучения аналогична задаче реляционного обучения, описанной в главе 18, в которой в качестве примеров применяются конструкции, выполненные из блоков (рис. 19.2), и существует иерархия между типами блоков (которая определяется предикатом `ako(X, Y)` — `Y` представляет собой разновидность (`ako` — a kind of) `X`; это отношение аналогично показанному на рис. 18.6). Примеры описывают конструкции, состоящие из трех блоков. Первые два блока из трех определяют стойки арки, а третий блок определяет перекрытие. Поэтому одним из положительных примеров является `arch(a1, b1, c1)`, где `a1`, `b1` и `c1` — прямоугольные блоки, `c1` опирается на `a1` и `b1`, а между `a1` и `b1` не определено отношение `touch` (соприкасается). Блоки `a2`, `b2` и `c2` образуют отрицательный пример (`nex(a2, b2, c2)`), поскольку блок `c2` не опирается на `a2` и `b2`. Блоки `a5`, `b5` и `c5` образуют еще один отрицательный пример, поскольку `c5` не соответствует определению “устойчивого многоугольного блока”. Определение задачи обучения распознаванию арки приведено в листинге 19.10. Фоновые предикаты в этом листинге включают также отрицание (`not`), которое применено к предикатам `touch/2` и `support/2`. Такой фоновый предикат, как обычно, выполняется непосредственно интерпретатором Prolog по принципу отрицания как недостижения цели. В отличие от рис. 18.4, на рис. 19.2 количество отрицательных примеров увеличено для ограничения возможности выбора совместимых гипотез. Результат обучения приведен ниже.

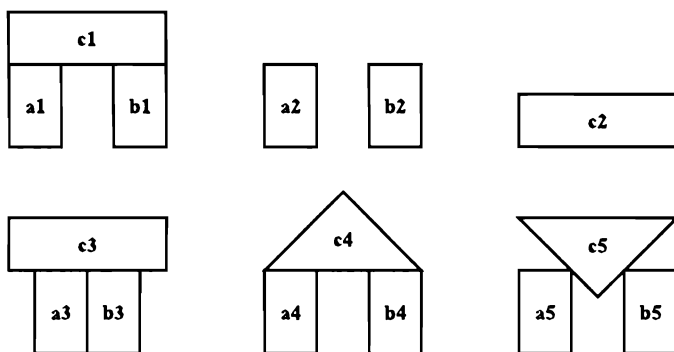


Рис. 19.2. Мир блоков с двумя положительными и тремя отрицательными примерами арки. Блоки `a1`, `b1` и `c1` показывают один положительный пример, а блоки `a4`, `b4` и `c4` — второй. Один из отрицательных примеров показывают блоки `a2`, `b2` и `c2`

Hypotheses generated: 368  
Hypotheses refined: 10  
To be refined: 151

```
arch(A,B,C) :- % Арка состоит из стоек A, B и перекладины C
 support(B,C), % C опирается на B
 not touch(A,B), % A и B не соприкасаются
 support(A,C), % C опирается на A
 isa(C,stable_poly). % C - устойчивый многоугольный блок
```

## Листинг 19.10. Изучение понятия арки

% Изучение понятия арки

```
backliteral(isa(X,Y), [X:object], []) :-
 member(Y, [polygon,convex_poly,stable_poly,unstable_poly,triangle,
 rectangle, trapezium, unstable_triangle, hexagon]). % Y - это блок
 % с любой из указанных форм

backliteral(support(X,Y), [X:object, Y:object], []).
backliteral(touch(X,Y), [X:object, Y:object], []).
backliteral(not G, [X:object,Y:object], []) :-
 G = touch(X,Y); G = support(X,Y).

prolog_predicate(isa(X,Y)).
prolog_predicate(support(X,Y)).
prolog_predicate(touch(X,Y)).
prolog_predicate(not G).

ako(polygon, convex_poly). % Блок выпуклой многоугольной формы
 % представляет собой одну из разновидностей блоков многоугольной формы
ako(convex_poly, stable_poly). % Блок устойчивой формы представляет
 % собой одну из разновидностей блоков выпуклой многоугольной формы
ako(convex_poly, unstable_poly). % Блок неустойчивой формы представляет
 % собой одну из разновидностей блоков выпуклой многоугольной формы
ako(stable_poly, triangle). % Блок треугольной формы представляет
 % собой одну из разновидностей блоков устойчивой формы
ako(stable_poly, rectangle). % Блок прямоугольной формы представляет
 % собой одну из разновидностей блоков устойчивой формы
ako(stable_poly, trapezium). % Блок трапециевидной формы представляет
 % собой одну из разновидностей блоков устойчивой формы
ako(unstable_poly, unstable_triangle). % Блок неустойчивой треугольной формы
 % представляет собой одну из разновидностей блоков неустойчивой формы
ako(unstable_poly, hexagon). % Блок шестиугольной формы представляет
 % собой одну из разновидностей блоков неустойчивой формы

ako(rectangle, X) :-
 member(X, [a1,a2,a3,a4,a5,b1,b2,b3,b4,b5,c1,c2,c3]). % Прямоугольные блоки

ako(triangle, c4). % Блок устойчивой треугольной формы
ako(unstable_triangle, c5). % Блок треугольной формы, перевернутый
 % основанием вверх

isa(Figure1, Figure2) :- % Блок Figure1 представляет собой блок Figure2
 ako(Figure2, Figure1).

isa(Fig0, Fig) :-
 ako(Fig1, Fig0),
 isa(Fig1, Fig).

support(a1,c1). support(b1,c1).
support(a3,c3). support(b3,c3). touch(a3,b3).
support(a4,c4). support(b4,c4).
support(a5,c5). support(b5,c5).

start_clause([arch(X,Y,Z)] / [X:object,Y:object,Z:object]).
```

```
ex(arch(a1,b1,c1)).
ex(arch(a4,b4,c4)).

nex(arch(a2,b2,c2)).
nex(arch(a3,b3,c3)).
nex(arch(a5,b5,c5)).
nex(arch(a1,b2,c1)).
nex(arch(a2,b1,c1)).
```

---

## Резюме

- *Индуктивное логическое программирование (ILP)* представляет собой сочетание логического программирования и машинного обучения.
- ILP — это метод *индуктивного обучения*, в котором в качестве языка гипотез используется логика предикатов. Кроме того, ILP представляет собой один из подходов к осуществлению автоматического программирования на примерах.
- По сравнению с другими подходами к машинному обучению *метод ILP* характеризуется следующими особенностями: во-первых, в нем используется более выразительный язык гипотез, который позволяет формировать рекурсивные определения гипотез, во-вторых, он обеспечивает применение фоновых знаний в более общей форме, и, в-третьих, в целом характеризуется большей комбинаторной сложностью по сравнению с обучением на основе атрибутов и значений.
- В *графе усовершенствования* предложений узлы соответствуют логическим предложениям, а дуги — отношениям усовершенствования между предложениями.
- В графе усовершенствования гипотез *узлы* соответствуют множествам логических предложений (программам Prolog), а *дуги* — отношениям усовершенствования между гипотезами.
- *Усовершенствование предложения* (гипотезы) приводит к более определенному предложению (гипотезе).
- Предложение может быть усовершенствовано с помощью следующих *операций*: во-первых, путем согласования двух переменных в предложении, или, во-вторых, в результате замены переменной термом, или, в-третьих, с помощью добавления литерала к телу предложения.
- *Тэта-классификация* — это отношение обобщения между предложениями, которое может быть определено синтаксически на основе замены переменных.
- *Программа HYPER*, разработанная в этой главе, осуществляет логический вывод программ Prolog на основании примеров, выполняя поиск среди гипотез в графе усовершенствования.
- В настоящей главе рассматривались следующие понятия:
  - индуктивное логическое программирование;
  - усовершенствование предложения;
  - усовершенствование гипотезы;
  - графы усовершенствования предложений или гипотез;
  - тэта-классификация;
  - автоматическое программирование на примерах.



## Дополнительные источники информации

Термин *индуктивное логическое программирование* был впервые введен Стефеном Магглтоном в [110]. К ранним работам в этой области, опубликованным еще до того, как она получила свое современное название, относятся [125], [136] и [146]. Программа HYPER, приведенная в этой главе, основана на программе, опубликованной в [18]. Подробное введение в проблематику ILP можно найти в [86]. В [42] и [112] опубликованы сборники статей по проблеме ILP. К числу наиболее широко известных систем ILP относятся Progol [111] и FOIL [129]. В [21] приведен обзор большого количества приложений в области ILP.

# Качественные рассуждения

### *В этой главе...*

|                                                                                    |     |
|------------------------------------------------------------------------------------|-----|
| 20.1. Здравый смысл, качественные рассуждения и обыденные физические представления | 478 |
| 20.2. Качественные рассуждения о статических системах                              | 482 |
| 20.3. Качественные рассуждения о динамических системах                             | 486 |
| 20.4. Программа качественного машинного моделирования                              | 493 |
| 20.5. Описание программы качественного машинного моделирования                     | 502 |

Традиционные методы разработки количественных теоретических и машинных моделей приводят к получению точных числовых ответов. Но для повседневного использования такие ответы часто бывают слишком подробными. Например, при заполнении ванны достаточно помнить, что если приток воды не будет вовремя остановлен, то вода постепенно достигнет края ванны. Чтобы вовремя перекрыть воду, не нужно знать с точностью до миллиметра уровень воды в тот или иной момент времени. Общепринятое описание процесса заполнения ванны является качественным, а не количественным: “уровень воды будет постоянно повышаться и в конечном итоге достигнет края ванны, что вызовет ее переполнение...”. В этом высказывании просто дана полезная краткая сводка количественной информации, которая, в отличие от качественной, может иметь весьма значительный объем. *Качественные рассуждения* — это область искусственного интеллекта, которая относится к формализации и созданию алгоритмов для разработки качественных теоретических и машинных моделей физического мира.

## 20.1. Здравый смысл, качественные рассуждения и обыденные физические представления

### 20.1.1. Различие между количественными и качественными рассуждениями

Рассмотрим схему ванны, показанную на рис. 20.1. Предположим, что первоначально ванна пуста, в нее непрерывно поступает вода из крана и слив закрыт, поэтому нет оттока воды. Что произойдет в дальнейшем?

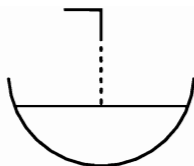


Рис. 20.1. Схема ванны, в которую поступает входной поток, а слив закрыт

Чтобы ответить на этот вопрос, физик-теоретик должен был бы составить модель этой системы в виде дифференциального уравнения и провести расчеты по этой модели с помощью методов численного машинного моделирования. В результате такого моделирования должна быть получена таблица, состоящая, допустим, из 1593 строк, в которой приведены точные значения уровня воды в последовательные моменты времени, отсчитанные по таблице. Например, в этой таблице могло бы быть показано, что уровень воды достигнет края ванны на высоте 62,53 см через 159,3 секунды. Но для повседневного использования такой сложный ответ не нужен. Достаточно просто получить ответ в рамках здравого смысла, который звучит примерно так: «В этих условиях уровень воды будет непрерывно повышаться и в конечном итоге достигнет края ванны. После этого вода начнет выливаться через край и вызовет затопление пола в ванной комнате». Ответ теоретика был количественным и содержал точную числовую информацию, а ответ, полученный на основе здравого смысла, был качественным и содержал просто краткую сводку большого объема количественной информации.

В проблематике искусственного интеллекта область применения качественных рассуждений связана с формализацией и созданием алгоритмов проведения качественных рассуждений о мире для получения качественных, нечисловых ответов на вопросы, которые обычно рассматриваются в теоретической физике как предмет числовых расчетов. Подчеркивая резкий контраст между теоретической физикой, которую изучают в школах и университетах, и качественными рассуждениями о физическом мире на уровне здравого смысла, последние иногда называют также *обыденными физическими представлениями*.

## 20.1.2. Качественное абстрагирование количественной информации

Качественные рассуждения часто рассматриваются как метод абстрагирования результатов количественных рассуждений. В соответствии с этим в качественных рассуждениях отбрасываются некоторые числовые данные, а вместо них используется гораздо более простая краткая качественная сводка этих числовых данных. Существует много способов абстрагирования подробной числовой информации. В табл. 20.1 приведены некоторые примеры количественных утверждений и их качественных абстракций, которые являются типичными для качественных рассуждений в искусственном интеллекте. Принципы абстрагирования, применяемые в этих примерах, рассматриваются в следующих разделах.

Таблица 20.1. Примеры количественных утверждений и их качественных абстракций

| Количественное утверждение     | Качественное утверждение     |
|--------------------------------|------------------------------|
| Level(3.2 s) = 2.6 cm          | Level(t1) = zero..top        |
| Level(3.2 s) = 2.6 cm          | Level(t1) = pos              |
| d/dt Level(3.2 s) = 0.12 m/s   | Уровень Level(t1) возрастает |
| Amount = Level * (Level + 5.7) | M*(Amount, Level)            |

| Время Time | Количество Amount |
|------------|-------------------|
| 0.0        | 0.00              |
| 0.1        | 0.02              |
| ...        | ...               |
| 159.3      | 62.53             |

`Amount(start..end) = zero..top/inc`

## Абстрагирование числовых данных путем их замены символическими значениями и интервалами

Количественное утверждение, что в момент времени 3,2 секунды уровень воды Level достигает значения 2,6 см, формально записывается следующим образом:

`Level(3.2 s) = 2.6 cm`

Качественная абстракция, согласно которой уровень Level в момент времени t1 находится между дном (уровень zero) и краем ванны (уровень top), может быть условно представлена таким образом:

`Level(t1) = zero..top`

Обратите внимание на то, что точные данные о времени 3,2 секунды были заменены символическим обозначением момента времени t1. Поэтому данное утверждение не задает точное время, а сообщает, что есть такой момент времени, обозначенный как t1, в который уровень Level имеет указанное качественное значение. А что касается самого качественного значения, то с его помощью все множество чисел от 0 до 62,53 было свернуто в символический интервал zero..top. В процессе дальнейшего абстрагирования может быть проигнорировано существование края ванны как важного компонента и сформулировано утверждение, что уровень Level в момент времени t1 является положительным, которое записывается следующим образом:

`Level(t1) = pos`

## Абстрагирование производных по времени путем их замены обозначениями направлений изменения

Рассмотрим в качестве примера следующее количественное утверждение о производной уровня Level по времени:

$\frac{d}{dt} \text{Level}(3.2 \text{ s}) = 0.12 \text{ m/s}$

Качественная абстракция этого утверждения состоит в том, что уровень Level в момент времени t1 повышается.

## Абстрагирование функций путем замены монотонными отношениями

Рассмотрим количественное утверждение: `Amount = Level * (Level + 5.7)`.

Вместо него может использоваться следующая качественная абстракция: если уровень Level  $\geq 0$ , то количество воды Amount представляет собой монотонно возрастающую функцию от Level, что формально записывается следующим образом: `M'(Amount, Level)`. Это означает, что при повышении значения Level увеличивается также значение Amount и наоборот.

## Абстрагирование возрастающих временных последовательностей

Вся таблица, содержащая значения количества воды Amount в последовательные моменты времени от 0 до 159,3 секунды, может быть абстрагирована путем замены ее одним качественным утверждением: "Значение количества воды Amount в интервале времени между start и end находится между zero и full, при этом уровень возрастает". Такое утверждение может быть формально записано следующим образом:

`Amount(start..end) = zero..full/inc`

Качественные рассуждения относятся к области *качественного теоретического моделирования*. Числовые модели представляют собой абстракцию реального мира, а качественные модели часто рассматриваются как дальнейшее абстрагирование числовых моделей. В этой абстракции отбрасывается некоторая количественная информация. Например, в количественной модели потока воды в реке может быть указано, что поток Flow связан с уровнем Level воды в реке некоторой сложной зависимостью, в которой также учитывается форма речного дна. В качественной модели такая зависимость может быть абстрагирована путем замены ее следующим монотонно возрастающим отношением:

$M^*(Level, Flow)$

В нем указано, что чем выше уровень, тем больше поток, но эта зависимость не задана с помощью какого-то более конкретного и детализированного способа. Безусловно, что разработка таких грубых качественных моделей намного проще по сравнению с точными количественными моделями.

### 20.1.3. Назначение и область применения качественного моделирования и качественных рассуждений

В данном разделе рассматриваются преимущества и недостатки качественного моделирования по сравнению с традиционным, количественным моделированием. Очевидно, что во многих ситуациях качественная модель неприемлема из-за отсутствия точной числовой информации. Но встречается также много других ситуаций, в которых качественная модель является наиболее предпочтительной.

Во-первых, качественное моделирование проще, чем количественное. Иногда бывает сложно или невозможно определить точные соотношения между переменными в моделируемой системе, но обычно все еще остается возможность установить некоторые качественные соотношения между этими переменными. Кроме того, даже если имеется полная количественная модель, для ее использования обычно требуется также определить все числовые параметры модели, количество которых может оказаться весьма значительным. Например, для использования числовой физиологической модели может потребоваться точное измерение электрической проводимости нейрона, его длины, ширины и т.д. Но измерение этих параметров может оказаться сложным или невозможным. К тому же для выполнения расчетов с помощью такой числовой модели на устройстве машинного моделирования требуется, чтобы пользователем были заданы значения всех этих параметров, прежде чем удастся начать процесс моделирования. В такой ситуации пользователь обычно пытается заменить значения неизвестных параметров предполагаемыми значениями в надежде на то, что они не слишком отличаются от истинных. Но в данном случае пользователь не может определить, насколько далеки результаты числового моделирования от истинных. Как правило, пользователь не знает даже, являются ли полученные результаты правильными с качественной точки зрения. Применение качественных моделей позволяет избежать необходимости выдвижения необоснованных предположений, поэтому пользователь может быть, по крайней мере, уверен в том, что результаты моделирования являются правильными с качественной точки зрения. Таким образом возникает парадоксальная ситуация, при которой количественные результаты, будучи более точными по сравнению с качественными результатами, чаще всего оказываются неправильными и полностью бесполезными, поскольку общая накопленная ошибка становится слишком большой. Например, как показывает практика, в области экологического моделирования применение качественной модели позволяет успешно найти ответ на вопрос, вымрет ли в конечном итоге некоторый вид животных или будет периодически меняться ролями как доминирующий с другими видами, даже если неизвестны точные показатели роста, смертности, рождаемости и т.д. Качественная машинная модель позволяет найти такой ответ путем перебора всех возможных качественных вариантов поведения, которые соответствуют всем возможным сочетаниям значений параметров в модели.

Во-вторых, при решении многих задач точные числовые данные даже не требуются. Такие данные во многих случаях лишь скрывают существенные свойства системы. К типичным задачам, в которых качественное моделирование часто является более приемлемым, относятся функциональные рассуждения, диагностика и структурный синтез. Эти задачи рассматриваются ниже в данном разделе.

Функциональные рассуждения связаны с поиском ответов на вопросы, подобные следующим: “Как работает такое-то устройство или такая-то система?” Примеры таких вопросов приведены ниже.

- Как действует термостат?
- Как функционирует замок?
- Как работают часы?
- Каким образом холодильник выполняет свои функции охлаждения?
- Каким образом сердце обеспечивает перекачку крови?

Во всех этих случаях нас интересует (качественный) механизм работы системы. Если даже числовые значения параметров системы немного изменяются, обычно основной функциональный механизм остается тем же. Все сердца немного отличаются друг от друга, но основной принцип их действия всегда остается одинаковым.

При решении диагностической задачи интерес представляют в основном неисправности, которые вызвали наблюдаемое аномальное поведение системы. Но, как правило, хотелось бы определить только такие отклонения от нормального состояния, которые стали причиной поведения, качественно отличающегося от нормального.

Проблема структурного синтеза состоит в следующем: даны некоторые основные конструктивные блоки; найти такую комбинацию этих блоков, которая позволяет реализовать заданную функцию. Например, требуется так соединить доступные компоненты, чтобы добиться эффекта охлаждения. Иными словами, нужно изобрести холодильник на основе “исходных принципов”. Основными конструктивными блоками могут быть доступные машинные компоненты, или просто законы физики, или материалы с некоторыми свойствами. При таком проектировании, основанном на использовании исходных принципов, задача состоит в синтезе структуры, способной осуществлять некоторые заданные функции с использованием некоторого механизма. На ранней, самой новаторской стадии такого проекта создается качественное описание соответствующего механизма, и только на следующем этапе проектирования, когда структура уже известна, становится также важным количественный синтез.

Для использования качественных моделей требуются определенные методы проведения качественных рассуждений. В оставшейся части этой главы будут описаны и реализованы некоторые идеи качественного моделирования и качественных рассуждений. Прежде всего, в разделе 20.2 рассматриваются статические системы (характеризующиеся тем, что количественные показатели в системе не изменяются во времени). В разделе 20.3 речь идет о качественных рассуждениях, касающихся динамических систем, в которых требуется также проведение рассуждений об изменениях во времени. Математической основой подхода, применяемого в последнем разделе, являются качественные дифференциальные уравнения (Qualitative Differential Equation — QBE), представляющие собой абстрактную версию обычных дифференциальных уравнений.

## 20.2. Качественные рассуждения о статических системах

Рассмотрим простые электрические схемы, состоящие из выключателей, ламп и источников питания (рис. 20.2). Выключатели могут быть разомкнутыми или замкнутыми (т.е. выключенными или включенными), а лампы могут быть светящимися или темными, сгоревшими или исправными. Нас интересует поиск ответов на вопро-

сы, которые относятся к области прогнозирования состояний, диагностики или управления такими электрическими схемами. Одним из примеров вопроса по схеме 1, который относится к области диагностики, является следующий: “Если выключатель включен, а лампа остается темной, то каково состояние этой лампы?” Для того чтобы определить, что данная лампа сгорела, достаточно применить простые качественные рассуждения.

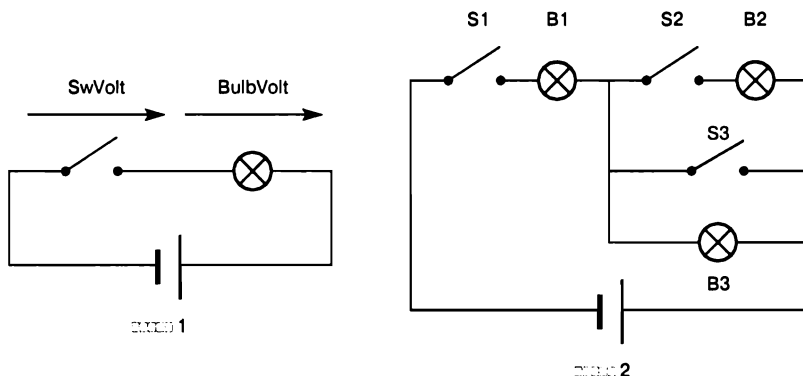


Рис. 20.2. Примеры простых схем, состоящих из выключателей, ламп и источников питания

Одним из примеров более интересного диагностического вопроса, касающегося схемы 2, является следующий: “Видя то, что лампа 2 светится, а лампа 3 остается темной, можно ли сделать достоверный вывод, что лампа 3 сгорела?” Качественные рассуждения подтверждают, что лампа 3, безусловно, является сгоревшей. Для того чтобы светилась лампа 2, через лампу 2 должен проходить ненулевой ток, а выключатель 2 должен быть включен. Если же через лампу 2 течет ненулевой ток, то на лампе 2 должно быть ненулевое напряжение. Для этого требуется, чтобы выключатель 3 был выключен. Такое же ненулевое напряжение поступает на лампу 3. Итак, при подаче одного и того же напряжения лампа 2 светится, а лампа 3 остается темной; это означает, что лампа 3 должна быть сгоревшей.

В рассматриваемой качественной модели подобных схем электрические токи и напряжения имеют лишь качественные значения “pos” (положительный), “zero” (нулевой) и “neg” (отрицательный). Правило абстрагирования, применяемое для преобразования действительного числа  $X$  в качественное значение, приведено ниже.

Если  $X > 0$ , то pos  
 Если  $X = 0$ , то zero  
 Если  $X < 0$ , то neg

В типичных числовых моделях электрических схем используются некоторые фундаментальные законы, такие как законы Кирхгофа и закон Ома. Законы Кирхгофа гласят: во-первых, сумма всех напряжений вдоль любого замкнутого контура в схеме равна 0, во-вторых, сумма всех токов в любом соединении в схеме равна 0. Чтобы применить эти законы в качественной модели, необходимо предусмотреть качественную версию операции арифметического суммирования. В рассматриваемой программе вместо обычного арифметического суммирования  $X + Y = Z$  применяется сокращенный вариант в виде операции качественного суммирования, которая реализована в форме следующего предиката:

`qsum( X, Y, Z)`

Отношение `qsum` может быть определено просто как множество фактов. Эти факты, например, утверждают, что сумма двух положительных чисел представляет собой положительное число:

`qsum( pos, pos, pos).`

Сумма положительного и отрицательного чисел может представлять собой любое из следующих значений:

```
qsum(pos, neg, pos).
qsum(pos, neg, zero).
qsum(pos, neg, neg).
```

Такая операция суммирования является недетерминированной (неопределенной). Из-за отсутствия точной информации, потерянной в процессе качественного абстрагирования, иногда невозможно определить, каковым фактически является результат суммирования. Такого рода недетерминированность является довольно типичной для качественных рассуждений.

Программа, приведенная в листинге 20.1, определяет качественную модель рассматриваемых схем и формализует качественные рассуждения об этой модели. В модели схемы определяются компоненты схемы и учитываются соединения между компонентами. Качественное поведение компонентов двух типов (выключателей и ламп) определяется с помощью следующих предикатов:

```
switch(SwitchPosition, Voltage, Current)
bulb(BulbState, Lightness, Voltage, Current)
```

где SwitchPosition — положение выключателя, Voltage — напряжение, Current — ток, BulbState — состояние лампы, а Lightness — ее светимость.

### Листинг 20.1. Программа качественного моделирования простых электрических схем

```
% Программа качественного моделирования простых электрических схем
% Качественные значения напряжений и токов: neg, zero, pos

% Определение выключателя
% switch(SwitchPosition, Voltage, Current)

switch(on, zero, AnyCurrent). % Выключатель замкнут - напряжение равно нулю
switch(off, AnyVoltage, zero). % Выключатель разомкнут - ток равен нулю

% Определение лампы
% bulb(BulbState, Lightness, Voltage, Current)

bulb(blown, dark, AnyVoltage, zero).
bulb(ok, light, pos, pos).
bulb(ok, light, neg, neg).
bulb(ok, dark, zero, zero).

% Простая электрическая схема, состоящая из лампы, выключателя
% и источника питания

circuit1(SwitchPos, BulbState, Lightness) :-
 switch(SwitchPos, SwVolt, Curr),
 bulb(BulbState, Lightness, BulbVolt, Curr),
 qsum(SwVolt, BulbVolt, pos). % Напряжение источника питания равно pos

% Более интересная схема, состоящая из источника питания, трех ламп
% и трех выключателей

circuit2(Sw1, Sw2, Sw3, B1, B2, B3, L1, L2, L3) :-
 switch(Sw1, VSw1, C1),
 bulb(B1, L1, VB1, C1),
 switch(Sw2, VSw2, C2),
 bulb(B2, L2, VB2, C2),
 qsum(VSw2, VB2, V3),
 switch(Sw3, V3, CSw3),
 bulb(B3, L3, V3, CB3),
 qsum(VSw1, VB1, V1),
 qsum(V1, V3, pos),
 qsum(CSw3, CB3, C3),
 qsum(C2, C3, C1).
```



```
% qsum(Q1, Q2, Q3):
% Q3 = Q1 + Q2, качественное суммарное значение
% в области определения [pos, zero, neg]

qsum(pos, pos, pos).
qsum(pos, zero, pos).
qsum(pos, neg, pos).
qsum(pos, neg, zero).
qsum(pos, neg, neg).
qsum(zero, pos, pos).
qsum(zero, zero, zero).
qsum(zero, neg, neg).
qsum(neg, pos, pos).
qsum(neg, pos, zero).
qsum(neg, pos, neg).
qsum(neg, zero, neg).
qsum(neg, neg, neg).
```

---

Эти варианты качественного поведения являются простыми и могут быть заданы в виде фактов Prolog. Например, на разомкнутом (off) выключателе ток является нулевым (zero), а напряжение может иметь любое значение, как показано ниже.

```
switch(off, AnyVoltage, zero).
```

Сгоревшая (blown) лампа остается темной (dark), через нее не проходит ток, а напряжение может иметь любое значение, как показано ниже.

```
bulb(blown, dark, AnyVoltage, zero).
```

Исправная (intact) лампа светится постоянно, за исключением того случая, когда и напряжение, и ток в лампе равны нулю. При этом предполагается, что любой ненулевой ток является достаточно большим для того, чтобы заставить лампу светиться. Напряжение и ток могут одновременно либо равняться нулю, либо быть положительными, либо быть отрицательными. Обратите внимание на то, что это — качественная абстракция закона Ома, который формулируется следующим образом:

$$\text{Voltage} = \text{Resistance} * \text{Current}$$

Поскольку сопротивление Resistance является положительным, то напряжение Voltage и ток Current должны иметь одинаковый знак и поэтому одно и то же качественное значение.

После того как определены отдельные компоненты, задача определения всей схемы становится несложной. Каждая конкретная схема может быть определена с помощью предиката, например, следующим образом:

```
circuit1(SwPos, BulbState, Lightness)
```

В данном случае предполагается, что наиболее важными свойствами схемы являются положение выключателя SwPos, состояние лампы BulbState и светимость Lightness, поскольку эти переменные используются в качестве параметров предиката circuit1. Но другие количественные показатели схемы, такие как ток, проходящий через лампу, остаются невидимыми за пределами предиката circuit1. Создание модели схемы сводится к определению ограничений, которым должны подчиняться эти параметры, как показано ниже.

1. Законы функционирования лампы.
2. Законы функционирования выключателя.
3. Закон Кирхгоффа (в качественной формулировке): напряжение на выключателе + напряжение на лампе = напряжение источника питания.

Здесь также неявно учтены физические соединения между компонентами, поскольку ток через выключатель равен току через лампу.

Модель схемы 2 (circuit2), хотя и более сложная, создается аналогичным образом.

Ниже приведены некоторые обычные типы вопросов, на которые программа, приведенная в листинге 20.1, легко находит ответ.

## Вопросы прогностического типа

Какими будут наблюдаемые результаты некоторого “входного воздействия” на систему (изменения положений выключателей), если дано некоторое функциональное состояние системы (лампы — исправные или сгоревшие). Например, что произойдет, если будут включены (on) все выключатели, притом что все лампы исправны (ok)?

```
?- circuit2(on,on,on,ok,ok,ok,L1,L2,L3).
L1 = light
L2 = dark
L3 = dark
```

## Вопросы диагностического типа

Если известны входные воздействия на систему и некоторые результаты наблюдений, то каково состояние функционирования системы (исправна она или неисправна, и в чем состоит неисправность?). Например, если лампа 1 светится, лампа 3 остается темной, а выключатель 3 выключен, то каковы состояния ламп?

```
?- circuit2(_, _, off, B1, B2, B3, light, _, dark).
B1 = ok
B2 = ok
B3 = blown
```

## Вопросы управленческого типа

Каким должно быть управляющее воздействие, позволяющее достичь желаемого результата? Например, какими должны быть положения выключателей, чтобы заставить светиться лампу 3, при условии, что все лампы исправны?

```
?- circuit2(SwPos1, SwPos2, SwPos3, ok, ok, ok, _, _, light).
SwPos1 = on
SwPos2 = on
SwPos3 = off;
SwPos1 = on
SwPos2 = off
SwPos3 = off
```

## Упражнения

**20.1.** Определите следующее качественное отношение умножения со знаками:

```
qmult(A, B, C)
```

Здесь  $C = A \cdot B$ , а переменные A, B и C могут иметь качественные значения pos, zero или neg.

**20.2.** Определите качественные модели резистора и диода следующим образом:

```
resistor(Voltage, Current)
diode(Voltage, Current)
```

Диод пропускает ток только в одном направлении. В резисторе напряжение Voltage и ток Current имеют одинаковые знаки. Определите качественные модели некоторых схем с резисторами, диодами и источниками питания.

## 20.3. Качественные рассуждения о динамических системах

В данном разделе рассматривается один из подходов к проведению качественных рассуждений о динамических системах. Рассматриваемый здесь подход основан на использовании так называемых качественных дифференциальных уравнений (Quali-

tative Differential Equation — QDE). Уравнения QDE могут рассматриваться как качественная абстракция обычных дифференциальных уравнений. Для формирования интуитивного представления и ознакомления с основными идеями этого подхода рассмотрим пример заполнения ванны при открытом сливе (рис. 20.3). Для начала проведем неформально некоторые качественные рассуждения об этой системе. При этом будем рассматривать следующие переменные: поток, поступающий в ванну (приток), исходящий поток (отток), количество и уровень воды в ванне.

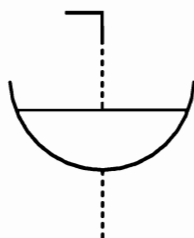


Рис. 20.3. Схема ванны с открытым сливом и постоянным притоком

Предположим, что процесс начинается с пустой ванны. Отток воды, вытекающий через слив, зависит от уровня воды: чем выше уровень, тем больше отток. Приток воды является постоянным. Чистое пополнение объема воды в ванне выражается как разность между притоком и оттоком. Первоначально уровень воды является низким, отток меньше, чем приток, следовательно, количество воды в ванне возрастает. Поэтому уровень также повышается, что приводит к увеличению оттока воды. Таким образом в какой-то момент времени отток может стать равным притоку. По данным точного количественного анализа такая ситуация возникает лишь после “очень продолжительного” времени (которое можно считать бесконечным). После того как это произойдет, оба потока приходят в состояние равновесия и уровень воды становится стабильным. График изменения уровня воды во времени (количественный) выглядит примерно так, как показано на рис. 20.4.



Рис. 20.4. График изменения уровня воды во времени

Количественное поведение уровня, показанное на рис. 20.4, можно упрощенно представить в виде описания качественного поведения следующим образом. Первоначально уровень является нулевым и возрастающим. Выберем для представления этой ситуации следующее обозначение:

Level = zero/inc

После этого наступает такой промежуток времени, в котором уровень находится в пределах от нуля (zero) до максимального значения (top) и продолжает возрастать.

В качественной модели невозможно провести различия между точными числовыми значениями, находящимися в пределах от zero до top. Поэтому считается, что эти значения являются достаточно аналогичными по смыслу и, следовательно, остаются одинаковыми с качественной точки зрения. Итак, справедлива следующая формула:

$Level = zero..top/inc$

Очередное качественное изменение происходит после того, как уровень перестает повышаться и становится стабильным, как показано ниже.

$Level = zero..top/std$

Это — окончательное качественное состояние уровня воды.

Теперь формализуем качественные рассуждения, приведенные выше. Вначале определим качественную модель системы с ванной. Переменные этой системы перечислены ниже.

- Level. Уровень воды.
- Amount. Количество воды.
- Inflow. Входной поток (приток).
- Outflow. Выходной поток (отток).
- Netflow. Чистое пополнение объема воды в ванне ( $Netflow = Inflow - Outflow$ ).

Для каждой переменной определяются ее различные значения, называемые отметками (landmark). Как правило, среди отметок встречаются минус бесконечность (minf), нуль (zero) и плюс бесконечность (inf). Что касается уровня воды в ванне, Level, то важное значение имеет также максимально допустимый уровень воды в ванне (top), поэтому было решено включить в состав отметок и это значение. С другой стороны, поскольку уровень никогда не принимает отрицательного значения, то нет необходимости включать minf в состав отметок для уровня Level. Отметки всегда упорядочены. Поэтому для уровня (Level) задано следующее упорядоченное множество отметок:

$zero < top < inf$

Для количества воды в ванне (Amount) могут быть выбраны такие отметки:

$zero < full < inf$

Теперь определим зависимости между переменными в модели. Эти зависимости называются *ограничениями*, поскольку налагают определенные пределы на значения переменных.

В разрабатываемой модели применяются ограничения некоторых видов, типичные для качественных рассуждений. Одно из таких ограничений определяет зависимость между переменными Amount и Level — чем больше количество воды, тем выше уровень. Это ограничение записывается следующим образом:

$M_0^*(Amount, Level)$

В общем обозначение  $M^*(X, Y)$  указывает, что  $Y$  — монотонно возрастающая функция от  $X$ : после каждого увеличения  $X$  увеличивается также  $Y$  и наоборот. Запись  $M_0^*(X, Y)$  указывает, что  $Y$  — монотонно возрастающая функция от  $X$ , такая, что  $Y(0) = 0$ . Принято считать, что для такой связи  $M^*$  пара  $(0, 0)$  включает соответствующие ей значения. Еще одной парой соответствующих значений для этой связи  $M^*$  является  $(full, top)$ . Важно подчеркнуть, что запись  $M^*(X, Y)$  эквивалентна записи  $M^*(Y, X)$ .

Ограничения, определяющие монотонно возрастающую функцию, являются очень удобными и часто позволяют намного упростить определение модели. Применяя запись  $M_0^*(Amount, Level)$ , мы просто утверждаем, что уровень повышается при каждом увеличении количества и снижается после уменьшения этого количества. Следует отметить, что такое ограничение остается справедливым для любого контейнера

произвольной формы. Если бы вместо этого мы решили определить точное количественное функциональное отношение:

$\text{Amount} = f(\text{Level})$

то оно зависело бы от формы контейнера, как показано на рис. 20.5. Но с качественной точки зрения отношение между уровнем и количеством всегда является монотонно возрастающим, независимо от формы контейнера. Поэтому, чтобы определить качественную модель ванны, нет необходимости подробно изучать ее форму. Благодаря этому задача моделирования часто значительно упрощается, но упрощенная, качественная модель все еще остается приемлемой для достоверного выявления некоторых важных свойств моделируемой системы. Например, если в контейнер поступает приток воды, а отток отсутствует, то количество воды будет увеличиваться, а уровень — повышаться. Поэтому когда-то наступит такой момент времени, что уровень достигнет края контейнера и вода начнет переливаться через край. Подобное качественное поведение является характерным для всех контейнеров (независимо от того, является ли их форма простой или сложной).

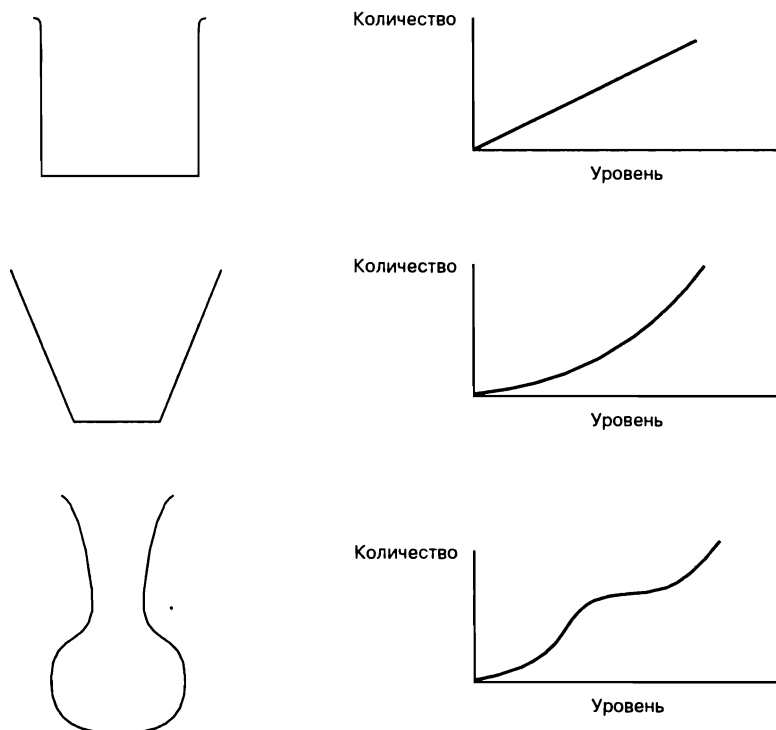


*Рис. 20.5. Точное отношение между количеством и уровнем воды зависит от формы контейнера. Но количество воды всегда является монотонно возрастающей функцией от уровня*

Аналогичным образом, может оказаться сложным точное отношение между оттоком Outflow и уровнем Level. Но с качественной точки зрения можно констатировать, что это отношение является монотонно возрастающим.

Типы ограничений, которые будут использоваться в рассматриваемых качественных моделях, приведены в табл. 20.2. В модели с ванной применяются следующие ограничения:

```

M0+(Amount, Level)
M0+(Level, Outflow)
sum(Outflow, Netflow, Inflow)
deriv(Amount, Netflow)
Inflow = constant = inflow/std

```

**Таблица 20.2. Типы качественных ограничений**

| Ограничение            | Описание                                            |
|------------------------|-----------------------------------------------------|
| $M^+(X, Y)$            | $Y$ — монотонно возрастающая функция от $X$         |
| $M^-(X, Y)$            | $Y$ — монотонно убывающая функция от $X$            |
| $\text{sum}(X, Y, Z)$  | $Z = X + Y$                                         |
| $\text{minus}(X, Y)$   | $Y = -X$                                            |
| $\text{mult}(X, Y, Z)$ | $Z = X * Y$                                         |
| $\text{deriv}(X, Y)$   | $Y = dx/dt$ ( $Y$ — производная по времени от $X$ ) |

Как обычно, имена переменных начинаются с прописных букв, а имена констант — со строчных.

Иногда удобно проиллюстрировать такие ограничения с помощью графа. Узлы графа соответствуют переменным модели, а соединения между узлами обозначают ограничения. На рис. 20.6 показана модель ванны, представленная в виде подобного графа.

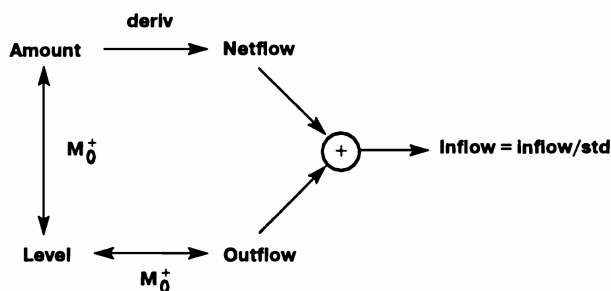


*Рис. 20.6. Графическое представление модели ванны*

Теперь приступим к проведению машинного моделирования по методу качественных рассуждений с использованием модели, приведенной на рис. 20.6. Для этого применяется несколько произвольная система обозначений, которая, тем не менее, допускает только однозначное толкование. Например, запись  $\text{Amount} = \text{zero}$  означает, что качественное значение переменной  $\text{Amount}$  равно нулю, а запись  $\text{Amount} = \text{zero}/\text{inc}$  означает, что качественное значение переменной  $\text{Amount}$  первоначально было равно нулю, а теперь возрастает. Начнем работу со следующего начального условия:

$\text{Amount} = \text{zero}$

С учетом наличия ограничения  $M_0^+$ , связывающего переменные  $\text{Amount}$  и  $\text{Level}$  (см. рис. 20.6), с помощью логического вывода получим следующее:

$\text{Level} = \text{zero}$

К этому значению применяется другое ограничение  $M_0^+$ , что позволяет логически вывести следующее:

$\text{Outflow} = \text{zero}$

После этого ограничение  $\text{Outflow} + \text{Netflow} = \text{Inflow}$  конкретизируется таким образом:

$\text{zero} + \text{Netflow} = \text{inflow}$

Это позволяет получить выражение  $\text{Netflow} = \text{inflow}$ .

Теперь рассмотрим ограничение  $\text{deriv}$  между переменными  $\text{Amount}$  и  $\text{Netflow}$ , которое указывает, что значение  $\text{Netflow}$  равно производной по времени от  $\text{Amount}$ . Поскольку  $\text{Netflow} = \text{inflow} > \text{zero}$ , с помощью логического вывода определяем, что значение  $\text{Amount}$  должно быть возрастающим, как показано ниже.

$\text{Amount} = \text{zero}/\text{inc}$

Применяя к этому значению ограничение  $M_0^+$ , получим следующее:

$\text{Level} = \text{zero}/\text{inc}$

$\text{Outflow} = \text{zero}/\text{inc}$

Затем ограничение  $\text{Outflow} + \text{Netflow} = \text{Inflow}$  конкретизируется, как показано ниже.

$\text{zero}/\text{inc} + \text{Netflow} = \text{inflow}/\text{std}$

Для того чтобы удовлетворялось это ограничение, переменная  $\text{Netflow}$  должна иметь следующее значение:

$\text{Netflow} = \text{inflow}/\text{dec}$

Таким образом, полное первоначальное качественное описание состояния ванны имеет вид

$\text{Amount} = \text{zero}/\text{inc}$

$\text{Level} = \text{zero}/\text{inc}$

$\text{Outflow} = \text{zero}/\text{inc}$

$\text{Netflow} = \text{inflow}/\text{dec}$

Теперь рассмотрим возможные переходы в следующее качественное состояние системы. Предполагается, что кривые изменения значений всех переменных во времени являются гладкими; это означает, что изменение значений происходит непрерывно и производные этих значений по времени также являются непрерывными. Следовательно, переменная, которая является отрицательной, не может стать положительной, не приняв перед этим нулевое значение. Поэтому отрицательная величина может в следующий момент времени либо остаться отрицательной, либо стать равной нулю. Аналогичным образом, возрастающая переменная может либо продолжать возрастать, либо оставаться постоянной. Но она не может внезапно стать уменьшающейся, поскольку вначале эта переменная должна в течение некоторого времени оставаться постоянной. Иными словами, если направление изменения значения переменной обозначается как "inc" (возрастающая), то эта переменная может либо оставаться в состоянии "inc", либо перейти в состояние "std" (неизменная), но не сразу в состояние "dec" (уменьшающаяся). Еще одно ограничение, которое распространяется на возможные переходы между состояниями, состоит в том, что изменяющаяся переменная не может обозначаться больше чем одним экземпляром одного и того же значения отметки. Поэтому переход из состояния "zero/inc" в состояние "zero/inc" невозможен.

Очевидно, что предположение о гладкости в условиях рассматриваемой модели ванны является вполне обоснованным, по меньшей мере в той ситуации, когда уровень  $\text{Level}$  находится в пределах между  $\text{zero}$  и  $\text{top}$ , в связи с чем отток воды через край отсутствует. С учетом ограничения, согласно которому кривые изменения значений переменных должны быть гладкими, следующее качественное состояние переменной  $\text{Level}$  может быть представлено таким образом:

$\text{Level} = \text{zero}..\text{top}/\text{inc}$

Это значение наряду с ограничениями модели на рис. 20.6 определяет качественные состояния других переменных. Поэтому очередное качественное состояние системы принимает следующий вид:

$\text{Level} = \text{zero}..\text{top}/\text{inc}$

$\text{Amount} = \text{zero}..\text{full}/\text{inc}$

$\text{Outflow} = \text{zero}..\text{inflow}/\text{inc}$

$\text{Netflow} = \text{zero}..\text{inflow}/\text{dec}$

Каковы следующие возможные качественные состояния переменной Level? Теперь существуют четыре такие возможности.

1. Level = zero..top/inc.
2. Level = zero..top/std.
3. Level = top/std.
4. Level = top/inc.

В первом случае качественное значение переменной Level остается таким же, как и в предыдущем состоянии. При этом данная модель позволяет определить, что остальные переменные также остаются неизменными. Поэтому в данном случае остается в силе предыдущее описание качественного состояния и нет необходимости вводить новое качественное состояние. Следует отметить, что одно и то же качественное состояние может сохраняться в течение всего рассматриваемого интервала времени (как и в этом случае).

Остальные три возможных перехода соответствуют трем альтернативным вариантам поведения системы, которые описаны ниже.

1. Переменная Level перестает увеличиваться и принимает постоянное значение до того, как соответствующий ей уровень воды достигнет края ванны. Ограничения, показанные на рис. 20.6, позволяют сделать вывод, что другие переменные также принимают постоянные значения и с тех пор не происходят какие-либо изменения. Поэтому в данном случае конечное состояние процесса машинного моделирования может быть представлено следующим образом:

```
Level = zero..top/std
Amount = zero..full/std
Outflow = inflow/std
Netflow = zero/std
```

Строго говоря, это устойчивое состояние достигается только после прохождения бесконечно большого промежутка времени, но это соображение в данном качественном описании не имеет значения, поскольку в нем не учитываются продолжительность интервалов времени.

2. Переменная Level принимает постоянное значение точно в тот момент времени, когда соответствующий ей уровень воды достигает края ванны. Несмотря на то что теоретически такая ситуация возможна, в действительности она является весьма маловероятной. В таком случае все другие переменные также принимают постоянные значения и снова наступает конечное состояние процесса машинного моделирования, аналогичное случаю 1.
3. Уровень воды достигает края ванны и с этого момента времени продолжает возрастать. После этого вода начинает переливаться через край, а модель, приведенная на рис. 20.6, перестает действовать. В такой ситуации происходит резкий переход в новую *рабочую область*. Теперь для этой новой рабочей области требуется другая модель. Кроме того, для описания резких переходов из одной рабочей области в другую нужен особый подход. Такая задача здесь не рассматривается. Поэтому в данном случае выход из рабочей области модели, приведенной на рис. 20.6, рассматривается как переход еще в одно конечное состояние.

Этот пример показывает, что качественная модель способна продемонстрировать несколько вариантов качественного поведения. Безусловно, что в реальной ванне, характеризующейся конкретными физическими параметрами и постоянным притоком, может наблюдаться только один из этих трех вариантов развития событий. Рассматриваемая качественная модель является довольно грубой абстракцией действительности, поскольку в ней исключена вся числовая информация, характеризующая процессы, происходящие при заполнении ванны. Поэтому качественное машинное моделирование не позволяет определить, какой из трех качественных вариантов поведения соответствует развитию событий при фактическом заполнении ванны. Вме-



сто этого в данном случае качественное моделирование позволяет лишь выдвинуть разумные предположения о том, какие варианты развития событий являются возможными.

В данном разделе представлены некоторые основные идеи в области качественного теоретического и машинного моделирования. На основании этих идей разработана программа качественного машинного моделирования, приведенная в следующем разделе.

## 20.4. Программа качественного машинного моделирования

В листинге 20.2 представлена программа качественного машинного моделирования, которая позволяет эксплуатировать модели QDE в соответствии с подходом, описанным в предыдущем разделе. Подробные сведения об этой реализации приведены ниже.

**Листинг 20.2. Программа машинного моделирования, предназначенная для решения качественных дифференциальных уравнений. В этой программе используются обычные предикаты обработки списков `member/2` и `conc/3`**

```
% Интерпретатор качественных дифференциальных уравнений

:- op(100, xfx, ..).
:- op(500, xfx, :).

% landmarks(Domain, [Land1, Land2, ...]
% Land1, Land2 и т.д. - отметки для области определения Domain; они входят
% в состав определения качественной модели, заданного пользователем

% correspond(Constraint):
% переменная Constraint задает соответствующие значения
% для ограничения некоторого типа

correspond(sum(Dom1:zero, Dom2:zero, Dom3:zero)).

correspond(sum(Dom1:L, Dom2:zero, Dom1:L)) :-
 qmag(Dom1:L, L \== zero, not (L = _.._)). % L - это ненулевая отметка
 % в области определения Dom1

correspond(sum(Dom1:zero, Dom2:L, Dom2:L)) :-
 qmag(Dom2:L, L \== zero, not (L = _.._)). % L - это ненулевая отметка
 % в области определения Dom2

correspond(sum(V1, V2, V3)) :-
 correspond(V1, V2, V3). % Определяемые пользователем соответствующие значения

% Ниже приведено пустое определение отношения correspond/3, которое
% предназначено лишь для предотвращения возникновения ошибки "Неопределенный
% предикат" в некоторых версиях системы Prolog

correspond(dummy, dummy, dummy).

% qmag(Domain:QualMagnitude)

qmag(Domain:Qm) :-
 landmarks(Domain, Lands),
 qmag(Lands, Qm).

qmag(Lands, L) :-
 member(L, Lands),
 L \== minf, L \== inf. % Конечная отметка
```

```

qmag(Lands, L1..L2) :- % Интервал
 conc(_, [L1,L2 | _], Lands). % Две смежные отметки

% relative_qmag(Domain1:QM, Domain2:Landmark, Sign):
% переменная Sign - это знак разности между переменными QM и Landmark;
% если QM < Landmark, то Sign = neg и т.д.

relative_qmag(Domain:Ma..Mb, Domain:Land, Sign) :- !,
 landmarks(Domain, Lands),
 (compare_lands(Ma, Land, Lands, neg), Sign = neg, !
 ;
 Sign = pos
).

relative_qmag(Domain:M1, Domain:M2, Sign) :-
 landmarks(Domain, Lands),
 compare_lands(M1, M2, Lands, Sign), !.

% qdir(Qdir, Sign):
% переменная Qdir - это качественное значение направления изменения
% со знаком Sign

qdir(dec, neg).
qdir(std, zero).
qdir(inc, pos).

% Законы качественного суммирования

% qsum(Q1, Q2, Q3):
% Q3 = Q1 + Q2, качественная операция суммирования
% в области определения [pos,zero,neg]

qsum(pos, pos, pos).
qsum(pos, zero, pos).
qsum(pos, neg, pos).
qsum(pos, neg, zero).
qsum(pos, neg, neg).
qsum(zero, pos, pos).
qsum(zero, zero, zero).
qsum(zero, neg, neg).
qsum(neg, pos, pos).
qsum(neg, pos, zero).
qsum(neg, pos, neg).
qsum(neg, zero, neg).
qsum(neg, neg, neg).

% qdirsum(D1, D2, D3):
% качественная операция суммирования обозначений направления изменения

qdirsum(D1, D2, D3) :-
 qdir(D1, Q1), qdir(D2, Q2), qdir(D3, Q3),
 qsum(Q1, Q2, Q3).

% sum(QV1, QV2, QV3):
% QV1 = QV2 + QV3, качественная операция суммирования качественных значений
% в форме Domain:Qmag/Dir. При вызове этого предиката предполагается, что
% области определения всех трех параметров конкретизированы

sum(D1:QM1/Dir1, D2:QM2/Dir2, D3:QM3/Dir3) :-
 qdirsum(Dir1, Dir2, Dir3), % Направления изменений: Dir1 + Dir2 = Dir3
 qmag(D1:QM1), qmag(D2:QM2), qmag(D3:QM3),
 % QM1+QM2=QM3 должны быть согласованы со всеми соответствующими значениями:
 not (

```

```

correspond(sum(D1:V1, D2:V2, D3:V3)), % V1 + V2 = V3
relative_qmag(D1:QM1, D1:V1, Sign1),
relative_qmag(D2:QM2, D2:V2, Sign2),
relative_qmag(D3:QM3, D3:V3, Sign3),
not qsum(Sign1, Sign2, Sign3)).

% mplus(X, Y):
% переменная Y - монотонно возрастающая функция от X

mplus(D1:QM1/Dir, D2:QM2/Dir) :- % Равные направления изменения
qmag(D1:QM1), qmag(D2:QM2),
% переменные QM1, QM2 согласованы со всеми соответствующими значениями
% от D1 до D2:
not (correspond(D1:V1, D2:V2),
 relative_qmag(D1:QM1, D1:V1, Sign1),
 relative_qmag(D2:QM2, D2:V2, Sign2),
 Sign1 \== Sign2).

% deriv(Var1, Var2):
% производная по времени переменной Var1 качественно равна Var2

deriv(Dom1:Qmag1/Dir1, Dom2:Qmag2/Dir2) :-
qdir(Dir1, Sign1),
qmag(Dom2:Qmag2),
relative_qmag(Dom2:Qmag2, Dom2:zero, Sign2), % Sign2 - знак переменной Qmag2
Sign1 = Sign2.

% transition(Domain:Qmag1/Dir1, Domain:Qmag2/Dir2):
% предикат, который определяет переход переменных из одного состояния
% в другое в "последовательные" моменты времени

transition(Dom:L1..L2/std, Dom:L1..L2/Dir2) :-
qdir(Dir2, AnySign).

transition(Dom:L1..L2/inc, Dom:L1..L2/inc).
transition(Dom:L1..L2/inc, Dom:L1..L2/std).
transition(Dom:L1..L2/inc, Dom:L2/inc) :-
L2 \== inf.
transition(Dom:L1..L2/inc, Dom:L2/std) :-
L2 \== inf.
transition(Dom:L1..L2/dec, Dom:L1..L2/dec).
transition(Dom:L1..L2/dec, Dom:L1..L2/std).
transition(Dom:L1..L2/dec, Dom:L1/dec) :-
L1 \== minf.
transition(Dom:L1..L2/dec, Dom:L1/std) :-
L1 \== minf.
transition(Dom:L1/std, Dom:L1/std) :-
L1 \== A..B. % Переменная L1 не относится к интервалу
transition(Dom:L1/std, Dom:L1..L2/inc) :-
qmag(Dom:L1..L2).
transition(Dom:L1/std, Dom:L0..L1/dec) :-
qmag(Dom:L0..L1).
transition(Dom:L1/inc, Dom:L1..L2/inc) :-
qmag(Dom:L1..L2).

```

```

transition(Dom:L1/dec, Dom:L0..L1/dec) :-
 qmag(Dom:L0..L1).

% system_trans(State1, State2):
% переход системы из одного состояния в другое; состояние системы - это
% список значений переменных

system_trans([], []).

system_trans([Val1 | Vals1], [Val2 | Vals2]) :-
 transition(Val1, Val2),
 system_trans(Vals1, Vals2).

% legal_trans(State1, State2):
% переход из одного состояния в другое, допустимый в соответствии с моделью

legal_trans(State1, State2) :-
 system_trans(State1, State2),
 State1 \== State2, % Следующее состояние, качественно отличное от предыдущего
 legalstate(State2). % Состояние, допустимое в соответствии с моделью

% simulate(SystemStates, MaxLength):
% переменная SystemStates представляет собой последовательность состояний
% моделируемой системы, не превышающую по длине MaxLength

simulate([State], MaxLength) :-
 (MaxLength = 1 % Достигнута максимальная длина
 ;
 not legal_trans(State, _) % Отсутствует допустимое следующее состояние
), !.

simulate([State1,State2 | Rest], MaxLength) :-
 MaxLength > 1, NewMaxL is MaxLength - 1,
 legal_trans(State1, State2),
 simulate([State2 | Rest], NewMaxL).

% simulate(InitialState, QualBehaviour, MaxLength)

simulate(InitialState, [InitialState | Rest], MaxLength) :-
 legalstate(InitialState), % Проверка соответствия модели системы
 simulate([InitialState | Rest], MaxLength).

% compare_lands(X1, X2, List, Sign):
% если переменная X1 предшествует переменной X2 в списке List, то Sign = neg;
% если X2 предшествует X1, то Sign = pos; в ином случае Sign = zero

compare_lands(X1, X2, [First | Rest], Sign) :-
 X1 = X2, !, Sign = zero
 ;
 X1 = First, !, Sign = neg
 ;
 X2 = First, !, Sign = pos
 ;
 compare_lands(X1, X2, Rest, Sign).

```

---

## 20.4.1. Способы представления качественных состояний

Переменные в качественной модели могут принимать качественные значения из конкретных областей определения. Например, переменные *Outflow* и *Netflow* могут принимать значения, определяемые отметками из области определения "flow", которая задана для модели ванны. Область определения задается с помощью имени и отметок, например, следующим образом:

```
landmarks(flow, [minf, zero, inflow, inf]).
```

Качественное состояние переменной имеет форму

Domain:Qmag/Dir

где Qmag — качественная величина, которая может представлять собой либо отметку, либо интервал между двумя смежными отметками, который обозначается как Land1..Land2, а Dir — направление изменения, возможными значениями которого являются inc, std, dec. Ниже приведены два примера качественных состояний переменной Outflow.

```
flow:inflow/dec
flow:zero..inflow/dec
```

Качественное состояние системы представляет собой список качественных состояний переменных системы. Например, первоначальное состояние модели ванны состоит из следующих значений четырех переменных Level, Amount, Outflow и Netflow:

```
[level:zero/inc, amount:zero/inc, flow:zero/inc, flow:inflow/dec]
```

Вариант качественного поведения представляет собой список последовательных качественных состояний.

## 20.4.2. Ограничения

В программе, приведенной в листинге 20.2, реализованы три типа ограничений QDE в виде предикатов `deriv( X, Y)`, `sum( X, Y, Z)`, `mplus( X, Y)`. Все параметры этих предикатов ( $X$ ,  $Y$  и  $Z$ ) представляют собой качественные состояния переменных. Рассмотрим каждое из этих ограничений.

- Ограничение `deriv( X, Y)`. С качественной точки зрения  $Y$  — это производная от  $X$  по времени. Проверка этого ограничения является очень простой — направление изменения переменной  $X$  должно соответствовать знаку переменной  $Y$ .
- Ограничение `mplus( X, Y)`. Переменная  $Y$  представляет собой монотонно возрастающую функцию от  $X$ . Здесь  $X$  и  $Y$  имеют форму  $Dx:QmagX/DirX$  и  $Dy:QmagY/DirY$ . При этом, во-первых, направления изменений должны быть согласованы:  $DirX = DirY$ , во-вторых, должны соблюдаться заданные соответствующие значения. Метод проверки второго требования основан на использовании *относительных качественных величин* переменных  $X$  и  $Y$  (они являются относительными применительно к парам соответствующих значений). Например, относительная качественная величина переменной `level:zero..top` по отношению к `top` равна `neg`. Для каждой пары соответствующих значений качественные величины переменных  $X$  и  $Y$  преобразуются в относительные качественные величины. Результирующая относительная качественная величина переменной  $X$  должна быть равна таковой для переменной  $Y$ .
- Ограничение `sum( X, Y, Z)`. Это ограничение соответствует выражению  $X + Y = Z$ , в котором все переменные ( $X$ ,  $Y$  и  $Z$ ) представляют собой качественные состояния переменных в форме `Domain:Qmag/Dir`. Этому ограничению суммирования должны соответствовать и направления изменений, и качественные величины. Во-первых, проверяется согласованность направлений изменений. Например, выражение

```
inc + std = inc
```

является истинным, а следующее выражение — ложным:

```
inc + std = std
```

Во-вторых, качественные величины должны быть согласованы с операцией суммирования. В частности, они должны быть согласованы по отношению ко всем заданным соответствующим значениям, выбранным среди значений  $X$ ,  $Y$  и  $Z$ . Ниже приведены три примера качественных величин, удовлетворяющих ограничению `sum`.

```

flow:zero + flow:inflow = flow:inflow
flow:zero..inflow + flow:zero..inflow = flow:inflow
flow:zero..inflow + flow:zero..inflow = flow:zero..inflow

```

Но следующее выражение является ложным:

```

flow:zero..inflow + flow:inflow = flow:inflow

```

Метод проверки согласованности качественных величин по отношению к соответствующим значениям состоит в следующем. Вначале необходимо преобразовать заданные качественные величины в относительные качественные величины (они являются относительными применительно к соответствующим значениям). Затем необходимо проверить согласованность этих относительных качественных значений применительно к операции качественного суммирования. Рассмотрим следующий пример:

```

sum(flow:zero..inflow/inc, flow:zero..inflow/dec, flow:inflow/std)

```

Во-первых, направления изменений соответствуют ограничению:  $inc + dec = std$ . Во-вторых, рассмотрим три соответствующих значения для операции суммирования, которая при любых условиях является действительной:

```

correspond(sum(D1:zero, D2:Land, D2:Land))

```

где D1 и D2 — области определения, а Land — отметка в области определения D2. Применяя это выражение к области определения потока, получим следующее:

```

correspond(sum(flow:zero, flow:inflow, flow:inflow))

```

Являются ли качественные величины в ограничении sum согласованными с этими соответствующими значениями? Для проверки этого преобразуем три качественные величины в ограничении sum в относительные качественные величины следующим образом:

```

flow:zero..inflow соответствует 'pos' относительно flow:zero
flow:zero..inflow соответствует 'neg' относительно flow:inflow
flow:inflow соответствует 'zero' относительно flow:inflow

```

Теперь эти относительные качественные величины должны удовлетворять ограничению  $qsum( pos, neg, zero)$ . Такое утверждение является истинным.

Математические основы для этой процедуры проверки ограничения sum состоят в следующем. Предположим, что ограничение представлено в виде выражения  $sum(X, Y, Z)$  с тремя соответствующими значениями ( $x_0, y_0, z_0$ ). Переменные X, Y и Z могут быть представлены в терминах их отличий от соответствующих значений таким образом:

$$X = x_0 + \Delta X, Y = y_0 + \Delta Y, Z = z_0 + \Delta Z$$

В таком случае ограничение sum означает следующее:

$$x_0 + \Delta X + y_0 + \Delta Y = z_0 + \Delta Z$$

Поскольку  $x_0 + y_0 = z_0$ , можно сделать вывод, что  $\Delta X + \Delta Y = \Delta Z$ . Знаки переменных  $\Delta X, \Delta Y$  и  $\Delta Z$  определяют относительные качественные величины переменных X, Y и Z применительно к переменным  $x_0, y_0$  и  $z_0$ . Эти относительные качественные величины должны удовлетворять отношению  $qsum$ .

### 20.4.3. Переходы между качественными состояниями

Основным отношением программы, приведенной в листинге 20.2, является следующее:

```

transition(QualState1, QualState2)

```

где QualState1 и QualState2 — последовательные качественные состояния переменной. Они имеют вид  $Domain:Qmag/Dir$ . Отношение transition определяет возможные переходы между качественными состояниями переменных с учетом предположения о гладкости функциональных зависимостей. Согласно этому предположе-

нию кривые изменения переменных в системе в пределах одной и той же рабочей области должны быть непрерывными и гладкими. Например, если значение переменной приближается к некоторой отметке, оно может достичь этой отметки, но не перейти за нее. Аналогичным образом, если направление изменения переменной обозначено как inc, оно может оставаться возрастающим (inc) или стать постоянным (std), но не может стать уменьшающимся (dec), не приняв перед этим значение std. Кроме того, предикат transition позволяет добиться того, чтобы непостоянная переменная сохраняла значение любой отметки лишь на мгновение, но не дольше. Последовательные качественные состояния соответствуют последовательным моментам времени, которые, в принципе, могут быть бесконечно близкими друг к другу. Но чтобы исключить необходимость вырабатывать в процессе машинного моделирования бесконечные последовательности неотличимых друг от друга качественных состояний, данная программа машинного моделирования формирует следующее состояние системы, только если происходит некоторое качественное изменение относительно предыдущего состояния. Вследствие этого любое качественное состояние системы может либо сохраняться в течение единственного момента времени, либо занимать весь рассматриваемый интервал времени. Такой интервал времени соответствует всем последовательным моментам времени, в которые не происходили качественные изменения.

В листинге 20.3 представлена модель ванны в той форме, которая может применяться в программе машинного моделирования, приведенной в листинге 20.2. Ниже показан запрос, который позволяет начать машинное моделирование с состояния, соответствующего пустой ванне, задавая максимальную длину цепочки состояний, равную 10.

```
?- initial(S), simulate(S, Behaviour, 10).
```

### Листинг 20.3. Качественная модель ванны

```
% Модель ванны

landmarks(amount, [zero, full, inf]).
landmarks(level, [zero, top, inf]).
landmarks(flow, [minf, zero, inflow, inf]).

correspond(amount:zero, level:zero).
correspond(amount:full, level:top).

legalstate([Level, Amount, Outflow, Netflow]) :-
 mplus(Amount, Level),
 mplus(Level, Outflow),
 Inflow = flow:inflow/std, % Постоянный приток
 sum(Outflow, Netflow, Inflow), % Netflow = Inflow - Outflow
 deriv(Amount, Netflow),
 not overflowing(Level). % Вода не переливается через край

overflowing(level:top..inf/_). % Вода переливается через край

initial([level: zero/inc,
 amount: zero/inc,
 flow: zero/inc,
 flow: inflow/dec]).
```

Ниже приведен первый из ответов системы Prolog на этот запрос (немного отредактированный).

```
S = [level:zero/inc, amount:zero/inc, flow:zero/inc, flow:inflow/dec]
Behaviour = [
[level:zero/inc,amount:zero/inc,flow:zero/inc,flow:inflow/dec],
[level:zero..top/inc,amount:zero..full/inc,flow:zero..inflow/inc,
flow:zero..inflow/dec],
[level:zero..top/std,amount:zero..full/std,flow:inflow/std,flow:zero/std]]
```

Программу машинного моделирования, приведенную в листинге 20.2, можно легко применить для эксплуатации других моделей. На рис. 20.7 показана электрическая схема с двумя конденсаторами и резистором, а в листинге 20.4 приведена качественная модель этой динамической схемы и соответствующее начальное состояние. В начальном состоянии на левом конденсаторе имеется некоторое начальное напряжение, а правый конденсатор разряжен. Ниже приведены запрос, позволяющий начать процедуру машинного моделирования, и ответ программы моделирования (немного отредактированный).

```
?- initial(S), simulate(S, Behaviour, 10).
Behaviour =
[[volt:v0/dec,volt:zero/inc,...],
[volt:zero..v0/dec,volt:zero..v0/inc,...],
[volt:zero..v0/std,volt:zero..v0/std,...]]
```

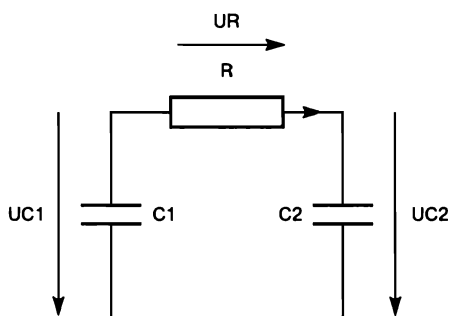


Рис. 20.7. Электрическая схема с двумя конденсаторами и резистором

#### Листинг 20.4. Качественная модель схемы, приведенной на рис. 20.7

---

% Качественная модель электрической схемы с резистором и конденсаторами

```
landmarks(volt, [minf, zero, v0, inf]). % Напряжение на конденсаторах
landmarks(voltR, [minf, zero, v0, inf]). % Напряжение на резисторе
landmarks(current, [minf, zero, inf]).
```

```
correspond(voltR:zero, current:zero).
```

```
legalstate([UC1, UC2, UR, CurrR]) :-
 sum(UR, UC2, UC1),
 mplus(UR, CurrR), % Закон Ома для резистора
 deriv(UC2, CurrR),
 sum(CurrR, current:CurrC1, current:zero/std), % CurrC1 = - CurrR
 deriv(UC1, current:CurrC1). % CurrC1 = d/dt UC1
```

```
initial([volt:v0/dec, volt:zero/inc, voltR:v0/dec, current:zero..inf/dec]).
```

---

По сути, этот ответ означает, что напряжение на конденсаторе C1 будет уменьшаться, а на конденсаторе C2 увеличиваться до тех пор, пока оба напряжения не станут равными (после этого ток и напряжение на резисторе примут нулевое значение).

## Упражнения

20.3. В некоторой системе определены две переменные — X и Y. Варианты их (количественного) поведения во времени имеют следующую форму:

$$X(t) = a_1 \sin(k_1 t), \quad Y(t) = a_2 \sin(k_2 t)$$



Здесь  $a_1$ ,  $a_2$ ,  $k_1$  и  $k_2$  — постоянные параметры системы; значения всех этих параметров больше 0. Начальным моментом времени является  $t_0 = 0$ , поэтому начальное качественное состояние системы может быть представлено в виде  $X(t_0) = Y(t_0) = \text{zero/inc}$ .

- а) Приведите все возможные последовательности первых трех качественных состояний этой системы.
- б) Теперь примите предположение, что между  $X$  и  $Y$  определено качественное ограничение  $M_0^+(X, Y)$ . Перечислите все возможные последовательности первых трех качественных состояний системы, совместимые с этим ограничением.

**20.4.** Качественная модель некоторой системы содержит переменные  $X$ ,  $Y$  и  $Z$  и определяет следующие ограничения:

$M_0^+(X, Y)$   
 $\text{sum}(X, Y, Z)$

Для этих трех переменных заданы такие отметки:

$X, Y$ : minf, zero, inf  
 $Z$ : minf, zero, landz, inf

В момент времени  $t_0$  качественное значение  $x$  равно  $x(t_0) = \text{zero/inc}$ . Каковы качественные значения  $Y(t_0)$  и  $Z(t_0)$ ? Каковы возможные качественные значения  $X$ ,  $Y$  и  $Z$  в следующем качественном состоянии системы, которое сохраняется в течение интервала времени  $t_0..t_1$ , до следующего качественно-го изменения? Каковы возможные новые качественные значения  $X(t_1)$ ,  $Y(t_1)$  и  $Z(t_1)$  после следующего качественного изменения, в момент времени  $t_1$ ?

**20.5.** Определите качественную модель системы сообщающихся сосудов (рис. 20.8) в форме программы Prolog, применимой для программы машинного моделирования (см. листинг 20.2). В этой системе два контейнера соединены тонкой трубой (настолько тонкой, что можно пренебречь инерцией потока воды в трубе). Проведите эксперименты с этой моделью, задавая разные начальные состояния (см. рис. 20.8).

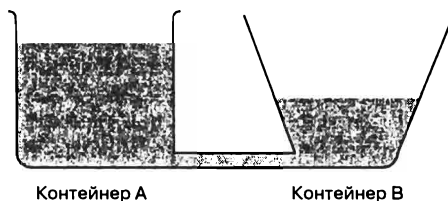


Рис. 20.8. Сообщающиеся сосуды — два контейнера, соединенные тонкой трубой

**20.6.** Дополните программу машинного моделирования, приведенную в листинге 20.2, реализовав другие качественные ограничения, которые часто применяются для решения качественных дифференциальных уравнений:  $\text{minus}(X, Y)$  (где  $X = -Y$ ),  $\text{m\_minus}(X, Y)$  (где  $Y$  — монотонно убывающая функция от  $X$ ),  $\text{mult}(X, Y, Z)$  (где  $Z = X * Y$ ).

**20.7.** Рассмотрите следующую качественную модель движения с ускорением:

```
landmarks(x, [minf, zero, x1, inf]).
landmarks(v, [minf, zero, v0, inf]).
legalstate([X, V]) :-
 V = v:/inc, % Положительное ускорение
 deriv(X, V).
initial([x:zero..x1/inc, v:v0/inc]).
```

Программа качественного машинного моделирования, приведенная в листинге 20.2, вырабатывает следующие результаты (вывод интерпретатора Prolog немного отредактирован):

```
?- initial(S), simulate(S, Behav, 3).
Behav = [[x : zero..x1/inc, v:v0/inc],
[x:zero..x1/inc, v:v0..inf/inc],
[x:x1/inc, v:v0..inf/inc]] ;
Behav = [[x zero..x1/inc, v:v0/inc],
[x:x1/inc, v:v0..inf/inc],
[x:x1..inf/inc, v:v0..inf/inc]]
```

Строго говоря, второй из этих двух выработанных вариантов поведения является неправильным. Проблема возникает при переходе между первым и вторым состояниями системы. Первое состояние включает промежуточное состояние  $v:v0/inc$ , которое может продолжаться лишь в течение единственного момента времени (значение возрастающей переменной может применяться в качестве отметки только на мгновение, но не дольше). Второе состояние включает промежуточное состояние  $x:x1/inc$ , которое также является одномоментным состоянием. Но за одномоментным состоянием не может непосредственно следовать другое кратковременное состояние. Между этими двумя моментами времени должен находиться непустой интервал времени (интервал времени, в котором переменная  $X$  должна достичь значения  $x1$  из интервала  $zero..x1$ ). Нет смысла доказывать, что значение  $X$  в начальном состоянии может быть сколь угодно близким к  $x1$ . Дело в том, что, независимо от того, насколько близким является начальное значение  $X$  к  $x1$ , всегда имеется другое действительное число между начальным значением  $X$  и  $x1$ , поэтому значение  $X$  должно стать равным этому числу, прежде чем достичь  $x1$ . Как можно исправить программу, приведенную в листинге 20.2, чтобы устранить этот недостаток? Подсказка: откорректируйте процедуру `legal_trans`.

## 20.5. Описание программы качественного машинного моделирования

Программа качественного машинного моделирования, представленная в предыдущем разделе, главным образом основана на алгоритме QSIM ([84], [85]). Но в целях упрощения введены некоторые различия между программой, приведенной в листинге 20.2, и первоначальным алгоритмом QSIM. В частности, немного иначе трактуются временные интервалы, что позволяет несколько упростить таблицу переходов между качественными состояниями, реализованы не все типы ограничений, данная программа не вырабатывает новые отметки в процессе моделирования, а в алгоритме QSIM все это предусмотрено. Несмотря на эти различия, приведенное ниже описание преимуществ и недостатков в целом относится к методам машинного моделирования, основанным на использовании качественных дифференциальных уравнений.

Начнем с анализа некоторых преимуществ. Как уже было отмечено, задача формирования качественных моделей обычно проще по сравнению с количественными моделями, к тому же качественные модели в большей степени приемлемы для задач некоторых типов. Кроме того, качественное машинное моделирование по типу QSIM имеет существенное преимущество над числовым машинным моделированием в том, что используемый в нем интервал времени моделирования является адаптивным, тогда как в числовом машинном моделировании интервал времени обычно остается постоянным. Такая гибкость качественного моделирования может оказаться особенно удобной по сравнению с числовым моделированием на основе постоянного временного интервала в тех случаях, когда поведение моделируемой системы резко изменяется. В качестве иллюстрации подобной ситуации снова рассмотрим ванну. До тех пор

пока уровень воды остается ниже края (top), система ведет себя в соответствии с ограничениями, заданными в модели, а после того как уровень достигает края, может начаться переполнение и переход в другую рабочую область. В связи с этим резко изменяются законы функционирования модели. Поэтому в процессе моделирования важно точно определить тот момент, когда уровень воды достигает края воды. В методе QSIM событие, связанное с достижением переменной своей отметки, определяется как качественное изменение, поэтому программа моделирования автоматически фиксирует именно тот момент времени, когда  $Level=top$ . С другой стороны, в программе числового моделирования с постоянным временным интервалом вероятность того, что уровень будет точно равен высоте края ванны в один из зафиксированных моментов времени, весьма мала. Более вероятно то, что следующее моделируемое значение уровня будет либо ниже края, либо (что недопустимо) немного выше.

Теперь рассмотрим некоторые проблемы, возникающие в процессе моделирования по методу QSIM. На первый взгляд кажется вполне обоснованным утверждение, что качественное моделирование является более экономичным с вычислительной точки зрения по сравнению с числовым моделированием. Но в действительности, как это ни парадоксально, на практике может оказаться истинным обратное. Причина этого состоит в недетерминированности качественного моделирования. Даже в рассматриваемой простой модели ванны формируются три разных варианта поведения, а при использовании более сложных моделей количество возможных вариантов поведения часто растет экспоненциально с увеличением количества переходов между состоянием, что может привести к комбинаторному взрыву. При подобных обстоятельствах метод качественного моделирования становится фактически неприемлемым.

В примере с ванной недетерминированность была обусловлена недостаточным объемом информации в модели. Но все три обнаруженные варианта поведения были совместимы с моделью. Можно действительно найти такие ванны, при заполнении которых водой происходят события, соответствующие трем качественным вариантам поведения, обнаруженным данной программой машинного моделирования. Поэтому вполне оправданно то, что результаты, полученные с помощью этой программы, разветвляются по трем направлениям. Но при использовании машинного моделирования по методу QSIM возможен также более проблематичный вид комбинаторного ветвления. Моделирование с помощью такого метода может иногда приводить к формированию вариантов поведения, которые не соответствуют какому-либо из конкретных количественных аналогов качественной модели. Такие варианты поведения являются просто неправильными; они несовместимы с заданной качественной моделью. Эти варианты формально называются *фиктивными вариантами поведения* (spurious behaviour).

В качестве примера рассмотрим простую колебательную систему, состоящую из скользящего блока и пружины (рис. 20.9). Предполагается, что коэффициент трения между блоком и поверхностью равен нулю. Допустим, что первоначально, в момент времени  $t_0$ , пружина растянута на свою *остаточную длину* ( $x = zero$ ) и блок движется с некоторой начальной скоростью  $v_0$  вправо. После этого значение  $x$  возрастет до предела и пружина потянет блок назад, вызывая отрицательное ускорение блока, до тех пор, пока блок не остановится и не начнет движение в противоположном направлении. Затем блок пересечет нулевую отметку с некоторой отрицательной скоростью, достигнет крайней позиции слева и возвратится в положение  $x = zero$ . Поскольку трение отсутствует, можно предположить, что в этот момент времени скорость блока снова будет равна  $v_0$ . В дальнейшем весь этот цикл будет повторяться. Результатирующий вариант поведения представляет собой устойчивые колебания.

Проверим эту модель с помощью программы машинного моделирования (см. листинг 20.2). Количественная модель этой системы выглядит так:

$$\frac{d^2}{dt^2} x = A$$

$$A = -\frac{kx}{m}$$

где  $X$  — положение блока,  $A$  — его ускорение,  $m$  — масса,  $k$  — коэффициент упругости пружины. Соответствующая качественная модель приведена в листинге 20.5.

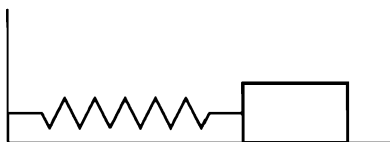


*Рис. 20.9. Схема механической системы, состоящей из скользящего блока и пружины, в которой отсутствует трение между блоком и поверхностью опоры*

#### Листинг 20.5. Качественная модель системы, состоящей из блока и пружины

```
% Модель системы, состоящей из блока и пружины

landmarks(x, [minf, zero, inf]). % Положение блока
landmarks(v, [minf, zero, v0, inf]). % Скорость блока
landmarks(a, [minf, zero, inf]). % Ускорение блока

correspond(x:zero, a:zero).

legalstate([X, V, A]) :-
 deriv(X, V),
 deriv(V, A),
 MinusA = a:_,
 sum(A, MinusA, a:zero/std), % MinusA = -A
 mplus(X, MinusA). % Пружина тянет блок назад

initial([x:zero/inc, v:v0/std, a:zero/dec]).
```

Вызовем эту модель на выполнение из начального состояния с помощью следующего запроса:

```
?- initial(S), simulate(S, Beh, 11).
```

Сформированный вариант поведения `Beh` соответствует ожидаемому вплоть до состояния 8, как показано ниже.

```
[x:minf..zero/inc,v:zero..v0/inc,a:zero..inf/dec]
```

После этого варианты поведения разветвляются в трех направлениях. В первой ветви поведение развивается следующим образом:

```
[x:minf..zero/inc,v:v0/inc,a:zero..inf/dec]
[x:minf..zero/inc,v:v0..inf/inc,a:zero..inf/dec]
[x:zero/inc,v:v0..inf/std,a:zero/dec]
```

В данном случае скорость достигает начального значения скорости  $v_0$  еще до того, как  $X$  становится равным нулю. В тот момент, когда  $X = \text{zero}$ , скорость уже больше  $v_0$ . Создается впечатление, что при этом возникает ситуация, невозможная с физической точки зрения, поскольку суммарная энергия в системе возрастает, а сам этот вариант поведения выглядит как возрастание интенсивности колебаний. Во второй ветви за состоянием 8 следуют такие переходы между состояниями:

```
[x:zero/inc,v:zero..v0/std,a:zero/dec]
[x:zero..inf/inc,v:zero..v0/dec,a:minf..zero/dec]
[x:zero..inf/std,v:zero/dec,a:minf..zero/std]
```

В данном случае блок достигает позиции  $X = \text{zero}$  при значении скорости меньше  $v_0$ . При этом суммарная энергия в системе становится меньше, чем в начальном

состоянии, поэтому создается впечатление, что колебания затухают. А в третьей ветви за состоянием 8 следуют переходы

```
[x:zero/inc,v:v0/std,a:zero/dec]
```

...

Этот вариант развития событий соответствует ожидаемому варианту с устойчивыми колебаниями.

Возникает вопрос: является ли появление этих двух непредвиденных вариантов поведения только следствием отсутствия в качественной модели необходимой информации, или проблема заключается в самом алгоритме машинного моделирования? Можно показать, что в действительности эта модель, хотя и является качественной, содержит достаточно информации, чтобы допускать формирование только варианта с устойчивыми колебаниями. Поэтому остальные два варианта поведения, с возрастающими и затухающими колебаниями, являются математически несовместимыми с данной моделью. Такие варианты называются *фиктивными*. Недостаток, который приводит к появлению этих вариантов, заключается в алгоритме моделирования. В связи с этим сразу же возникает вопрос о том, почему же нельзя исправить этот дефект в самом алгоритме! Но сложность решения такой задачи состоит в том, что это — не простой дефект, а сложная вычислительная проблема. Для решения указанной проблемы необходимо найти способ проверки соответствия качественных вариантов поведения всем ограничениям, налагаемым моделью. Алгоритм моделирования по методу QSIM предусматривает проверку совместимости с ограничениями отдельных состояний, а не всех последовательностей этих состояний в целом. Несмотря на то что удалось найти много способов усовершенствования этого алгоритма, позволяющих устранить значительную часть фиктивных вариантов поведения, полное решение этой проблемы еще не обнаружено.

Зная об этом недостатке моделирования по методу QSIM, можем ли мы гарантировать получение приемлемых результатов моделирования? Доказана теорема [84], согласно которой метод QSIM гарантирует выработку всех качественных вариантов поведения, совместимых с моделью. Поэтому неправильные результаты применения этого метода ограничиваются лишь противоположными случаями — при использовании метода QSIM наряду со всеми правильными могут вырабатываться и неправильные варианты поведения, несовместимые с моделью (т.е. фиктивные варианты поведения). На рис. 20.10 показаны отношения между рассматриваемыми уровнями абстракции — обыкновенными дифференциальными уравнениями и их решениями на количественном уровне, с одной стороны, а также качественными дифференциальными уравнениями и качественными вариантами поведения, вырабатываемыми с помощью метода QSIM на качественном уровне, с другой. Те сформированные варианты поведения, которые не представляют собой абстракции каких-либо количественных решений, являются фиктивными.

Практическая значимость такой “односторонней” правильности метода QSIM состоит в следующем. Предположим, что программа машинного моделирования, в которой реализован алгоритм QSIM, сформировала некоторые качественные варианты поведения на основании рассматриваемой модели. Нам известно, что это множество вариантов является полным в том смысле, что в моделируемой системе не существуют какие-либо иные варианты поведения, которые еще не включены в состав этих результатов. Мы знаем, что любые другие варианты не могут возникнуть. Но у нас нет гарантии того, что все сформированные варианты поведения моделируемой системы действительно возможны.

Один из способов устранения фиктивных вариантов поведения состоит в том, чтобы ввести в модель дополнительные ограничения, но при том условии, что должна быть сохранена правильность этой модели. Обычно такая задача является непростой, но в случае механической системы, состоящей из блока и пружины, ее решить не сложно. Известно, что энергия в этой системе должна быть постоянной, поскольку отсутствуют потери энергии под действием трения, а также не происходит поступление энергии в систему. Совокупная энергия представляет собой сумму кинетической

и потенциальной энергии, поэтому ограничение, соответствующее закону сохранения энергии, может быть выражено следующим образом:

$$\frac{1}{2} mV^2 + \frac{1}{2} kX^2 = \text{const.}$$

где  $m$  — масса блока, а  $k$  — коэффициент упругости пружины. Но в данном случае достаточно применить упрощенную версию этого ограничения. А именно, из приведенной выше формулы сохранения энергии следует, что при  $X = 0$  справедливо выражение  $V^2 = v_0^2$ . Эквивалентное ограничение состоит в том, что если  $V = v_0$ , то  $X = 0$ , а если  $X = 0$ , то  $V = v_0$  или  $V = -v_0$ . При использовании приведенной ниже модификации модели системы, состоящей из блока и пружины (см. листинг 20.5), фиктивные решения не вырабатываются.

```
legalstate([X, V, A]) :-
 deriv(X, V),
 deriv(V, A),
 MinusA = a:_,
 sum(A, MinusA, a:zero/std), % MinusA = -A
 mplus(X, MinusA), % Пружина тянет блок назад
 energy(X, V). % Слабое ограничение сохранения энергии
energy(X, V) :-
 V = v:v0/_, !, X = x:zero/_ % Если V=v0, то переменная X
 % должна быть равна zero
;
X = x:zero/_, !, V = v:minf..zero/_ % Здесь переменная V
 % должна быть равна -v0
;
true. % В противном случае X имеет значение, отличное от zero,
 % и на переменную V не налагаются какие-либо ограничения
```

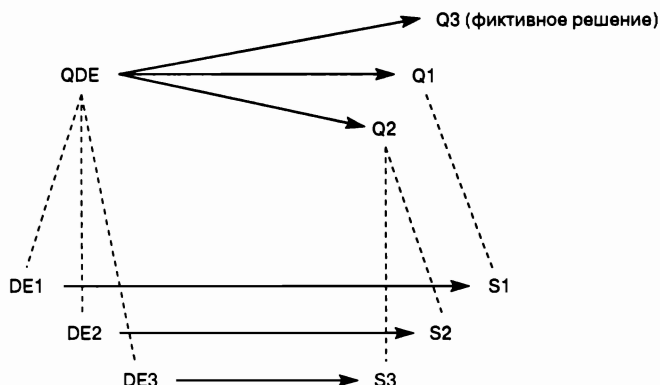


Рис. 20.10. Качественные абстракции дифференциальных уравнений и их решений, где QDE — качественная абстракция трех дифференциальных уравнений, DE1, DE2 и DE3, а S1, S2 и S3 — решения этих трех дифференциальных уравнений. Качественные варианты поведения Q1, Q2 и Q3 выработаны как решения уравнений QDE; Q1 — качественная абстракция S1, Q2 — абстракция S2 и S3, Q3 — фиктивный вариант, поскольку он не является абстракцией решения какого-либо из соответствующих дифференциальных уравнений

- Решения, полученные в рамках “теоретической” физики, часто содержат гораздо больше числовых данных, чем требуется в повседневной жизни, поэтому в подобных случаях более подходящими являются здравый смысл, *качественные рассуждения* и “обыденные физические представления”.
- *Качественное теоретическое моделирование* и качественные рассуждения обычно рассматриваются как *абстракция* количественного теоретического моделирования и количественных рассуждений. Числа заменяются их знаками (положительными или отрицательными), символическими значениями (иногда называемыми *отметками*) или интервалами, вместо точных значений времени применяются символические обозначения моментов времени и интервалов, а производные по времени упрощенно представляются в виде направлений изменения (возрастающее, уменьшающееся или постоянное). Конкретные функциональные зависимости могут быть заменены менее определенными, такими как монотонные зависимости.
- Задача создания *качественных моделей* является более простой по сравнению с созданием количественных моделей. В связи с недостаточно высокой числовой точностью качественные модели не всегда позволяют получить достаточно точное решение, но в целом являются более подходящими для решения задач диагностики, функциональных рассуждений и проектирования на основе “исходных принципов”, например законов физики.
- При проведении качественных рассуждений арифметические операции упрощенно представляются с помощью *качественных арифметических действий*. Примером может служить качественное суммирование, в котором учитываются лишь положительные и отрицательные знаки, а также нулевые значения — pos, zero и neg. Как правило, результаты качественных арифметических действий являются недетерминированными.
- *Качественные дифференциальные уравнения* (Qualitative Differential Equation — QDE) представляют собой абстракцию обыкновенных дифференциальных уравнений. QSIM — это алгоритм качественного моделирования, предназначенный для моделей, которые определены с помощью качественных дифференциальных уравнений. Основная предпосылка, лежащая в основе моделирования по методу QSIM, состоит в том, что кривые, на основании которых разработана модель, являются *гладкими* — в пределах одной и той же *рабочей области* значения переменных могут изменяться только непрерывно.
- *Сложность* обеспечения правильного качественного моделирования состоит в том, что вырабатываемые при этом результаты являются полными, но не всегда правильными. Можно сравнительно легко добиться того, чтобы программа моделирования выработала все варианты поведения, совместимые с заданными качественными дифференциальными уравнениями. С другой стороны, гораздо сложнее обеспечить формирование только тех вариантов поведения, которые являются совместимыми с заданными качественными дифференциальными уравнениями. Эта проблема известна как проблема выработки *фиктивных вариантов поведения*.
- В этой главе рассматривались следующие понятия:
  - качественные рассуждения;
  - качественное теоретическое моделирование;
  - здравый смысл и обыденные физические представления;
  - качественные абстракции;
  - отметки;

- качественные арифметические операции;
- качественная операция суммирования;
- качественное дифференциальное уравнение (Qualitative Differential Equation — QDE);
- монотонные функциональные ограничения;
- качественное значение, качественное состояние переменной;
- качественное машинное моделирование;
- алгоритм QSIM;
- применение предположения о гладкости кривых в качественном моделировании динамических систем;
- непрерывный переход между качественными состояниями;
- фиктивный вариант поведения.

## Дополнительные источники информации

В журнале *Artificial Intelligence* был опубликован специальный выпуск [8], озаглавленный как “Qualitative Reasoning about Physical Systems” (Качественные рассуждения о физических системах). Некоторые материалы из этого тома журнала стали классическими источниками информации в области качественных рассуждений; к ним относятся статья, посвященная конфлюэнтному анализу [40], и статья под названием *Qualitative Process Theory* (Теория качественных процессов) [53]. Несмотря на то что эти статьи в большей степени представляют интерес для исследований, проводимых на более высоких уровнях, чем качественные дифференциальные уравнения (QDE), они тесно связаны с проблематикой QDE. Качественные дифференциальные уравнения могут рассматриваться как основополагающая формальная система низкого уровня, на базе которой могут быть составлены применяемые в этих статьях описания проблем более высокого уровня. Спустя некоторое время в журнале *Artificial Intelligence* был опубликован еще один специальный выпуск [41], посвященный качественным рассуждениям. В дальнейшем был выпущен сборник наиболее важных статей, опубликованных до 1990 года [167]. Еще одним сборником статей по качественным рассуждениям является [48]. Для безотлагательной публикации результатов современных исследований в области качественных рассуждений открыт специальный форум — ежегодный симпозиум *Workshop on Qualitative Reasoning*.

Программа качественного моделирования динамических систем, приведенная в этой главе, основана на алгоритме QSIM [84] для моделей QDE. В целях упрощения в этой программе не реализован полный набор качественных ограничений алгоритма QSIM, в процессе моделирования не вводятся новые отметки и не проводятся явные различия между моментами времени и интервалами времени. В [92] и [135] приведен анализ основных сложностей моделирования в стиле QSIM. В [85] описаны усовершенствования, позволяющие частично решить проблему фиктивных вариантов поведения, а также приведены некоторые другие результаты разработок в области создания алгоритма QSIM. В [138]–[140] описаны другие достижения в области устранения фиктивных вариантов поведения. Упражнение, приведенное в данной книге под номером 20.7, было предложено автору Симом Сэем (Cem Say) при личном общении.

Безусловно, что качественные рассуждения о физиологических системах не должны быть основаны на дифференциальных уравнениях или на их непосредственной абстракции. Один из подходов к моделированию сложной физиологической системы, в котором не предусматривается использование какой-либо основополагающей связи с дифференциальными уравнениями, описан в [20], где модель сердца, объясняющая отношения между сердечными аритмиями и сигналами электрокардиограммы, определена в терминах качественных описаний на основе логики.



В [52] рассматривается перспективная идея объединения качественного и числового моделирования.

Один из важных практических вопросов состоит в том, как следует формировать качественные модели и можно ли автоматизировать этот процесс. Проблема автоматического формирования моделей QDE динамических систем по наблюдаемым результатам исследования вариантов поведения моделируемых систем рассматривалась в [22], [36], [64], [83], [141] и [162]. Во всех этих работах приведены результаты синтеза небольших моделей типа QDE по заданным результатам наблюдения вариантов поведения. В [108] и [109] (а также в [20]) описаны результаты синтеза сложной качественной модели электрической активности сердца, полученной с помощью машинного обучения, при формировании которой не использовались качественные дифференциальные уравнения.

# Обработка лингвистической информации с помощью грамматических правил

*В этой главе...*

|                                                                    |     |
|--------------------------------------------------------------------|-----|
| 21.1. Применение грамматических правил в программе на языке Prolog | 510 |
| 21.2. Обработка смысла                                             | 516 |
| 21.3. Определение смысла фраз на естественном языке                | 521 |

Во многих реализациях Prolog предусмотрена дополнительная система обозначений, называемая DCG (Definite Clause Grammar — грамматика определенных предложений, или DC-грамматика). Это дополнение намного упрощает задачу определения формальных грамматик на языке Prolog. Грамматика, заданная в системе обозначений DCG, может быть непосредственно вызвана на выполнение с использованием интерпретатора Prolog в качестве синтаксического анализатора. Расширение DCG упрощает также трактовку семантики языка, в результате чего семантические (смысловые) конструкции в языковой фразе, которая определена с помощью DC-грамматики, могут чередоваться с синтаксическими. В этой главе показано, благодаря чему расширение DCG обеспечивает изящное определение синтаксиса и смысла нетривиальных фраз на естественном (английском) языке, например, таких: “Every woman that admires a man that paints likes Monet” (Каждая женщина, которая восхищается мужчиной, умеющим рисовать, любит Моне).

## 21.1. Применение грамматических правил в программе на языке Prolog

*Грамматика* — это формальный механизм для определения множеств последовательностей символов. Такая последовательность символов может быть отвлеченной, не имеющей какого-либо практического значения, или, что более интересно, может представлять собой инструкцию на языке программирования, или целую программу, а также может быть фразой на естественном языке, таком как английский.

Одной из популярных систем определения грамматики является BNF (Backus-Naur Form — форма Бэкуса-Наура), которая часто используется при описании языков программирования. Начнем изложение рассматриваемой темы с анализа системы BNF. Грамматика состоит из порождающих правил. Ниже приведена простая грамматика BNF, состоящая из двух правил.

```
<s>::=ab
<s>::=a<s>b
```

Первое правило гласит: каждое вхождение символа  $s$  в строке может быть заменено последовательностью  $ab$ , а второе правило утверждает, что символ  $s$  может быть заменен последовательностью, состоящей из символа  $a$ , за которым следует  $s$ , затем  $b$ . В этой грамматике символ  $s$  всегда заключен в угловые скобки " $<>$ ". Такое обозначение указывает, что  $s$  — нетерминальный символ грамматики. С другой стороны,  $a$  и  $b$  — это терминальные символы. Терминальные символы ни в коем случае не подлежат замене. В системе BNF приведенные выше два порождающих правила обычно записываются вместе в виде одного правила:

```
<s>::=ab|a<s>b
```

Но в данной главе для упрощения принята расширенная, более полная форма.

Грамматика может использоваться для формирования строки символов, которая называется *фразой*. Процесс формирования всегда начинается с некоторого начального нетерминального символа, в данном примере — с символа  $s$ . После этого символы в текущей последовательности заменяются другими строками в соответствии с правилами грамматики. Процесс формирования заканчивается, когда текущая последовательность не содержит ни одного нетерминального символа. В данном примере грамматики процесс формирования может происходить, как описано ниже. Формирование начинается со следующего символа:

$s$

Теперь в соответствии со вторым правилом  $s$  заменяется такой последовательностью символов:

$a\ s\ b$

После этого снова используется второе правило, что приводит к получению последовательности:

$a\ a\ s\ b\ b$

После применения первого правила окончательно формируется следующая фраза:

$a\ a\ a\ b\ b\ b$

Безусловно, что эта грамматика позволяет формировать и другие фразы, например  $ab$ ,  $aabb$  и т.д. В общем эта грамматика позволяет создавать строки в форме  $a^n b^n$ , где  $n = 1, 2, 3, \dots$ . Множество фраз, формируемых с помощью грамматики, называется *языком*, определяемым этой грамматикой.

Рассматриваемый пример грамматики является простым и очень далеким от практических потребностей. Но грамматики могут использоваться для определения гораздо более интересных языков. В частности, формальные грамматики применяются для определения языков программирования и подмножеств естественных языков.

Следующая рассматриваемая грамматика все еще остается очень простой, но имеет большее практическое значение. Предположим, что манипулятору робота могут передаваться показанные ниже последовательности команд.

- up — переместиться вверх на один шаг;
- down — переместиться на один шаг вниз.

Ниже приведены два примера последовательностей команд, которые может воспринимать такой робот.

```
up
up up down up down
```

Подобная последовательность активизирует соответствующую последовательность шагов, выполняемых роботом. Мы будем называть любую последовательность шагов "движением". Таким образом, любое движение (move) состоит либо из одного шага (step), либо из любого шага, за которым следует любое движение. Такая организация функционирования робота определяется с помощью следующей грамматики:

```

<move> ::= <step>
<move> ::= <step> <move>
<step> ::= up
<step> ::= down

```

Эта грамматика будет использоваться ниже в данной главе для иллюстрации того, каким образом может осуществляться обработка смысла в рамках системы обозначения грамматик на языке Prolog.

Как было показано выше, грамматика позволяет формировать фразы. С другой стороны, грамматика может использоваться для распознавания заданной фразы. Механизм распознавания позволяет определить, принадлежит ли данная фраза к некоторому языку; это означает, что механизм распознавания устанавливает, может ли заданная фраза быть сформирована с помощью соответствующей грамматики. Процесс распознавания фраз, по сути, является обратным по отношению к процессу их формирования. При распознавании работа начинается с заданной строки символов, к которой применяются грамматические правила в направлении, противоположном формированию: если текущая строка содержит подстроку, равную правой части некоторого правила грамматики, то эта подстрока заменяется левой частью такого правила. Процесс распознавания завершается успешно после того, как вся заданная фраза приводится к начальному нетерминальному символу грамматики. А если не удастся обнаружить способ приведения заданной фразы к начальному нетерминальному символу, то механизм распознавания отбрасывает эту фразу.

В таком процессе распознавания заданная фраза фактически раскладывается на синтаксические компоненты, поэтому подобный процесс часто называют также *синтаксическим анализом*. Обычно для реализации некоторой грамматики требуется не только составить ее правила, но и подготовить программу синтаксического анализа для этой грамматики. Как показано ниже в этой главе, такие программы синтаксического анализа могут быть написаны на языке Prolog очень легко. Эта задача может быть решена с помощью языка Prolog особенно изящно благодаря использованию специальной системы обозначений правил грамматики, называемой DCG (Definite Clause Grammar — грамматика определенных предложений). Такая специальная система обозначений для грамматик поддерживается многими реализациями Prolog. Любая грамматика, оформленная с помощью системы DCG, может непосредственно использоваться в качестве программы синтаксического анализа фраз, сформированных с помощью этой грамматики. Для преобразования грамматики BNF в формат DCG достаточно лишь применить немного иные соглашения системы обозначений. В частности, приведенные выше в качестве примера грамматики BNF могут быть записаны в формате DCG следующим образом:

```

s-->[a],[b].
s-->[a],s,[b].

move --> step.
move --> step, move.
step --> [up].
step --> [down].

```

Обратите внимание на то, в чем состоят различия между системами обозначений BNF и DCG. Символ “::=” заменяется символом “-->”. Нетерминальные символы больше не записываются в угловых скобках, а терминальные символы помещаются в квадратные скобки и тем самым преобразуются в списки Prolog. Кроме того, теперь символы отделены друг от друга запятыми, а каждое правило оканчивается точкой, как и любое предложение Prolog.

В реализациях Prolog, которые допускают возможность применения системы обозначений DCG, такие преобразованные грамматики могут непосредственно использоваться в качестве механизмов распознавания фраз. Соответствующий механизм распознавания, по сути, рассчитан на то, что фразы представлены в виде разностных списков терминальных символов. (Представление в виде разностных списков было описано в главе 8.) Поэтому каждая фраза оформлена в виде двух списков и пред-

ставляет собой разность между этими двумя списками. Эти два списка не являются уникальными, например, как показано ниже.

Фраза aabb может быть представлена списками [ a, a, b, b] и [ ] или списками [ a, a, b, b, c] и [ c] или списками [ a, a, b, b, 1, 0, 1] и [ 1, 0, 1]

С учетом того, что для представления фраз применяется такая форма, можно воспользоваться системой обозначений DCG для распознавания некоторых фраз, сформированных с помощью грамматик, рассматриваемых в качестве примера, введя следующие вопросы:

```
?- s([a, a, b, b], []). % Распознать строку aabb
yes
?- s([a,a,b], []).
no
?- move([up, up, down], []).
yes
?- move([up, up, left], []).
no
?- move([up, X, up], []).
X = up;
X = down;
no
```

Теперь рассмотрим, каким образом в интерпретаторе Prolog используется заданная форма DCG для поиска ответов на такие вопросы. В ходе применения для консультаций правил грамматики интерпретатор Prolog автоматически преобразует их в обычные предложения Prolog. Таким образом, интерпретатор Prolog преобразует заданные правила грамматики в программу для распознавания фраз, сформированных с помощью этой грамматики. Применение такого преобразования иллюстрируется в следующем примере. Приведенные выше четыре правила DCG, касающиеся движений робота, преобразуются в следующие четыре предложения:

```
move(List, Rest) :-
 step(List, Rest).
move(List1, Rest) :-
 step(List1, List2),
 move(List2, Rest).

step([up | Rest], Rest).
step([down | Rest], Rest).
```

Какая задача фактически решается с помощью такого преобразования? Рассмотрим процедуру move. Отношение move имеет два параметра — два списка:

```
move(List, Rest)
```

Это отношение принимает истинное значение, если разность между списками List и Rest представляет собой допустимое движение. В качестве примеров отношений можно указать следующие:

```
move([up, down, up], []).
```

или

```
move([up, down, up, a, b, c], [a, b, c]).
```

или

```
move([up, down, up], [down, up]).
```

На рис. 21.1 показано, что подразумевается под следующим предложением:

```
move(List1, Rest) :-
 step(List1, List2),
 move(List2, Rest).
```

Это предложение можно описать словами таким образом:

Разность списков List1 и Rest обозначает движение, если разность между списками List1 и List2 соответствует шагу и разность между списками List2 и Rest — движению.

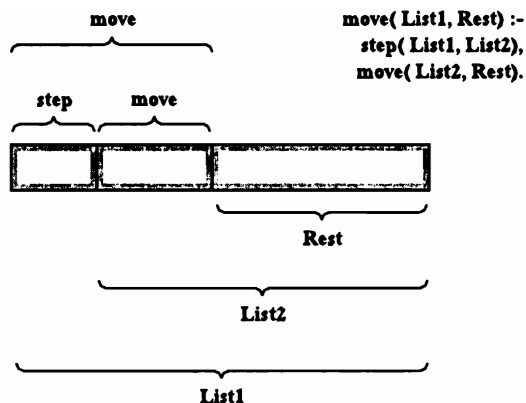


Рис. 21.1. Отношения между последовательностями символов

Такое описание позволяет также понять, почему используется представление в виде разностных списков: пара ( List1, Rest) представляет собой конкатенацию списков, оформленных в виде пар ( List1, List2) и ( List2, Rest). Как показано в главе 8, конкатенация списков, представленных таким образом, осуществляется намного эффективнее по сравнению с конкатенацией обычных списков с помощью предиката conc.

Теперь мы можем приступить к подготовке более общего описания процесса трансляции, в котором устанавливается связь между системой обозначений DCG и стандартным языком Prolog. Каждое правило DCG транслируется в предложение Prolog в соответствии с описанной ниже основной схемой. Вначале предположим, что правило DCG выглядит следующим образом:

```
n --> n1, n2, ..., nn.
```

Если все компоненты n1, n2, ..., nn этого правила представляют собой нетерминальные символы, то оно транслируется в следующее предложение:

```
n(List1, Rest) :-
 n1(List1, List2),
 n2(List2, List3),
 ...
 nn(Listn, Rest).
```

Если же какой-либо из компонентов n1, n2, ..., nn является терминальным символом (задан в правиле DCG в квадратных скобках), то он обрабатывается иначе. Он не появляется в виде одной из целей в предложении, а непосредственно вставляется в соответствующий список. В качестве примера рассмотрим следующее правило DCG:

```
n-->n1, [t2], n3, [t4].
```

где n1 и n3 — нетерминальные символы, а t2 и t4 — терминальные. Это правило транслируется в такое предложение:

```
n(List1, Rest) :-
 n1(List1, [t2 | List3]),
 n3(List3, [t4 | Rest]).
```

Более интересные примеры грамматик можно найти в языках программирования и в естественных языках. И те и другие могут быть изящно реализованы с помощью DCG. Ниже приведен пример грамматики для простого подмножества английского языка.

```
sentence --> noun_phrase, verb_phrase.
verb_phrase --> verb, noun_phrase.
noun_phrase --> determiner, noun.
```

```
determiner --> [a].
determiner --> [the].
noun --> [cat],
noun --> [mouse].
verb --> [scares],
verb --> [hates].
```

Ниже приведены примеры фраз, формируемых с помощью этой грамматики.

```
[the, cat, scares, a, mouse] % Кошка пугает мышь
[the, mouse, hates, the, cat] % Мышь ненавидит кошку
[the, mouse, scares, the, mouse] % Мышь пугает мышь
```

Теперь введем в эту грамматику существительные и глаголы во множественном числе, чтобы иметь возможность формировать фразы наподобие [ the, mice, hate, the, cats] (Мыши ненавидят кошек) следующим образом:

```
noun --> [cats].
noun --> [mice].
verb --> [scare].
verb --> [hate].
```

Грамматика, дополненная таким образом, позволяет сформировать желаемую фразу. Но, к сожалению, она, кроме того, формирует также некоторые нежелательные, *неправильные* английские фразы, например, такие:

```
[the, mouse, hate, the cat] % Мышь ненавидят кошку
```

Проблема заключается в том, что в грамматику входит следующее правило:

```
sentence --> noun_phrase, verb_phrase.
```

Оно указывает, что для формирования фразы можно соединить любую именную конструкцию с любой глагольной конструкцией. Но в английском языке и многих других языках именная и глагольная конструкции во фразе не являются независимыми — они должны быть согласованы в числе. Обе эти конструкции должны быть одновременно заданы либо в единственном, либо во множественном числе. Такая особенность языка называется *контекстной зависимостью* (context dependency). Возможность применения той или иной конструкции зависит от контекста, в котором она встретилась. Контекстные зависимости не могут быть учтены непосредственно в грамматиках BNF, но могут быть легко обработаны с помощью грамматик DCG, с использованием расширения, предусмотренного в системе DCG, в отличие от BNF, а именно параметров, которые разрешается дополнительно указывать в нетерминальных символах грамматики. Например, в качестве параметра именной и глагольной конструкции можно указать число (Number) следующим образом:

```
noun_phrase(Number)
verb_phrase(Number)
```

После добавления этого параметра появляется возможность легко откорректировать грамматику, рассматриваемую в качестве примера, чтобы обеспечить согласование в числе между именной и глагольной конструкцией, как показано ниже.

```
sentence(Number) --> noun_phrase(Number), verb_phrase(Number).
verb_phrase(Number) --> verb(Number), noun_phrase(Number1).
noun_phrase(Number) --> determiner(Number), noun(Number).
noun(singular) --> [mouse]. % Единственное число
noun(plural) --> [mice]. % Множественное число
...
```

В процессе считывания интерпретатором Prolog правил DCG и преобразования в предложение Prolog параметры нетерминальных символов добавляются к обычным параметрам двух списков на основании того соглашения, что эти два списка вводятся в последнюю очередь. Таким образом, правило

```
sentence(Number) --> noun_phrase(Number), verb_phrase(Number).
```

преобразуется в такое предложение:

```
sentence(Number, List1, Rest) :-
 noun_phrase(Number, List1, List2),
 verb_phrase(Number, List2, Rest).
```

Теперь необходимо откорректировать соответствующим образом вопросы к системе Prolog, чтобы включить в них дополнительные параметры, как показано ниже.

```
?- sentence(plural, [the, mice, hate, the, cats], []).
yes
?- sentence(plural, [the, mice, hates, the, cats], []).
no
?- sentence(plural, [the, mouse, hates, the, cat], []).
no
?- sentence(Number, [the, mouse, hates, the, cat], []).
Number = singular
?- sentence(singular, [the, What, hates, the, cat], []).
What = cat;
What = mouse;
no
```

## Упражнения

**23.1.** Преобразуйте в стандартное предложение Prolog следующее правило DCG:

```
s-->[a],s,[b].
```

**21.2.** Напишите процедуру Prolog

```
translate(DCGrule, PrologClause)
```

которая преобразует заданное правило DCG в соответствующее предложение Prolog.

**21.3.** Одно из правил DCG в грамматике, описывающей движения робота, выглядит таким образом:

```
move --> step, move.
```

Если это правило будет заменено правилом

```
move --> move, step.
```

то язык, определяемый грамматикой, модифицированной соответствующим образом, остается тем же самым. Но соответствующая процедура распознавания на языке Prolog становится иной. Проанализируйте, в чем состоит различие. Каково преимущество первоначальной грамматики? Каким образом эти две грамматики обрабатывают следующий вопрос:

```
?- move([up, left], []).
```

## 21.2. Обработка смысла

### 21.2.1. Формирование деревьев синтаксического анализа

Вначале проиллюстрируем понятие дерева синтаксического анализа на примере. В соответствии с определением рассматриваемой грамматики синтаксический анализ фразы

```
[the, cat, scares, the, mice]
```

осуществляется с помощью дерева синтаксического анализа, как показано на рис. 21.2. Некоторые части этой фразы называются *конструкциями*; таковыми являются части, которые соответствуют нетерминальным символам в дереве синтаксического анализа. В рассматриваемом примере [ the, mice] представляет собой конструкцию, соответствующую нетерминальному символу noun\_phrase (именная конструкция), а [ scares, the, mice] — это конструкция, соответствующая нетерминальному символу verb\_phrase (глагольная конструкция). Как показано на рис. 21.2, дерево синтаксического анализа фразы содержит в качестве поддеревьев деревья синтаксического анализа отдельных конструкций.



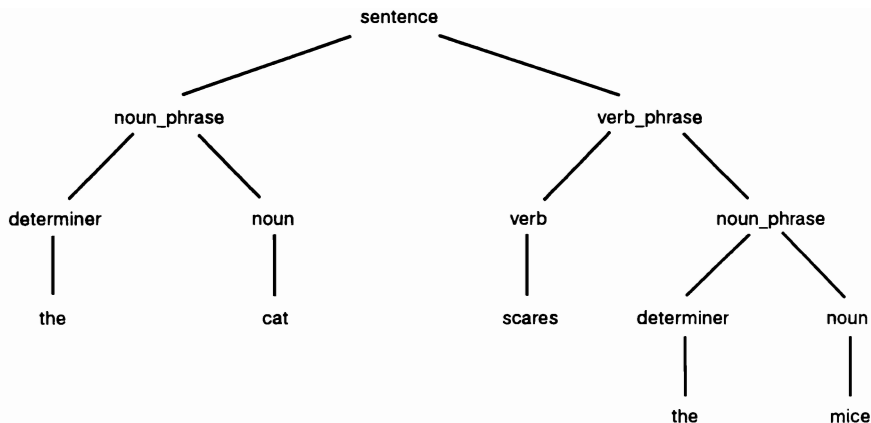


Рис. 21.2. Дерево синтаксического анализа фразы “the cat scares the mice” (кошка пугает мышей)

Ниже приведено определение понятия *дерева синтаксического анализа*. Дерево синтаксического анализа любой грамматической конструкции представляет собой дерево со следующими свойствами.

1. Все листья дерева обозначены терминальными символами грамматики.
2. Все внутренние узлы дерева обозначены нетерминальными символами; корень дерева обозначен нетерминальным символом, который соответствует рассматриваемой конструкции.
3. Родительско-дочерние отношения в дереве определяются правилами грамматики; например, если грамматика содержит правило

$s \rightarrow p, q, r$ .

то дерево может содержать узел  $s$ , дочерними узлами которого являются узлы  $p, q$  и  $r$  (рис. 21.3).

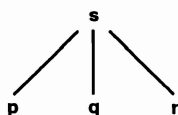


Рис. 21.3. Родительский узел  $s$  и дочерние узлы  $p, q$  и  $r$

Иногда удобно иметь дерево синтаксического анализа, явно представленное в программе для выполнения над ним некоторых вычислений, например, для извлечения смысла фразы. Дерево синтаксического анализа можно легко сформировать с использованием в качестве параметров нетерминальных символов в системе обозначений DCG. Дерево синтаксического анализа может быть легко представлено с помощью термина Prolog, функтором которого является корень дерева, а параметрами — поддеревья дерева. Например, дерево синтаксического анализа для именной конструкции “the cat” можно представить следующим образом:

```
noun_phrase(determiner(the), noun(cat))
```

Для формирования дерева синтаксического анализа грамматика DCG может быть модифицирована путем добавления к каждому нетерминальному символу дерева его синтаксического анализа в виде параметра. Например, дерево синтаксического анализа именной конструкции в рассматриваемой грамматике имеет такую форму:

```
noun_phrase(DetTree, NounTree)
```

где `DetTree` и `NounTree` — деревья синтаксического анализа определяющего слова (*determiner*) и существительного (*noun*). После введения этих деревьев синтаксического анализа в качестве параметров в правило грамматики, касающееся именных конструкций, будет получено следующее модифицированное правило:

```
noun_phrase(noun_phrase(DetTree, NounTree)) -->
 determiner(DetTree), noun(NounTree).
```

Это правило можно описать словами, как показано ниже.

Именная конструкция, имеющая дерево синтаксического анализа `noun_phrase( DetTree, NounTree)`, состоит из таких компонентов:

- определяющее слово, деревом синтаксического анализа которого является `DetTree`;
- существительное, деревом синтаксического анализа которого служит `NounTree`.

Теперь вся рассматриваемая грамматика может быть откорректирована соответствующим образом. Для обеспечения согласования в числе можно оставить число в качестве первого параметра и ввести дерево синтаксического анализа в виде второго параметра. Часть модифицированной грамматики показана ниже.

```
sentence(Number, sentence(NP, VP)) -->
 noun_phrase(Number, NP),
 verb_phrase(Number, VP).
```

```
verb_phrase(Number, verb_phrase(Verb, NP)) -->
 verb(Number, Verb),
 noun_phrase(Number1, NP).
```

```
noun_phrase(Number, noun_phrase(Det, Noun)) -->
 determiner(Det),
 noun(Number, Noun).
```

```
determiner(determiner(the)) --> [the].
noun(singular, noun(cat)) --> [cat].
noun(plural, noun(cats)) --> [cats].
```

В процессе чтения интерпретатором *Prolog* эта грамматика автоматически преобразуется в стандартную программу *Prolog*. Первое правило грамматики, приведенное выше, преобразуется в следующее предложение:

```
sentence(Number, sentence(NP, VP), List, Rest) :-
 noun_phrase(Number, NP, List, Rest0),
 verb_phrase(Number, VP, Rest0, Rest).
```

В соответствии с этим вопрос к системе *Prolog* для синтаксического анализа некоторой фразы должен быть задан в соответствующем формате, например:

```
?- sentence(Number, ParseTree, [the, mice, hate, the, cat], []).
Number = plural
ParseTree = sentence(noun_phrase(determiner(the), noun(mice)),
 verb_phrase(verb(hate), noun_phrase(determiner(the),
 noun(cat)))))
```

## 21.2.2. Применение деревьев синтаксического анализа для извлечения смысла

Грамматики *Prolog* чрезвычайно хорошо приспособлены для трактовки смысла фраз, особенно на естественных языках. Для обработки смысла фраз могут использоваться параметры, которые закреплены за нетерминальными символами грамматики. Один из подходов к решению задачи извлечения смысла предусматривает использование двух этапов.

1. Формирование дерева синтаксического анализа заданной фразы.
2. Обработка дерева синтаксического анализа для определения смысла.

Безусловно, такой подход может применяться на практике, только если синтаксическая структура, представленная с помощью дерева синтаксического анализа, отражает также семантическую структуру; это означает, что и синтаксическая, и семантическая декомпозиции имеют аналогичные структуры. В таком случае смысл фразы можно составить из смыслов синтаксических конструкций, на которые раскладывается фраза в результате синтаксического анализа.

Проиллюстрируем этот двухэтапный метод на несложном примере. Для упрощения снова будем рассматривать движения робота. Грамматика с описанием движений робота, позволяющая вырабатывать дерево синтаксического анализа, приведена ниже.

```
move(move(Step)) --> step(Step).
move(move(Step, Move)) --> step(Step), move(Move).
step(step(up)) --> [up].
step(step(down)) --> [down].
```

Теперь определим *смысл движения* как расстояние между положением робота перед выполнением движения и после него. Пусть каждый шаг имеет длину 1 мм в положительном или отрицательном направлении. Поэтому смысл движения “up up down up” составляет  $1 + 1 - 1 + 1 = 2$ . Расстояние перемещения (distance) в результате выполнения движения можно вычислить на основании дерева синтаксического анализа движения, как показано на рис. 21.4. Ниже приведены соответствующие арифметические расчеты.

```
distance('up up down up') =
distance('up') + distance('up down up') = 1 + 1 = 2
```

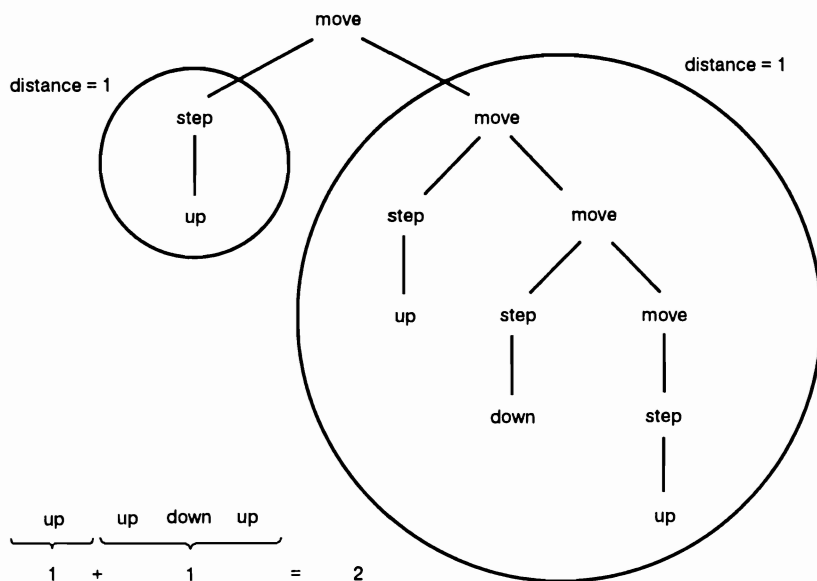


Рис. 21.4. Извлечение смысла движения как расстояния перемещения в результате его выполнения

Ниже приведена процедура, которая определяет смысл движения (как соответствующего расстояния перемещения) на основании дерева синтаксического анализа движения.

```
% meaning(ParseTree, Value)
meaning(move(Step, Move), Dist) :-
 meaning(Step, D1),
 meaning(Move, D2),
```

Dist is D1 + D2.

```
meaning(move(Step), Dist) :-
 meaning(Step, Dist).
```

```
meaning(step(up), 1).
meaning(step(down), -1).
```

Этой процедурой можно воспользоваться, например, для вычисления смысла движения “up up down up” следующим образом:

```
?- move(Tree, [up, up, down, up], [], meaning(Tree, Dist)).
Dist = 2
Tree = move(step(up), move(step(up), move(step(down), move(step(up)))))
```

## Упражнение

**21.4.** Эта грамматика и процедура meaning могут использоваться для решения противоположной задачи, т.е. для поиска движения, позволяющего переместить манипулятор робота на заданное расстояние, например, как показано ниже.

```
?- move(Tree, Move, []), meaning(Tree, 5).
```

Обсудите перспективы применения такого подхода.

## 21.2.3. Совместное применение синтаксических и семантических конструкций в системе обозначений DCG

Система обозначений DCG фактически позволяет включать определение смысла непосредственно в грамматику, что дает возможность избежать необходимости промежуточного формирования дерева синтаксического анализа. Это — еще одно расширение системы обозначений, поддерживаемое DCG, которое является очень удобным для решения указанной задачи. Это расширение позволяет вставлять обычные цели Prolog в грамматические правила. Такие цели должны быть заключены в фигурные скобки, чтобы их можно было отличить от других символов грамматики. Таким образом, любое выражение, которое заключено в фигурные скобки, будучи обнаруженным интерпретатором Prolog, выполняется как обычная цель Prolog. Такое выполнение может, например, предусматривать арифметические вычисления.

Это средство может использоваться для подготовки такого варианта грамматики движения робота, который позволяет непосредственно совмещать операции извлечения смысла с синтаксическими конструкциями. Для этого необходимо добавить определение смысла; иными словами, ввести расстояние перемещения в результате некоторого движения в качестве параметра нетерминальных символов move и step, например, следующим образом:

```
move(Dist)
```

Этот предикат теперь обозначает синтаксическую конструкцию move, смыслом которой является Dist. Соответствующая грамматика приведена ниже.

```
move(D) --> step(D).
move(D) --> step(D1), move(D2), {D is D1 + D2}.
step(1) --> [up].
step(-1) --> [down].
```

Второе правило этой грамматики может служить примером применения системы обозначений, в которой используются фигурные скобки. Это правило можно описать словами, как показано ниже.

Движение, смыслом которого является D, состоит из следующего:

- шаг, смысловым значением которого является D1;
- движение, имеющее смысловое значение D2, при выполнении которого должно также быть удовлетворено отношение  $D \text{ is } D1 + D2$ .

В действительности способ обработки семантических конструкций путем включения правил формирования смысла непосредственно в грамматику DCG является настолько удобным, что с его помощью часто удается полностью избежать необходимости применять промежуточный этап формирования дерева синтаксического анализа. Благодаря исключению этого промежуточного этапа программы становятся, как принято их называть, *свернутыми* (collapsed). Обычно в результате этого достигаются такие преимущества, что программы, кроме того, становятся более краткими и эффективными. Но этот подход характеризуется также некоторыми недостатками: свернутая программа может оказаться менее наглядной, менее гибкой и более сложной для корректировки.

В качестве еще одной иллюстрации данного метода совместного применения синтаксических и смысловых конструкций рассмотрим немного более интересный вариант грамматики движений робота. Предположим, что для управления манипулятором робота могут использоваться два приводных механизма —  $g_1$  и  $g_2$ . После получения команды перемещения на один шаг роботом, использующим приводной механизм  $g_1$ , робот выполняет движение на 1 мм вверх или вниз, а при использовании приводного механизма  $g_2$  происходит движение на расстояние 2 мм. Предположим, что вся программа управления роботом состоит из команд переключения на тот или иной приводной механизм ( $g_1$  или  $g_2$ ) и команд перемещения на один шаг, а оканчивается командой `stop`. Примеры таких программ приведены ниже.

```
stop
g1 up up stop
g1 up up g2 down up stop
g1 g1 g2 up up g1 up down up g2 stop
```

Последняя программа имеет следующий смысл (т.е. расстояние перемещения в результате движения):

$$\text{Dist} = 2 * (1 + 1) + 1 * (1 - 1 + 1) = 5$$

Для обработки таких программ управления роботом существующая грамматика движений робота должна быть расширена с помощью следующих правил:

```
prog(0) --> [stop].
prog(Dist) --> gear(_), prog(Dist).
prog(Dist) --> gear(G), move(D), prog(Dist1), {Dist is G * D + Dist1}.
gear(1) --> [g1].
gear(2) --> [g2].
```

## 21.3. Определение смысла фраз на естественном языке

### 21.3.1. Представление смысла простых фраз с помощью логических высказываний

*Определение смысла естественного языка* — это чрезвычайно сложная проблема, которая продолжает оставаться темой интенсивных исследований. Окончательное решение проблемы формализации всего синтаксиса и семантики языка, подобного английскому, далеко еще не найдено. Но (относительно) простые подмножества естественных языков были успешно формализованы, а затем реализованы в виде действующих программ.

При определении смысла языка необходимо в первую очередь найти ответ на вопрос о том, каким образом должен быть представлен смысл. Безусловно, существует много разных вариантов решения этой задачи, и наиболее приемлемый вариант зависит от конкретного приложения. Поэтому возникает не менее важный вопрос — для

чего может использоваться смысл, извлеченный из текста на естественном языке? Типичной областью его применения является доступ с помощью естественного языка к базе данных. В эту область входит поиск ответов на вопросы, сформулированные на естественном языке, касающиеся информации в базе данных, а также обновление базы данных путем ввода новой информации, которая извлечена из входных данных, сформулированных на естественном языке. В таком случае целевым представлением для процесса извлечения смысла должен быть язык операторов запроса и обновления базы данных.

Общепризнано, что для представления смысла фраз на естественном языке хорошо подходят логические высказывания. В целом язык логики является более мощным по сравнению с формальными языками доступа к базе данных; он фактически не только охватывает любые формальные языки доступа к базе данных, но и позволяет учитывать более тонкие семантические закономерности. В данном разделе будет показано, каким образом могут быть сформированы варианты интерпретации простых фраз естественного языка на языке логики с помощью системы обозначений DCG. Эти логические интерпретации будут представлены в виде термов Prolog. Здесь рассматриваются только некоторые интересные идеи, поэтому многие дополнительные сведения, необходимые для более общего описания, будут опущены. Более полное описание данной темы выходит за рамки настоящей книги.

Для начала удобнее всего рассмотреть некоторые фразы и конструкции естественного языка и попытаться выразить их смысл в форме логических высказываний. Вначале рассмотрим фразу "John paints" (Джон рисует). Наиболее приемлемый способ представления смысла этой фразы в форме логического высказывания (в виде термина Prolog) состоит в следующем:

```
paints(john)
```

Обратите внимание на то, что в данном случае глагол "рисует" является непереходным, поэтому соответствующий предикат `paints` имеет только один параметр.

Следующим примером фразы является "John likes Annie" (Джон любит Энни). Формализованное выражение смысла этой фразы может состоять в следующем:

```
likes(john, annie)
```

Глагол "любит" является переходным, и в соответствии с этим предикат `likes` имеет два параметра. Теперь попытаемся определить смысл подобных простых фраз с помощью правил DCG. Прежде всего начнем исключительно с синтаксиса, а затем будем постепенно вводить в эти правила смысловые интерпретации. Ниже приведена грамматика, которая позволяет легко охватить синтаксис этих фраз, рассматриваемых в качестве примера.

```
sentence --> noun_phrase, verb_phrase. % фраза состоит из именной
 % и глагольной конструкции
noun_phrase --> proper_noun. % Имя собственное
verb_phrase --> intrans_verb. % Непереходный глагол
verb_phrase --> trans_verb, noun_phrase. % Переходный глагол
intrans_verb --> [paints].
trans_verb --> [likes].
proper_noun --> [john].
proper_noun --> [annie].
```

Теперь введем в эти правила смысл. Начнем с более простых категорий (с существительного и глагола), а затем перейдем к более сложным грамматическим категориям. Приведенные выше примеры показывают, что требуется ввести следующие определения. Во-первых, необходимо определить смысл имени собственного. В данном случае смысл имени собственного `john` — это просто `john`:

```
proper_noun(john) --> [john].
```

Во-вторых, необходимо определить смысл непереходного глагола, такого как глагол "рисует". Эта задача немного сложнее. Предикат с обозначением смысла этого глагола можно представить следующим образом:

```
paints(X)
```

где  $X$  — переменная, значение которой становится известным только из контекста, иными словами, из именной конструкции. В соответствии с этим правило DCG для предиката `paints` состоит в следующем:

```
intrans_verb(paints(X)) --> [paints].
```

Теперь рассмотрим такой вопрос: как сформировать из этих двух смысловых значений, `john` и `paints( X)`, требуемый смысл всей фразы `paints( john)`? Для этого необходимо обеспечить, чтобы параметр  $X$  предиката `paints` стал равным `john`.

На данном этапе целесообразно проанализировать применяемые операции с помощью графической схемы, как показано на рис. 21.5. На этом рисунке показано, как смысл отдельных конструкций фразы преобразуется в смысл всей фразы. Для достижения эффекта распространения семантических (смысловых) значений конструкций фразы можно определить, что предикаты `noun_phrase` и `verb_phrase` получают свои семантические значения из предикатов `proper_noun` и `intrans_verb`, как показано ниже.

```
noun_phrase(NP) --> proper_noun(NP).
verb_phrase(VP) --> intrans_verb(VP).
```

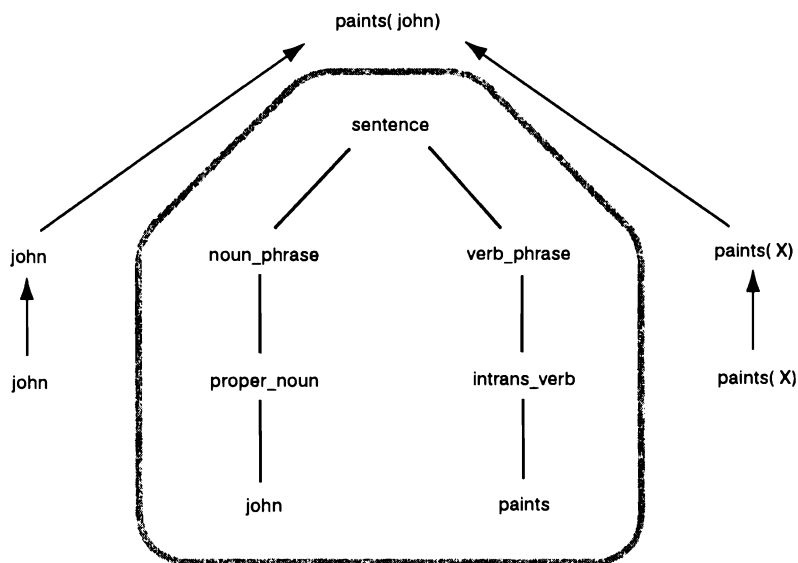


Рис. 21.5. Дерево синтаксического анализа фразы “John paints”, в котором показано, какое смысловое значение соответствует каждому из узлов. Логический смысл каждой конструкции фразы закреплен за соответствующим нетерминальным узлом в дереве. Стрелки показывают, каким образом накапливаются смысловые значения конструкций

Остается определить смысл всей фразы. Ниже показаны результаты первой попытки решения этой задачи.

```
sentence(S) --> noun_phrase(NP), verb_phrase(VP), {compose(NP, VP, S)}.
```

В соответствии с первоначальным замыслом цель `compose( NP, VP, S)` должна была аккумулировать смысловые значения именной конструкции `john` и глагольной конструкции `paints( X)`. Введем такое понятие, что  $X$  — является деятелем в предикате `paints( X)`, и определим следующее отношение:

```
actor(VP, Actor)
```

В этом отношении Actor — это деятель в смысловом значении глагольной конструкции VP. В таком случае одно из предложений процедуры actor принимает следующий вид:

```
actor(paints(X), X).
```

После определения отношения actor появляется возможность создать композицию смысловых значений именной и глагольной конструкций следующим образом:

```
compose(NP, VP, VP) :- % Смысловым значением фразы является VP, где
 actor(VP, NP). % деятелем actor в конструкции VP служит NP
```

Очевидно, что первая попытка определения смысла всей фразы оказалась удачной, но можно найти более удобный метод. Мы сможем избежать необходимости введения дополнительных предикатов compose и actor, если сделаем параметр X в терме paints( X) “видимым” вне термина, чтобы он стал доступным для конкретизации. Этой цели можно достичь, переопределив смысловые значения предикатов verb\_phrase и intrans\_verb таким образом, чтобы X стал дополнительным параметром, как показано ниже.

```
intrans_verb(Actor, paints(Actor)) --> [paints].
verb_phrase(Actor, VP) --> intrans_verb(Actor, VP).
```

Благодаря этому параметр Actor становится легко доступным и обеспечивает возможность сформулировать более простое определение смысла фразы следующим образом:

```
sentence(VP) --> noun_phrase(Actor), verb_phrase(Actor, VP).
```

Эта конструкция вынуждает параметр “деятель” (Actor) смыслового значения глагола стать равным смыслу именной конструкции.

Такой метод обеспечения доступа к частям грамматических конструкций, выражающим смысловые значения, представляет собой один из приемов, часто используемых при включении смысловых значений в правила DCG. По сути, этот метод действует следующим образом. Смысл одной из конструкций фразы определяется в “эскизной” форме, например paints( SomeActor). Такой предикат определяет общую форму смыслового значения, но оставляет часть этого значения неконкретизированной (в данном случае это — переменная SomeActor). Такая неконкретизированная переменная служит в качестве *слота* (своего рода “гнезда”), который может быть заполнен позже с учетом смысловых значений других синтаксических конструкций в данном контексте. Подобное заполнение слотов может осуществляться с помощью операции согласования интерпретатора Prolog. Но для упрощения реализации этой операции слоты необходимо сделать видимыми, добавив их в качестве дополнительных параметров к нетерминальным символам. Введем следующее соглашение, касающееся порядка расположения этих параметров, — в начале должны находиться “видимые слоты” смыслового значения синтаксической конструкции, а за ними должно следовать само смысловое значение; в качестве примера применения такого соглашения можно указать предикат verb\_phrase( Actor, VPmeaning).

Этот метод можно также применить к переходным глаголам следующим образом. Смыслом глагола “любит” является likes( Somebody, Something) (кто-то любит что-то), где Somebody и Something — слоты, которые должны быть видимыми извне. Таким образом, может быть составлено следующее правило:

```
trans_verb(Somebody, Something, likes(Somebody, Something)) --> [likes].
```

Глагольная конструкция с переходным глаголом содержит именную конструкцию, которая предоставляет значение для Something, поэтому имеет место такое правило:

```
verb_phrase(Somebody, VP) -->
 trans_verb(Somebody, Something, VP), noun_phrase(Something).
```

Выше в данном разделе были представлены некоторые основные идеи в области обработки смысловых значений, но приведенные правила DCG позволяют обрабатывать лишь самые простые языковые фразы. А если именные конструкции содержат



такие *определяющие слова*, как “а” (неопределенный артикль) и “every” (местоимение “каждый”), то способы выражения смысла становятся гораздо более сложными. Эта тема рассматривается в следующем разделе.

## 21.3.2. Смысл определяющих слов “а” и “every”

Обработка фраз, содержащих такие определяющие слова, как “а”, становится намного более сложной по сравнению с фразами, которые описаны в предыдущем разделе. В качестве примера рассмотрим такую фразу: “A man paints” (Некий мужчина рисует). Было бы большой ошибкой считать, что эта фраза имеет смысл `paints( man)`, поскольку в ней фактически сказано: “Существует некий мужчина, который рисует”, а не “Мужчинам свойственно рисовать”. В форме логического высказывания эта фраза может быть представлена следующим образом:

Существует X, такой, что X – мужчина и X рисует.

Переменную X, представленную в составе такого логического высказывания, принято называть *переменной, к которой применяется квантор существования* (в виде глагола “существует”). Примем за основу способ представления переменных, определяемых таким образом, с помощью следующего термина Prolog:

```
exists(X, man(X) and paints(X))
```

Первым параметром этого термина является переменная X, которая подлежит определению с помощью квантора существования, а оператор `and` считается объявленным как инфиксный оператор следующим образом:

```
:- op(100, xfy, and).
```

Хотя на первый взгляд это может показаться неожиданным, но синтаксическим объектом, который диктует необходимость применения такой сложной логической интерпретации, является определяющее слово “а”. Поэтому слово “а” в определенном смысле доминирует над всей этой фразой. Чтобы лучше понять, каким образом формируется это смысловое значение, рассмотрим именную конструкцию “a man”. Она имеет следующий смысл:

Существует некий X, такой, что X – мужчина.

Но во фразах, где появляется именная конструкция “a man”, таких как “a man paints”, мы всегда стремимся сообщить об этом мужчине что-то еще (в данном случае — не только то, что *он существует*, но и что *он рисует*). Поэтому наиболее подходящая форма для выражения смысла именной конструкции “a man” состоит в следующем:

```
exists(X, man(X) and Assertion)
```

где *Assertion* — некоторое утверждение об X. Такое утверждение, касающееся X, зависит от контекста, т.е. от глагольной конструкции, которая следует за именной конструкцией “a man”. Переменная *Assertion* конкретизируется только после того, как становится известен контекст, в котором она появляется.

Аналогичный ход рассуждений позволяет найти правильную формулировку смысла определяющего слова “а”. Это определяющее слово показывает следующее:

Существует некий X, такой, что

X обладает некоторым свойством (например, `man( X)`) и является истинным еще какое-то утверждение, касающееся X (например, `paints( X)`).

Эта формулировка может быть представлена в виде термина Prolog таким образом:

```
exists(X, Property and Assertion)
```

Обе переменные, *Property* и *Assertion*, представляют собой слоты для смысловых значений, извлекаемых из контекста и помещаемых в эти слоты. Для упрощения переноса смысловых значений из других синтаксических конструкций в контексте можно сделать видимыми некоторые части смысловых значений определяющего

слова “а”, как описано в предыдущем разделе. Подходящее правило DCG для определяющего слова “а” приведено ниже.

```
determiner(X, Prop, Assn, exists(X, Prop and Assn)) --> [a].
```

Логический смысл крошечного определяющего слова “а” оказался на удивление сложным. Еще одно определяющее слово, “every”, можно проанализировать аналогичным образом. Рассмотрим фразу “Every woman dances” (Каждая женщина танцует). Логическая интерпретация этой фразы приведена ниже.

Для всех X, если X — женщина, то X танцует.

Эта формулировка может быть представлена с помощью следующего термина Prolog:

```
all(X, woman(X) => dances(X))
```

где “=>” — инфиксный оператор, обозначающий логическую импликацию. Таким образом, определяющее слово “every” указывает на наличие смыслового значения, которое имеет такую эскизную структуру:

```
all(X, Property => Assertion)
```

Правило DCG, которое описывает смысл определяющего слова “every” и делает слоты в этой эскизной структуре видимыми, приведено ниже.

```
determiner(X, Prop, Assn, all(X, Prop => Assn)) --> [every].
```

Теперь, после описания смысловых значений определяющих слов мы можем сосредоточиться на изучении того, как смысловые значения этих определяющих слов объединяются со смысловыми значениями других синтаксических конструкций в контексте и приводят в конечном итоге к получению смыслового значения всей фразы. Первое представление об этом можно получить, снова вернувшись к рассмотрению фразы “A man paints”, которая имеет такой смысл:

```
exists(X, man(X) and paints(X))
```

Смысл определяющего слова “а” уже был выявлен следующим образом:

```
exists(X, Prop and Assn)
```

После сравнения этих двух смысловых значений сразу же становится очевидно, что основная структура смыслового значения этой фразы продиктована определяющим словом. Смысл всей этой фразы можно скомпоновать, как показано на рис. 21.6; начнем с эскизного смыслового значения, предписанного определяющим словом “а” таким образом:

```
exists (X, Prop and Assn)
```

Затем переменная Prop становится конкретизированной значением именной конструкции, а Assn — значением глагольной конструкции. Основная структура смыслового значения этой фразы получена из именной конструкции. Обратите внимание на то, что в этом состоит отличие от более простой грамматики, описанной в предыдущем разделе, в которой смысловая структура фразы была продиктована глагольной конструкцией. Снова применяя метод, позволяющий сделать видимыми некоторые части смыслового значения, можно сформулировать отношения между смысловыми значениями, показанными на рис. 21.6, с помощью следующих правил DCG:

```
sentence(S) --> noun_phrase(X, Assn, S), verb_phrase(X, Assn).
noun_phrase(X, Assn, S) --> determiner(X, Prop, Assn, S), noun(X, Prop).
verb_phrase(X, Assn) --> intrans_verb(X, Assn).
intrans_verb(X, paints(X)) --> [paints].
determiner(X, Prop, Assn, exists(X, Prop and Assn)) --> [a].
noun(X, man(X)) --> [man].
```

После ввода этой грамматики в интерпретатор Prolog может быть задан вопрос для формирования смысла фразы “A man paints”, как показано ниже.

```
?- sentence(S, [a, man, paints], []).
S = exists(X, man(X) and paints(X))
```

Здесь ответ системы Prolog немного отредактирован таким образом, что имя переменной наподобие \_123, сформированное системой Prolog, заменено на X.

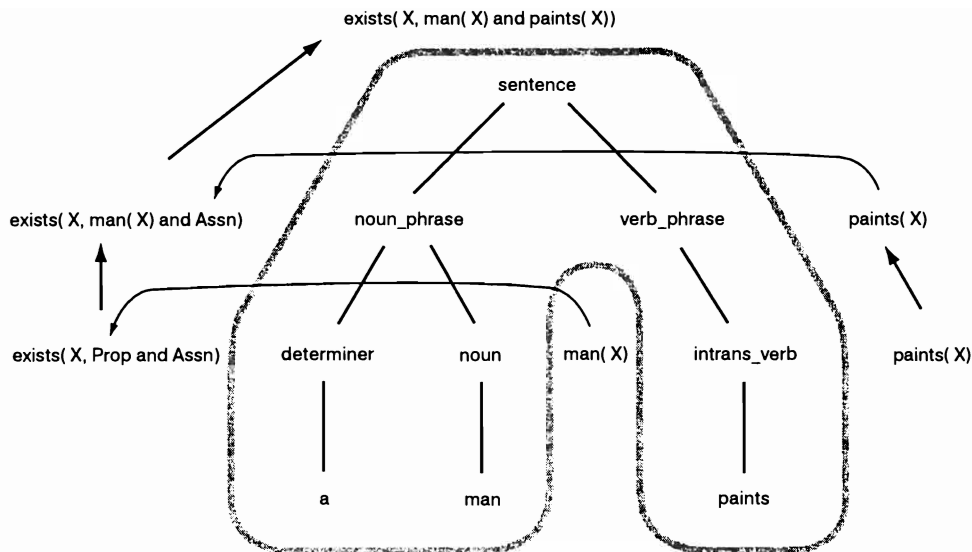


Рис. 21.6. Накопление смысловых значений для определения смысла фразы “A man paints”. Общая форма смыслового значения этой фразы продиктована определяющим словом “a”. Стрелки показывают направление переноса смысловых значений из одной синтаксической конструкции в другую

Грамматика, приведенная в предыдущем разделе, позволяет обрабатывать такие фразы, как “John paints”. После внесения изменений в эту грамматику необходимо обеспечить, чтобы новый вариант грамматики, позволяющий обрабатывать фразы типа “A man paints”, был способен также обрабатывать более простые фразы наподобие “John paints”. Для этого требуется включить в новую именную конструкцию средства определения смысла имен собственных. Ниже приведены правила, которые позволяют решить такую задачу.

```
proper_noun(john) --> [john].
noun_phrase(X, Assn, Assn) --> proper_noun(X).
```

Последнее из этих правил просто указывает, что весь смысл именной конструкции такого рода совпадает со значением второго “видимого слота”, т.е. Assn. Значение этого слота может быть получено из контекста (из глагольной конструкции), как при обработке следующего вопроса:

```
?- sentence(S, [john, paints], []).
S = paints(john)
```

## Упражнение

- 21.5. Изучите, как происходит процесс формирования смысла фразы “John paints” при использовании модифицированного варианта рассматриваемой грамматики. По сути, должно происходить следующее: смысловое значение, полученное из глагольной конструкции, становится смысловым значением именной конструкции, а последнее в конечном итоге становится смыслом фразы.

### 21.3.3. Обработка относительных предложений

В роли грамматических определений существительных могут выступать относительные предложения (relative clause), например, как во фразе “Every man that paints admires Monet” (Каждый мужчина, который рисует, обожает Моне). В этой

фразе синтаксическая конструкция “every man that paints” представляет собой именную конструкцию, в которой “that paints” является относительным предложением. Для охвата такой синтаксической конструкции можно переопределить грамматические правила, относящиеся к именной конструкции, следующим образом:

```
noun_phrase --> determiner, noun, rel_clause.
rel_clause --> [that], verb_phrase.
rel_clause --> [].
```

% Пустое относительное предложение

Теперь рассмотрим смысл подобных именных конструкций. С точки зрения логики фраза “Every man that paints admires Monet” означает следующее:

Для всех X,  
если X - мужчина и X рисует,  
то X обожает Моне.

Эту формулировку можно представить в виде такого термина Prolog:

```
all(X, man(X) and paints(X) => admires(X, monet))
```

Здесь предполагается, что оператор “and” связывает сильнее, чем оператор “=>”. Таким образом, подходящая форма для представления логического смысла именной конструкции “every man that paints” является следующей:

```
all(X, man(X) and paints(X) => Assn)
```

В целом эта форма имеет вид

```
all(X, Prop1 and Prop2 => Assn)
```

Здесь смысл переменной Prop1 определяется именной конструкцией, Prop2 — глагольной конструкцией относительного предложения, а Assn — глагольной конструкцией фразы. Правила DCG для этой именной конструкции, которые обеспечивают соответствующее определение смысла, приведены ниже.

```
rel_clause(X, Prop1, Prop1 and Prop2) --> [that], verb_phrase(X, Prop2).
noun_phrase(X, Assn, S) --> determiner(X, Prop12, Assn, S),
noun(X, Prop1),
rel_clause(X, Prop1, Prop12).
```

Для охвата того случая, когда относительное предложение является пустым, необходимо добавить такое правило:

```
rel_clause(X, Prop1, Prop1) --> [].
```

В листинге 21.1 приведено полное определение DCG с возможностями, описанными в этом разделе, включая определяющие слова “a” и “every”, а также относительные предложения. Эта грамматика способна извлекать логический смысл фраз, подобных приведенным ниже.

```
John paints.
Every man that paints admires Monet.
Annie admires every man that paints.
Every woman that admires a man that paints likes Monet.
```

#### Листинг 21.1. Правила DCG, которые обеспечивают обработку синтаксиса и смысла небольшого подмножества естественного языка

```
:- op(100, xfy, and).
:- op(150, xfy, =>).

sentence(S) -->
 noun_phrase(X, P, S), verb_phrase(X, P).

noun_phrase(X, P, S) -->
 determiner(X, P12, P, S), noun(X, P1), rel_clause(X, P1, P12).

noun_phrase(X, P, P) -->
 proper_noun(X).

verb_phrase(X, P) -->
```

```

trans_verb(X, Y, Pl), noun_phrase(Y, Pl, P).

verb_phrase(X, P) -->
 intrans_verb(X, P).

rel_clause(X, Pl, Pl and P2) -->
 [that], verb_phrase(X, P2).

rel_clause(X, Pl, Pl) --> [].

determiner(X, Pl, P, all(X, Pl => P)) --> [every].

determiner(X, Pl, P, exists(X, Pl and P)) --> [a].

noun(X, man(X)) --> [man].

noun(X, woman(X)) --> [woman].

proper_noun(john) --> [john].

proper_noun(annie) --> [annie].

proper_noun(monet) --> [monet].

trans_verb(X, Y, likes(X, Y)) --> [likes].

trans_verb(X, Y, admires(X, Y)) --> [admires].

intrans_verb(X, paints(X)) --> [paints].

% Некоторые проверки

test1(M) :-
 sentence(M, [john,paints],[]).

test2(M) :-
 sentence(M, [a, man, paints], []).

test3(M) :-
 sentence(M, [every,man,that,paints,admires,monet],[]).

test4(M) :-
 sentence(M, [annie,admires,every,man,that,paints],[]).

test5(M) :-
 sentence(M, [every,woman,that,admires,a,man,that,paints,likes,monet],[]).

```

Например, эти фразы могут быть переданы на обработку в рассматриваемой грамматике в виде следующих вопросов:

```

?- sentence(Meaning1, [every, man, that, paints, admires, monet], []).
Meaning1 = all(X, man(X) and paints(X) => admires(X, monet))

?- sentence(Meaning2, [annie, admires, every, man, that, paints], []).
Meaning2 = all(X, man(X) and paints(X) => admires(annie, X))

?- sentence(Meaning3, [every, woman, that, admires, a, man, that, paints,
 likes, monet], []).
Meaning3 = all(X, woman(X) and exists(Y, (man(Y) and paints(Y) and
 admires(X, Y)) => likes(X, monet))

```

Еще одна интересная проблема касается использования смысловых значений, извлеченных из текста на естественном языке, для формулировки ответов на вопросы. Например, рассмотрим, как изменить рассматриваемую программу, чтобы после обработки заданных фраз она могла отвечать на вопросы такого типа: “Does Annie

admire anybody who admires Monet?” (Обожает ли Энни кого-либо, кто обожает Моне). Ответ на этот вопрос логически следует из приведенных выше фраз, и нам достаточно просто заставить данную программу проводить некоторые необходимые рассуждения. В общем нам требуется определенный механизм автоматического доказательства теорем, обеспечивающий дедуктивный вывод ответов из смысловых значений, представленных в форме логических высказываний. Безусловно, с точки зрения практики удобнее всего использовать в качестве такого механизма автоматического доказательства теорем саму систему Prolog. Для этого достаточно преобразовать логические смысловые значения в эквивалентные предложения Prolog. В целом выполнение такого упражнения требует определенных усилий, но в некоторых случаях подобное преобразование является тривиальным. Ниже приведены некоторые легко преобразуемые смысловые значения, записанные в виде предложений Prolog.

```
paints(john).
```

```
admires(X, monet) :-
 man(X),
 paints(X).
```

```
admires(annie, X) :-
 man(X),
 paints(X).
```

Представленный выше в качестве примера вопрос “Does Annie admire anybody who admires Monet?” можно преобразовать в следующий запрос Prolog:

```
?- admires(annie, X), admires(X, monet).
X = john
```

## Упражнения

**21.6.** Сформулируйте в виде логических высказываний смысл следующих фраз:

- а) Mary knows all important artists (Мэри знает всех известных художников).
- б) Every teacher who teaches French and studies music understands Chopin (Каждый учитель, который учит французскому и изучает музыку, понимает Шопена).
- в) A charming lady from Florida runs a beauty shop in Sydney (Очаровательная леди из Флориды содержит салон красоты в Сиднее).

**21.7.** Грамматика, приведенная в листинге 21.1, может быть также вызвана на выполнение в противоположном направлении. В таком случае по заданному смыслу вырабатывается фраза с этим смыслом. Например, можно попытаться ввести запрос, обратный запросу test5, приведенному в листинге 21.1, таким образом:

```
?- M = all(X,woman(X) and exists(Y,(man(Y) and paints(Y)) and
 admires(X,Y))
=> likes(X,monet)), sentence(M, S, []).
```

Первый ответ системы Prolog состоит в следующем:

```
S = [every,woman,that,admires,a,man,that,paints,likes,monet]
M = all(_022C,woman(_022C) and exists(_02FC,(man(_02FC) and
 paints(_02FC))
 and admires(_022C,_02FC)) => likes(_022C,monet))
```

Он соответствует ожидаемому. Но если мы потребуем предоставить еще одно решение, то получим такой непредвиденный результат:

```
M = all(monet,woman(monet) and exists(_0364,(man(_0364) and paints(_0364))
 and admires(monet,_0364)) => likes(monet,monet))
S = [monet,likes,every,woman,that,admires,a,man,that,paints] % Моне любит любую
 % женщину, которая восхищается неким мужчиной, который рисует
```

Объясните, почему он был получен, и предложите внести определенное изменение в грамматику для предотвращения его возникновения. Подсказка: проблема состоит в том, что эта грамматика допускает возможность согласования переменной, на которую распространяется действие квантора всеобщности (например,  $X$  в предикате  $\text{all}(X, \dots)$ ), с именем собственным (*monet*); такую ситуацию можно легко предотвратить.

**21.8.** Дополните грамматику, приведенную в листинге 21.1, чтобы она позволяла обрабатывать составные фразы с союзами “if”, “then”, “and”, “or”, “neither”, “nor” и т.д. В качестве примера таких фраз можно привести следующие:

|                                            |                             |
|--------------------------------------------|-----------------------------|
| John paints and Annie sings.               | % Джон рисует и Энни поет   |
| If Annie sings then every teacher listens. | % Если Энни поет, то каждый |
|                                            | % учитель слушает           |

## Проект

Модифицируйте грамматику, приведенную в листинге 21.1, чтобы она представляла смысл фраз в виде непосредственно выполняемых предложений Prolog. Напишите программу, которая читает фразы на естественном языке в обычном текстовом формате (а не в виде списков; см. подходящую программу в главе 6) и вводит их смысловые значения в базу данных системы в виде предложений Prolog. Дополните эту грамматику, чтобы она обрабатывала простые вопросы на естественном языке, что привело бы в конечном итоге к созданию законченной диалоговой системы для небольшого подмножества естественного языка. Можно также предусмотреть использование этой грамматики для выработки фраз с заданным значением, предоставляемых пользователю в качестве ответов на естественном языке.

## Резюме

- Стандартные *грамматические системы обозначений*, такие как BNF, могут быть простейшим образом преобразованы в *систему обозначений DCG* (Definite Clause Grammar — грамматика определенных предложений). Грамматика, заданная в системе обозначений DCG, может считываться и вызываться на выполнение непосредственно интерпретатором Prolog, осуществляющим функции системы распознавания для языка, определенного с помощью этой грамматики.
- Система обозначений DCG позволяет вводить параметры в *нетерминальные символы грамматики*. Это дает возможность учитывать в грамматике контекстные зависимости и непосредственно включать в нее определения семантических значений языка.
- Разработаны интересные *грамматики DCG*, которые охватывают синтаксис и значение нетривиальных подмножеств естественного языка.

## Дополнительные источники информации

Определения синтаксиса и значения естественного языка в виде правил DCG, приведенные в этой главе, соответствуют классической статье [121]. В превосходной книге [120] приведено описание многих других достижений в этой области, включая изящное определение связи между соответствующими математическими основами задачи определения смысла и ее реализации на языке Prolog. В [93] рассматривается грамматика DCG, предназначенная для интерпретации запросов к базам данных на естественном языке. В [156] приведен пример применения грамматики DCG для трансляции языка программирования. Упражнение 21.8 заимствовано из [77], где также приведено решение. Многие темы, касающиеся обработки естественного языка, рассматриваются в [5], [38] и [57].

# Ведение игры

*В этой главе...*

|                                                                                     |     |
|-------------------------------------------------------------------------------------|-----|
| 22.1. Игры с полной информацией, с двумя участниками                                | 532 |
| 22.2. Принцип минимакса                                                             | 534 |
| 22.3. Альфа-бета-алгоритм: эффективная реализация принципа минимакса                | 537 |
| 22.4. Программы, основанные на принципе минимакса: усовершенствования и ограничения | 541 |
| 22.5. Типовые знания и механизм советов                                             | 543 |
| 22.6. Программа ведения шахматного эндшпиля на языке Advice Language 0              | 546 |

В данной главе рассматриваются методы ведения игр с полной информацией, с двумя участниками, таких как шахматы. В интересных играх деревья возможных продолжений являются слишком сложными для того, чтобы в них можно было выполнить исчерпывающий поиск, поэтому требуются другие подходы. Один из соответствующих методов основан на использовании принципа минимакса, который эффективно реализован в виде альфа-бета-алгоритма. Кроме этого стандартного метода, в этой главе показано, как разработать программу, основанную на применении подхода к внедрению в программу игры в шахматы знаний, представленных в виде образов на языке советов Advice Language. Этот довольно подробный пример позволяет дополнительно продемонстрировать, насколько хорошо язык Prolog подходит для реализации систем, основанных на использовании знаний.

## 22.1. Игры с полной информацией, с двумя участниками

В настоящей главе рассматриваются игры, которые относятся к играм с полной информацией, с двумя участниками. В качестве примеров игр такого типа можно назвать шахматы, шашки и го. В этих играх два игрока делают ходы поочередно и имеют полную информацию о текущей ситуации в игре. Таким образом, этому определению не соответствует большинство карточных игр. Игра заканчивается после того, как достигается позиция, которая оценивается по правилам игры как *заклучительная*, например мат в шахматах. Правила определяют также, каким является результат игры после ее завершения в *заклучительной* позиции.

Подобная игра может быть представлена в виде дерева игры. Узлы такого дерева соответствуют ситуациям игры, а дуги — ходам. Первоначальная ситуация игры представляет собой корневой узел, а листья дерева соответствуют *заклучительным* позициям.

В большинстве игр такого типа результатом игры может стать победа, поражение или ничья. Но в данной главе рассматриваются игры только с двумя результатами:



победа и поражение. В тех играх, где возможна ничья, результаты также можно свести к двум ситуациям: победа и отсутствие победы. Двух участников рассматриваемых игр принято называть *своим игроком* и *чужим игроком*. Свой игрок может выиграть в незаключительной позиции своего хода, если имеется допустимый ход, ведущий к выигранной позиции. С другой стороны, незаключительная позиция чужого хода является выигрышной для своего игрока, если все допустимые ходы ведут от этой позиции к выигрышным позициям. Эти правила соответствуют представлению задач в виде деревьев AND/OR, которые рассматривались в главе 13. В табл. 22.1 показано соответствие между понятиями, которые применяются при описании деревьев AND/OR и игр.

**Таблица 22.1. Соответствие между понятиями, которые применяются при описании игр и деревьев AND/OR**

| Игры                            | Деревья AND/OR                             |
|---------------------------------|--------------------------------------------|
| Игровые позиции                 | Проблемы                                   |
| Завершающая выигранная позиция  | Целевой узел, тривиально решаемая проблема |
| Завершающая проигранная позиция | Неразрешимая проблема                      |
| Выигранная позиция              | Решенная проблема                          |
| Позиция своего хода             | Узел OR                                    |
| Позиция чужого хода             | Узел AND                                   |

Очевидно, что для организации поиска в деревьях игры можно применить многие понятия, которые обеспечивают поиск в деревьях AND/OR.

Простая программа, позволяющая определить, является ли выигрышной некоторая позиция своего хода, приведена ниже.

```
won(Pos) :-
 terminalwon(Pos). % Заключительная выигранная позиция
won(Pos) :-
 not terminallost(Pos),
 move(Pos, Pos1), % Допустимый ход, ведущий к позиции Pos1
 not (move(Pos1, Pos2), % Ни один ход противника не ведет
 not won(Pos2)). % к невыигрышной позиции
```

Правила игры встроены в предикат `move( Pos, Pos1)`, который вырабатывает допустимые ходы, и в предикаты `terminalwon( Pos)` и `terminallost( Pos)`, предназначенные для распознавания заключительных позиций, являющихся выигрышными или проигрышными согласно правилам игры. Последнее из приведенных выше правил гласит, что не существует чужого хода, который ведет к невыигрышной позиции, поскольку в этом правиле используются два оператора отрицания. Иными словами, в ситуации выигрыша все чужие ходы должны вести к выигрышной позиции.

Как и другие аналогичные программы поиска в графах AND/OR, приведенная выше программа основана на использовании стратегии поиска в глубину. Кроме того, эта программа не предотвращает циклический переход от одной позиции к другой. Это может вызвать проблемы, поскольку правила некоторых игр не исключают возможности повторения позиций. Но такое повторение часто только внешне выглядит как результативное продолжение игры, а в действительности служит признаком безвыходной ситуации. Например, по правилам шахмат после трехкратного повторения позиции в игре может быть объявлена ничья.

Приведенная выше программа демонстрирует лишь основной принцип. Но для практической реализации программ ведения таких сложных игр, как шахматы или го, необходимы гораздо более мощные методы. Из-за большой комбинаторной сложности этих игр приведенный выше примитивный алгоритм поиска, который останавливается только в заключительных позициях игры, становится полностью непригод-

ным. На рис. 22.1 эта мысль иллюстрируется применительно к шахматам. Пространство поиска астрономических размеров в дереве этой игры включает приблизительно  $10^{120}$  позиций. Иногда можно услышать такие возражения, что в различных местах дерева, показанного на рис. 22.1, встречаются одинаковые позиции. Тем не менее доказано, что количество различных позиций на шахматной доске намного превосходит возможности любых компьютеров, которые могут быть созданы в обозримом будущем.

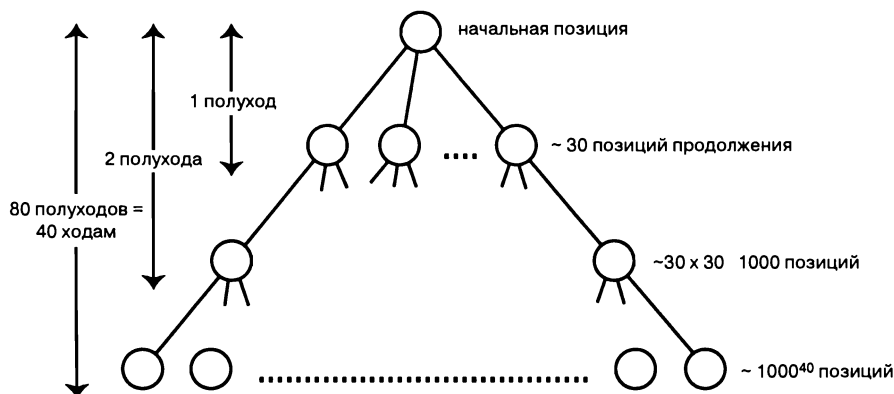


Рис. 22.1. Сложность деревьев игр в шахматах. Приведенные здесь оценки основаны на предположении, что из любой шахматной позиции может быть сделано приблизительно 30 допустимых ходов, а заключительные позиции возникают на глубине 40 ходов. Каждый ход состоит из 2 полуходов (по 1 полуходу от каждого участника)

## Проект

Напишите программу ведения какой-то простой игры (такой как *н.и.м.*) с использованием прямолинейного подхода, основанного на поиске в графе AND/OR.

## 22.2. Принцип минимакса

Поскольку для создания программ ведения многих интересных игр невозможно применить исчерпывающий поиск в деревьях игры, разработаны другие методы, которые основаны на поиске только в некоторой части дерева игры. К ним относится стандартный метод, применяемый при ведении на компьютере игр (таких как шахматы), который основан на принципе минимакса. Поиск в дереве игры осуществляется только до определенной глубины, как правило, на несколько ходов, а затем осуществляется оценка концевых узлов этого дерева поиска с помощью некоторой функции оценки. Идея состоит в том, что оценка этих заключительных позиций поиска происходит без выполнения поиска за их пределами, что позволяет сэкономить время. После этого оценки заключительных позиций распространяются вверх по дереву поиска в соответствии с принципом минимакса. Это позволяет определить оценки позиций для всех позиций в дереве поиска. Затем в игре фактически выполняется ход, который ведет от первоначальной, корневой позиции к ее наиболее перспективному преемнику (согласно этим оценкам).

Обратите внимание на то, что в приведенном выше описании проводится различие между понятиями “дерево игры” и “дерево поиска”. *Дерево поиска* обычно представляет собой лишь (верхнюю) часть дерева игры, иными словами, часть, явно формируемую в процессе поиска. Поэтому заключительные позиции поиска не обязательно должны быть заключительными позициями игры.

При этом многое зависит от функции оценки. Такая функция в большинстве интересных игр должна представлять собой эвристическую функцию, позволяющую оценить шансы на выигрыш с точки зрения одного из игроков. Чем выше эта оценка, тем больше шансов на выигрыш имеет игрок, а чем меньше это значение, тем больше шансы на выигрыш у его противника. Поскольку один из игроков получает возможность достичь позиции с высокой оценкой, а другой вынужден довольствоваться низкой оценкой, эти два игрока соответственно именуются как MAX и MIN. Каждый раз, когда ход должен сделать игрок MAX, он выбирает ход, который в максимальной степени увеличивает оценку своей позиции. В противоположность этому игрок MIN должен выбрать ход, который сводит к минимуму оценку позиции своего противника. Если известны значения позиций низкого уровня в дереве поиска, то такой принцип (называемый минимаксом) позволяет определить значения всех других позиций в дереве поиска, как показано на рис. 22.2. На этом рисунке уровни позиций, в которых должен ходить игрок MAX, чередуются с позициями, в которых право сделать ход передается игроку MIN. Значения позиций нижнего уровня определяются с помощью функции оценки. Стоимости внутренних узлов можно рассчитать, поднимаясь снизу вверх, от одного уровня к другому до тех пор, пока не будет достигнут корневой узел. На рис. 22.2 результирующая стоимость корневого узла равна 4, поэтому наилучшим ходом для игрока MAX в позиции а является а-б. Наилучшим ответом для игрока MIN является b-d и т.д. Такая последовательность позиций в игре называется также *основным вариантом*. Основной вариант определяет для обоих участников игру, оптимальную в соответствии с принципом минимакса. Обратите внимание на то, что стоимость позиций вдоль основного варианта не изменяется. В соответствии с этим *правильными* являются те ходы, которые позволяют сохранить стоимость игры.

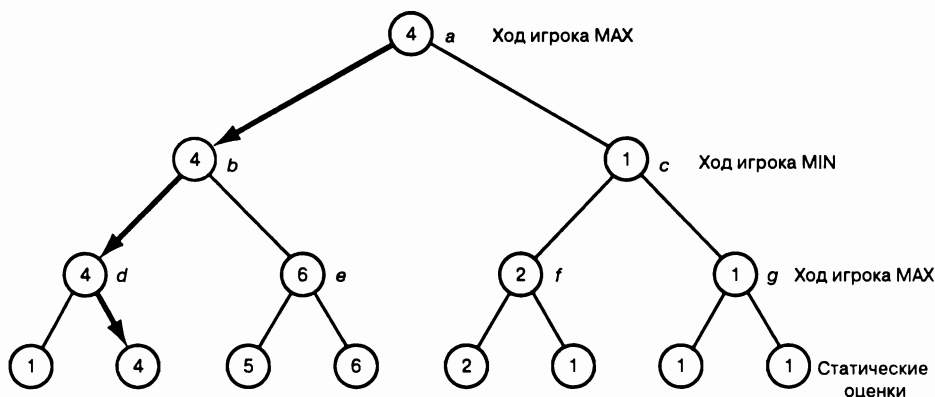


Рис. 22.2. Статические значения (нижний уровень) и зафиксированные минимаксные значения в дереве поиска. Ходы, обозначенные жирными стрелками, составляют основной вариант, т.е. игру обеих сторон, оптимальную в соответствии с принципом минимакса (описание этого дерева игры на языке Prolog приведено в листинге 22.1)

#### Листинг 22.1. Описание дерева игры, представленного на рис 22.2, на языке Prolog

```
% moves(Position, PositionList): возможные ходы

moves(a, [b,c]).
moves(b, [d,e]).
moves(c, [f,g]).
moves(d, [i,j]).
moves(e, [k,l]).
moves(f, [m,n]).
```

```

moves(g, [o,p]).

% min_to_move(Pos): игрок MIN должен сделать ход в позиции Pos

min_to_move(b).
min_to_move(c).

% max_to_move(Pos): игрок MAX должен сделать ход в позиции Pos

max_to_move(a).
max_to_move(d).
max_to_move(e).
max_to_move(f).
max_to_move(g).

% staticval(Pos, Value): переменная Value - статическое значение для Pos

staticval(i, 1).
staticval(j, 4).
staticval(k, 5).
staticval(l, 6).
staticval(m, 2).
staticval(n, 1).
staticval(o, 1).
staticval(p, 1).

```

При анализе процесса игры необходимо учитывать различия между значениями нижнего уровня и зафиксированными значениями. Первые называются *статическими*, поскольку они получены с помощью *статической* функции оценки в отличие от зафиксированных значений, получаемых *динамически* в процессе распространения статических значений вверх по дереву.

Правила распространения значений можно формализовать следующим образом. Обозначим статическое значение позиции  $P$  следующим образом:

$v(P)$

а зафиксированное значение таким образом:

$V(P)$

Предположим, что  $P_1, \dots, P_n$  — допустимые позиции-преемники  $P$ . В таком случае отношение между статическими и зафиксированными значениями можно определить, как показано ниже.

- Если  $P$  — заключительная позиция в дереве поиска ( $n = 0$ ), то  $V(P) = v(P)$ .
- Если  $P$  — позиция хода игрока MAX, то  $V(P) = \max_i V(P_i)$ .
- Если  $P$  — позиция хода игрока MIN, то  $V(P) = \min_i V(P_i)$ .

Программа Prolog, которая вычисляет минимаксное зафиксированное значение для заданной позиции, приведена в листинге 22.2. Основным отношением в этой программе является следующее:

```
minimax(Pos, BestSucc, Val)
```

где  $Val$  — минимаксное значение позиции  $Pos$ , а  $BestSucc$  — позиция, являющаяся наилучшим преемником позиции  $Pos$  (ход, который должен быть выполнен для достижения  $Val$ ). Отношение

```
moves(Pos, PosList)
```

соответствует правилам допустимого хода игры;  $PosList$  — это список допустимых позиций — преемников  $Pos$ . Предполагается, что предикат `moves` не достигает успеха, если  $Pos$  — заключительная позиция поиска (лист дерева поиска). А такое отношение:

```
best(PosList, BestPos, BestVal)
```

выбирает “наилучшую” позицию BestPos из списка возможных позиций PosList. Переменная BestVal представляет собой значение BestPos и поэтому также значение Pos. В данном случае “наилучший” рассматривается как максимальный или минимальный, в зависимости от того, чья сейчас очередь хода.

## Листинг 22.2. Простая реализация принципа минимакса

```
% minimax(Pos, BestSucc, Val):
% переменная Pos - это позиция, Val - ее минимаксное значение; наилучший
ход,
% сделанный в позиции Pos, ведет к позиции BestSucc

minimax(Pos, BestSucc, Val) :-
 moves(Pos, PosList), !, % Ходы, допустимые в позиции Pos,
 % составляют список PosList

 best(PosList, BestSucc, Val)
 ;
 staticval(Pos, Val). % Позиция Pos не имеет преемников,
 % поэтому для нее применяется статическая оценка

best([Pos], Pos, Val) :-
 minimax(Pos, _, Val), !.

best([Pos1 | PosList], BestPos, BestVal) :-
 minimax(Pos1, _, Val1),
 best(PosList, Pos2, Val2),
 betterof(Pos1, Val1, Pos2, Val2, BestPos, BestVal).

betterof(Pos0, Val0, Pos1, Val1, Pos0, Val0) :- % Позиция Pos0 лучше, чем Pos1
 min_to_move(Pos0), % В позиции Pos0 должен ходить
 % игрок MIN
 Val0 > Val1, ! % Игрок MAX предпочитает более
 % высокое значение
 ;
 max_to_move(Pos0), % В позиции Pos0 должен ходить игрок MAX
 Val0 < Val1, !. % Игрок MIN предпочитает более низкое значение

betterof(Pos0, Val0, Pos1, Val1, Pos1, Val1). % В противном случае позиция
 % Pos1 - лучше, чем Pos0
```

## 22.3. Альфа-бета-алгоритм: эффективная реализация принципа минимакса

Программа, приведенная в листинге 22.2, предусматривает систематический перебор в режиме поиска в глубину всех позиций в дереве поиска, вплоть до заключительных, и вычисление статических оценок для всех заключительных позиций этого дерева. Но обычно не требуется выполнять всю эту работу для того, чтобы правильно вычислить минимаксное значение корневой позиции. В соответствии с этим можно применить более экономичный алгоритм поиска. Усовершенствование этого алгоритма может быть основано на такой идее: предположим, что имеются два альтернативных хода; после того как стало полностью ясно, что один из них хуже другого, больше не требуется знать, насколько именно он хуже, чтобы принять правильное решение. Например, такой принцип можно использовать для сокращения объема поиска в дереве, показанном на рис. 22.2. В данном случае процесс поиска осуществляется, как описано ниже.

1. Начать с позиции а.
2. Перейти вниз к позиции б.

3. Перейти вниз к позиции d.

4. Выбрать максимальное значение для одного из преемников d, что приводит к получению  $V(d) = 4$ .

5. Выполнить возврат к позиции b, а затем спуститься вниз к позиции e.

6. Рассмотреть первого преемника e, значение которого равно 5. В данный момент для игрока MAX (который должен сделать ход в позиции e) гарантировано, по меньшей мере, значение 5 в позиции e, если даже не учитывать того, что из позиции e исходят другие (возможно лучшие) альтернативы. Эта оценка является достаточной для игрока MIN, чтобы понять, что в узле b альтернатива e хуже, чем d, даже не зная точного значения e.

На этом основании можно пренебречь вторым преемником позиции e и присвоить позиции e ориентировочное значение 5. Но такая замена точного значения приближенным не влияет на расчетное значение b и, следовательно, на значение a.

На этой идее основан знаменитый **альфа-бета-алгоритм** (называемый также **алгоритмом альфа-бета-отсечения**), применяемый для эффективной реализации принципа минимакса. На рис. 22.3 показано действие альфа-бета-алгоритма на примере дерева, приведенного на рис. 22.2. Как показано на рис. 22.3, некоторые из зафиксированных значений являются приблизительными. Но, несмотря на такую замену точных значений приблизительными, дерево содержит достаточно информации для точного определения значения корневого узла. В примере, показанном на рис. 22.3, применение альфа-бета-алгоритма позволяет сократить сложность поиска с восемью статических оценок (как было первоначально на рис. 22.2) до пяти статических оценок.

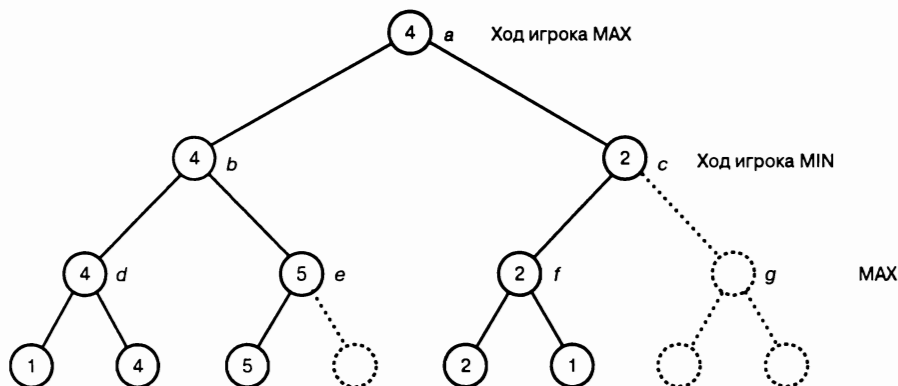


Рис. 22.3. Дерево, приведенное на рис. 22.2, поиск в котором осуществляется с помощью альфа-бета-алгоритма. При этом поиске отсекаются узлы, обозначенные штриховыми линиями, поэтому общий объем поиска уменьшается. В результате некоторые из зафиксированных значений становятся неточными (в частности, это характерно для узлов c, e, f; см рис. 22.2). Но, несмотря на такое снижение точности, дерево содержит достаточно информации для правильного определения значения корневого узла и поиска основного варианта

Как было указано выше, основная идея альфа-бета-отсечения состоит в том, чтобы найти “достаточно хороший” ход (не обязательно лучший), который вполне подходит для принятия правильного решения. Эту идею можно формализовать, наложив два ограничения на зафиксированное значение позиции, которые обычно обозначаются как Alpha и Beta. Эти ограничения имеют следующее значение: Alpha — это минимальное значение, достижение которого уже гарантировано для игрока MAX, а Beta — максимальное значение, которого может надеяться достичь игрок MAX. Если позиция рассматривается с точки зрения игрока MIN, то Beta — это наихудшее значение для

MIN, достижение которого гарантировано для игрока MIN. Поэтому фактическое значение (которое должно быть найдено) лежит в пределах от Alpha до Beta. Если оказалось, что некоторая позиция имеет значение, лежащее за пределами интервала Alpha-Beta, то этого достаточно, чтобы определить, что данная позиция не относится к основному варианту, даже не зная точного значения этой позиции. Точное значение позиции необходимо знать только в том случае, если оно находится между Alpha и Beta. Понятие *достаточно хорошего* зафиксированного значения  $V(P, \text{Alpha}, \text{Beta})$  позиции P по отношению к ограничениям Alpha и Beta можно определить формально как любое значение, которое удовлетворяет следующим требованиям:

$V(P, \text{Alpha}, \text{Beta}) < \text{Alpha}$ , если  $V(P) < \text{Alpha}$   
 $V(P, \text{Alpha}, \text{Beta}) = V(P)$ , если  $\text{Alpha} < V(P) < \text{Beta}$   
 $V(P, \text{Alpha}, \text{Beta}) > \text{Beta}$ , если  $V(P) > \text{Beta}$

Безусловно, мы можем всегда вычислить точное значение  $V(P)$  корневой позиции P, задавая для нее пределы следующим образом:

$V(P, -\text{бесконечность}, +\text{бесконечность}) = V(P)$

В листинге 22.3 приведена реализация альфа-бета-алгоритма на языке Prolog. Основным отношением в этой программе является следующее:

`alphabeta( Pos, Alpha, Beta, GoodPos, Val)`

где GoodPos — *достаточно хороший* преемник Pos, поэтому его значение Val соответствует требованиям, указанным выше, следующим образом:

`Val = V( Pos, Alpha, Beta)`

### Листинг 22.3. Реализация альфа-бета-алгоритма

% Альфа-Бета-алгоритм

```
alphabeta(Pos, Alpha, Beta, GoodPos, Val) :-
 moves(Pos, PosList), !,
 boundedbest(PosList, Alpha, Beta, GoodPos, Val);
 staticval(Pos, Val). % Статическое значение позиции Pos

boundedbest([Pos | PosList], Alpha, Beta, GoodPos, GoodVal) :-
 alphabeta(Pos, Alpha, Beta, _, Val),
 goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal).

goodenough([], _, _, Pos, Val, Pos, Val) :- !. % Другой кандидат отсутствует

goodenough(_, Alpha, Beta, Pos, Val, Pos, Val) :-
 min_to_move(Pos), Val > Beta, ! % Верхняя граница, достигнутая
 % максимизирующим оператором
 ;
 max_to_move(Pos), Val < Alpha, !. % Нижняя граница, достигнутая
 % минимизирующим оператором

goodenough(PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
 newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta), % Уточнить границы
 boundedbest(PosList, NewAlpha, NewBeta, Pos1, Val1),
 betterof(Pos, Val, Pos1, Val1, GoodPos, GoodVal).

newbounds(Alpha, Beta, Pos, Val, Val, Beta) :-
 min_to_move(Pos), Val > Alpha, !. % Максимизирующий оператор
 % увеличил нижнюю границу

newbounds(Alpha, Beta, Pos, Val, Alpha, Val) :-
 max_to_move(Pos), Val < Beta, !. % Минимизирующий оператор
 % уменьшил верхнюю границу

newbounds(Alpha, Beta, _, _, Alpha, Beta). % В противном случае границы
% не изменились
```

```

betterof(Pos, Val, Pos1, Val1, Pos, Val) :- % Позиция Pos лучше, чем Pos1
 min_to_move(Pos), Val > Val1, !
 ;
 max_to_move(Pos), Val < Val1, !.

betterof(_, _, Pos1, Val1, Pos1, Val1). % В противном случае лучше
 % позиция Pos1

```

---

## Процедура

```
boundedbest(PosList, Alpha, Beta, GoodPos, Val)
```

находит достаточно хорошую позицию GoodPos в списке PosList таким образом, что зафиксированное значение Val позиции GoodPos является достаточно хорошим приближением по отношению к Alpha и Beta.

Альфа-бета-интервал может стать более узким (но ни в коем случае не более широким!) при больших по глубине рекурсивных вызовах альфа-бета-процедуры. Отношение

```
newbounds(Alpha, Beta, Pos, Val, NewAlpha, NewBeta)
```

определяет новый интервал ( NewAlpha, NewBeta), который всегда меньше или равен старому интервалу ( Alpha, Beta). Поэтому по мере перехода на более глубокие уровни дерева поиска пределы Alpha-Beta сужаются и позиции этих глубоких уровней оцениваются в более узких пределах. Из-за сужения интервалов количество приближенных оценок увеличивается и поэтому отсечение поддеревьев в дереве становится все более интенсивным. В связи с этим возникает интересный вопрос: "Какой объем работы позволяет сэкономить альфа-бета-алгоритм по сравнению с программой исчерпывающего поиска по принципу минимакса, приведенной в листинге 22.2?" Эффективность поиска с использованием альфа-бета-алгоритма зависит от того, в каком порядке происходит перебор позиций. Выгоднее всего в первую очередь рассматривать самые сильные ходы для каждого из игроков. Можно легко показать на примерах, что если порядок перебора окажется неудачным, то альфа-бета-процедуре потребуются перебрать все позиции, рассматриваемые при исчерпывающем минимаксном поиске. Это означает, что в худшем случае альфа-бета-алгоритм не имеет преимуществ над исчерпывающим поиском по принципу минимакса. Но если порядок является благоприятным, экономия усилий может оказаться весьма значительной. Предположим, что  $N$  — количество заключительных позиций поиска, статически оцениваемых с помощью исчерпывающего минимаксного алгоритма. Было доказано, что в наилучшем случае, когда вначале всегда рассматривается самый сильный ход, альфа-бета-алгоритм требует статической оценки лишь  $\sqrt{N}$  позиций.

Добавим к этому, что данный результат подтвержден на практике при проведении шахматных турниров. На соревнованиях шахматной программе обычно дается определенное количество времени для вычисления следующего хода в игре, а глубина, на которую программа может выполнить поиск, зависит от этого количества времени. В наилучшем случае альфа-бета-алгоритм оказался способным выполнять поиск на глубину, в два раза большую по сравнению с исчерпывающим минимаксным поиском. Эксперименты показали, что одна и та же функция оценки, применяемая на большей глубине дерева, обычно позволяет обеспечить более сильную игру.

Результаты экономии усилий при использовании альфа-бета-алгоритма можно также выразить в терминах *эффективного коэффициента ветвления* (среднего количества ветвей, отходящих от каждого внутреннего узла) дерева поиска. Предположим, что дерево игры имеет единообразный коэффициент ветвления  $b$ . В результате отсечения альфа-бета-алгоритм требует поиска лишь в некоторых из ветвей, что фактически приводит к уменьшению коэффициента ветвления. В наилучшем случае такое сокращение находится в пределах от  $b$  до  $\sqrt{b}$ . В программах игры в шахматы фактический коэффициент ветвления в результате альфа-бета-отсечения становится приблизительно равным 6 по сравнению с общим количеством допустимых ходов, примерно равным 30. Менее оптимистические оценки этого результата показывают,



что в шахматах, даже при использовании альфа-бета-алгоритма, углубление поиска на 1 полуход (ход одной из сторон) приводит к увеличению количества заключительных позиций поиска примерно в 6 раз.

## Проект

Рассмотрите игру с двумя участниками (например, некоторую нетривиальную версию игры в крестики и нолики). Напишите отношения с определением игры (допустимых ходов и заключительных игровых позиций) и предложите статическую функцию оценки, применяемую для ведения этой игры с помощью альфа-бета-процедуры.

## 22.4. Программы, основанные на принципе минимакса: усовершенствования и ограничения

Принцип минимакса, наряду с альфа-бета-алгоритмом, является основой многих удачных программ ведения игр; наиболее известными из них являются шахматные программы. Общая схема такой программы состоит в следующем: выполнить альфа-бета-поиск в текущей позиции игры, вплоть до некоторой фиксированной предельной глубины (которая определяется временными ограничениями, налагаемыми правилами соревнований), используя характерную для данной игры функцию оценки для определения значений заключительных позиций поиска; после этого выполнить на игровой доске наилучший ход (в соответствии с альфа-бета-алгоритмом), принять ответ противника и снова приступить к выполнению того же цикла.

Поэтому двумя основными компонентами подобной программы являются альфа-бета-алгоритм и эвристическая функция оценки. Для создания качественной программы ведения такой сложной игры, как шахматы, необходимо ввести ряд усовершенствований в эту основную схему. В данном разделе рассматриваются некоторые стандартные методы.

Многое зависит от функции оценки. Если бы у нас была идеальная функция оценки, то достаточно было бы рассмотреть только непосредственных преемников текущей позиции, фактически устранив при этом поиск. Но в таких играх, как шахматы, любая функция оценки с практически приемлемой вычислительной сложностью обязательно должна быть просто эвристической функцией оценки. Эта оценка основана на “статических” свойствах позиции (в частности, числа фигур на доске) и поэтому в некоторых позициях является более надежной, чем в других. Например, рассмотрим подобную функцию оценки для шахмат, основанную на учете количества фигур (как принято называть их в шахматах — *материала*) и представим себе позицию, в которой белые имеют лишнего коня. Безусловно, что эта функция оценит позицию белых как более благоприятную. Разумеется, такая оценка является вполне приемлемой, если игра развивается спокойно и черные не имеют в своем распоряжении внезапных угроз. С другой стороны, если на следующем ходе черные могут взять ферзя белых, то такая оценка может привести к роковой ошибке, поскольку она не позволяет оценить позицию динамически. Безусловно, что статической оценке можно скорее доверять в спокойной игре, а не в резко изменяющихся позициях острой борьбы, когда каждая из сторон непосредственно угрожает взять фигуры противника. Очевидно, что статическая оценка может использоваться только в спокойных позициях. Поэтому стандартный прием состоит в том, что поиск в опасных позициях должен продлеваться за пределы глубины до тех пор, пока не будет достигнута спокойная позиция. В частности, для применения этого расширения необходимо запрограммировать последовательности взятия фигур с обеих сторон (обмена фигурами) в шахматах.

Еще одним усовершенствованием является эвристическое отсечение. Оно предназначено для достижения большего предела глубины благодаря исключению из рас-

смотрения некоторых менее перспективных продолжений. Этот метод позволяет отсекаать некоторые ветви дерева игры в дополнение к тем, которые были отсечены с помощью самого альфа-бета-алгоритма. Поэтому он связан с тем риском, что некоторые удачные продолжения не будут обнаружены, а значение минимакса вычислено неправильно.

Еще одним важным методом является последовательное углубление. Программа повторно выполняет альфа-бета-поиск, вначале осуществляя поиск на некоторую небольшую глубину, а затем увеличивая предел глубины после каждой итерации. Этот процесс останавливается после достижения определенного предела времени. Затем выполняется ход, который оказался наилучшим в соответствии с результатами наиболее глубокого поиска. Такой метод имеет следующие преимущества.

- Позволяет учитывать контроль времени; после достижения предела времени всегда имеется в запасе некоторый наилучший ход, найденный до сих пор.
- Минимаксные значения предыдущей итерации могут использоваться для предварительного упорядочения позиций в следующей итерации, что позволяет в альфа-бета-алгоритме в первую очередь выполнять поиск среди самых сильных ходов.

При последовательном углублении возникают определенные дополнительные издержки (связанные с повторным поиском в верхних частях дерева игры), но они являются довольно незначительными по сравнению с общим положительным результатом.

Известным недостатком программ, реализованных по этой общей схеме, является *эффект горизонта*. Предположим, что рассматривается шахматная позиция, в которой игрок, за которого выступает программа, неизбежно теряет коня. Но потерю коня можно отдалить, пожертвовав менее ценную фигуру, скажем, пешку. Такая промежуточная жертва может отодвинуть фактическую потерю коня за пределы поиска (за "горизонт", куда не может заглянуть программа). В таком случае, не видя, что конь в конечном итоге все равно будет потерян, программа выбирает этот вариант, а не такое продолжение, в котором потеря коня произойдет быстрее, но без ненужных жертв. Поэтому программа в конечном итоге потеряет и пешку (без необходимости), и коня. Эффект горизонта может быть устранен путем продления поиска вплоть до спокойной позиции.

Но программы, основанные на использовании принципа минимакса, имеют более фундаментальное ограничение, связанное с тем, что в них применяются лишь ограниченная форма знаний, характерных для данной проблемной области. Это становится особенно заметным при сравнении лучших шахматных программ с опытными шахматистами. Мощные программы часто выполняют поиск среди миллионов (и даже большего числа) позиций, прежде чем выбрать очередной ход. А психологические исследования показывают, что шахматисты обычно выполняют поиск лишь среди нескольких десятков позиций, самое большее среди нескольких сотен. Несмотря на такую "низкую производительность" поиска, шахматист все еще может выиграть у программы. Преимущество шахматных мастеров заключается в их знаниях, которые намного превосходят все, что содержится в программах. Соревнования между компьютерами и сильными шахматистами показывают, что невероятное превосходство в вычислительной мощности не всегда может полностью компенсировать нехватку знаний.

Знания, которые могут быть введены в программы, основанные на принципе минимакса, принимают следующие основные формы.

- Функция оценки.
- Эвристические методы отсечения поддеревьев дерева.
- Эвристические методы поиска спокойных позиций.

Функция оценки сводит многочисленные аспекты игровой ситуации к единственному числу, и такое сокращение может иметь отрицательный эффект. В отличие от этого, понимание игровой позиции хорошим игроком является многомерным. Рассмотрим пример из шахмат: некоторая функция оценки может оценить позицию как

равную, просто указав, что ее значение равно 0. А оценка той же позиции шахматистов может быть гораздо более содержательной и указывать на то, как дальше должна развиваться игра. Например, у черных есть лишняя пешка, но белые имеют хорошую атакующую инициативу, которая компенсирует материал, поэтому шансы равны.

В шахматах программы, основанные на принципе минимакса, обычно играют лучше в жестких тактических битвах, когда решающим становится точный расчет форсированных вариантов. Их слабости с большей вероятностью проявляются в спокойных позициях, когда игра этих программ характеризуется отсутствием долговременных планов, играющих решающую роль в медленных, стратегических играх. Из-за отсутствия плана создается впечатление, что на протяжении всей игры программа случайным образом переходит от одной шахматной идеи к другой.

В остальной части этой главы рассмотрим еще один подход к ведению игр, основанный на использовании в программе типовых знаний, которые представлены в виде *советов*. Это позволяет реализовывать в игровой программе целенаправленное плановое поведение.

## 22.5. Типовые знания и механизм советов

### 22.5.1. Цели и ограничения ходов

Метод представления знаний, относящихся к конкретной игре, который рассматривается в этом разделе, основан на использовании языка, принадлежащего к семейству языков советов *Advice Language*. В языке *Advice Language* пользователь определяет декларативным способом, какие идеи должны быть опробованы в ситуациях определенных типов. Идеи формулируются в терминах целей и средств достижения этих целей. После этого интерпретатор языка *Advice Language* находит посредством поиска, какая идея фактически применима в данной ситуации.

Основным понятием в языках *Advice Language* является *элементарный совет* (*piece-of-advice*). Элементарный совет подсказывает, какое очередное действие следует выполнить (или попытаться выполнить) в позиции определенного типа. Вообще говоря, совет выражается в терминах целей, которые должны быть достигнуты, и средств достижения этих целей. Два участника игры именуются *свой игрок* и *чужой игрок*; совет всегда определяет, как должен действовать свой игрок. Каждый элементарный совет состоит из четырех компонентов.

- *Лучшая цель* (*better-goal*) — цель, которая должна быть достигнута.
- *Консервативная цель* (*holding-goal*) — цель, которую нельзя упустить в процессе борьбы за достижение лучшей цели.
- *Ограничения своего хода* (*us-move-constraints*) — предикат с определением ходов, позволяющий выбрать подмножество всех допустимых ходов своего игрока (ходов, которые рассматриваются как наиболее перспективные в отношении достижения указанных целей).
- *Ограничения чужого хода* (*them-move-constraints*) — предикат, позволяющий выбрать ходы, которые могут рассматриваться как наиболее удачные чужим игроком (ходы, способные препятствовать достижению целей, стоящих перед своим игроком).

В качестве простого примера из шахматного эндшпиля, который относится к типу “король и пешка против короля”, рассмотрим непосредственную реализацию идеи превращения пешки в ферзя путем продвижения пешки вперед, без каких-либо дополнительных предосторожностей. Эта идея может быть выражена в форме совета следующим образом.

- Лучшая цель — превращение пешки в ферзя.
- Консервативная цель — предотвращение потери пешки.
- Ограничения своего хода — ход пешкой.
- Ограничения чужого хода — приближение к пешке короля.

## 22.5.2. Выполнимость совета

Принято считать, что элементарный совет является *выполнимым* в данной конкретной позиции, если свой игрок может форсировать достижение лучшей цели, заданной в совете, при следующих условиях.

1. Никогда не нарушается требование достижения консервативной цели.
2. Все выполняемые ходы своего игрока удовлетворяют ограничениям своего хода.
3. Чужому игроку разрешается выполнять только те ходы, которые удовлетворяют ограничениям чужого хода.

С концепцией выполнимости элементарного совета связано понятие форсирующего дерева. Форсирующее дерево представляет собой подробно разработанную стратегию, которая гарантирует достижение лучшей цели при условиях, заданных в элементарном совете. Таким образом, в форсирующем дереве точно указано, какие ходы должен выполнять свой игрок при любом ответе чужого игрока. Формально форсирующее дерево  $T$  для заданной позиции  $P$  и элементарного совета  $A$  представляет собой поддерево дерева игры, такое, что:

- корневым узлом  $T$  является  $P$ ;
- все позиции в  $T$  удовлетворяют консервативной цели;
- все терминальные узлы  $T$  удовлетворяют лучшей цели, и ни один внутренний узел в  $T$  не удовлетворяет лучшей цели;
- имеется точно один ход своего игрока из каждой внутренней позиции своего хода в  $T$ , и этот ход должен удовлетворять ограничениям своего хода;
- $T$  содержит все ходы чужого игрока (которые удовлетворяют ограничениям чужого хода) из каждой незаключительной позиции чужого хода в  $T$ .

Каждый элементарный совет может рассматриваться как определение небольшой игры по особым правилам, которые состоят в следующем. Каждому из соперников разрешается делать ходы, которые удовлетворяют ограничениям их хода; позиция, которая не удовлетворяет консервативной цели, является выигрышной для чужого игрока, а позиция, которая удовлетворяет консервативной цели и лучшей цели, — выигрышна для своего игрока. Незаключительная позиция является выигрышной для своего игрока, если в этой позиции может быть выполнен элементарный совет. В таком случае свой игрок выигрывает, выполняя в ходе игры ходы из соответствующего форсирующего дерева.

## 22.5.3. Объединение элементарных советов в правила и таблицы советов

В языках советов Advice Language отдельные элементарные советы объединяются в законченную схему представления знаний с помощью следующей иерархии. Элементарный совет является частью правила вывода. Совокупность правил вывода представляет собой таблицу советов. Множество таблиц советов структурируется в виде иерархической сети. Каждая таблица советов играет роль специализированного эксперта, позволяющего справиться с некоторой конкретной подпроблемой из всей этой проблемной области. Примером такого специализированного эксперта является таблица советов, содержащая знания о том, как поставить мат в шахматном энд-

шпилю “король и ладья против короля”. Эта таблица вызывается на выполнение, если в игре возникает такое окончание.

Для простоты в данной главе рассматривается сокращенная версия языка Advice Language, в которой разрешается использовать только одну таблицу советов. Назовем эту версию Advice Language 0, или сокращенно AL0. Ниже описана структура языка AL0, синтаксис которого уже был приспособлен для удобства реализации на языке Prolog.

Программа на языке AL0 называется *таблицей советов*. Таблица советов — это упорядоченная коллекция правил вывода. Каждое правило имеет следующую форму:

```
RuleName :: if Condition then AdviceList
```

где Condition — логическое выражение, которое состоит из имен предикатов, соединенных логическими связками and, or, not, а AdviceList — список имен элементарных советов. Ниже приведен пример правила “edge\_rule” (игра в углу), которое относится к эндшпилю “король и ладья против короля”.

```
edge_rule ::
 if their_king_on_edge and our_king_close then [mate_in_2, squeeze,
 approach, keeproom, divide].
```

Это правило гласит, что если в текущей позиции король чужого игрока находится в углу, а король своего игрока приблизился к нему (или точнее, короли отстоят друг от друга меньше чем на четыре клетки), то необходимо попытаться выполнить в указанном порядке приоритетов следующие элементарные советы: “mate\_in\_2”, “squeeze”, “approach”, “keeproom”, “divide”. В этом списке советов заданы элементарные советы в уменьшающемся порядке важности поставленных целей: вначале попытаться поставить мат в два хода; если это невозможно, то попробовать “зажать” короля противника в углу, и т.д. Обратите внимание на то, что при соответствующем определении операторов приведенное выше правило представляет собой синтаксически допустимое предложение Prolog.

Каждый элементарный совет можно определить в виде предложения Prolog в форме

```
advice(AdviceName,
 BetterGoal:
 HoldingGoal:
 Us_Move_Constraints :
 Them_Move_Constraints).
```

Цели — это выражения, состоящие из имен предикатов и логических связей and, or, not. Ограничения ходов также являются выражениями, состоящими из имен предикатов и связей and и then, где and имеет обычное логическое значение, а then предписывает упорядоченность. Например, ограничение хода в следующей форме:

```
MC1 then MC2
```

указывает — вначале рассмотреть те ходы, которые удовлетворяют ограничению MC1, а затем те, которые удовлетворяют MC2.

Например, элементарный совет по выполнению мата в 2 хода в эндшпилю “король и ладья против короля”, оформленный с применением такого синтаксиса, имеет следующий вид:

```
advice(mate_in_2, mate :
 not rooklost:
 (depth = 0) and legal then (depth = 2) and checkmove :
 (depth = 1) and legal).
```

В данном случае лучшая цель — mate (мат), а консервативная цель — not rooklost (предотвращение потери ладьи). В ограничениях своего хода указано, что на глубине 0 (в текущей позиции на доске) следует попытаться выполнить любой допустимый ход, а затем на глубине 2 (на втором ходе своего игрока) пытаться выполнять только ходы с шахом. Глубина измеряется в полуходах. Ограничения чужого хода состоят в следующем: любой допустимый ход на глубине 1.

Затем в процессе игры таблица советов используется в форме повторения до конца игры следующего основного цикла: формирование форсирующего дерева, после этого ведение игры в соответствии с этим деревом до тех пор, пока игра не выйдет за пределы данного дерева; формирование следующего форсирующего дерева и т.д. Форсирующее дерево создается каждый раз следующим образом: берется текущая позиция на доске Pos и просматриваются одно за другим правила в таблице советов; для каждого правила позиция Pos согласуется с предварительным условием этого правила, и поиск прекращается после обнаружения такого правила, что Pos удовлетворяет его предварительному условию. Теперь рассматривается список советов этого правила и последовательно обрабатываются элементарные советы из этого списка до тех пор, пока не будет найден элементарный совет, выполнимый в позиции Pos. Это приводит к получению форсирующего дерева, которое представляет собой подробное описание стратегии, реализуемой на игровой доске.

Следует отметить, что при этом важную роль играет упорядочение правил и элементарных советов. Используемым становится такое первое правило, предварительные условия которого соответствуют текущей позиции. Для любой возможной позиции в таблице советов должно существовать, по меньшей мере, одно правило, предварительное условие которого соответствует этой позиции. Таким образом выбирается список советов. После этого применяется первый найденный выполнимый элементарный совет из этого списка.

Поэтому таблица советов главным образом представляет собой непроцедурную программу. Интерпретатор языка AL0 принимает на входе некоторую позицию и, выполняя действия над таблицей советов, вырабатывает форсирующее дерево, которое определяет игру в данной позиции.

## 22.6. Программа ведения шахматного эндшпиля на языке Advice Language 0

В процессе реализации игровой программы на основе AL0 эту программу удобнее всего разделить на три модуля.

1. Интерпретатор AL0.
2. Таблица советов на языке AL0.
3. Библиотека предикатов, используемых в таблице советов (включая правила игры).

Эта структура соответствует обычной структуре систем, основанных на знаниях.

- Интерпретатор AL0 представляет собой машину логического вывода.
- Таблица советов и библиотека предикатов входят в состав базы знаний.

### 22.6.1. Миниатюрный интерпретатор AL0

Миниатюрный, независимый от конкретной игры интерпретатор AL0, реализованный на языке Prolog, показан в листинге 22.4. Эта программа осуществляет также взаимодействие с пользователем в процессе игры. Основное назначение этой программы состоит в использовании знаний из таблицы советов AL0; это означает, что программа интерпретирует таблицу советов AL0, формирует форсирующие деревья, а затем применяет их в игре. Основной алгоритм формирования форсирующего дерева аналогичен алгоритму поиска в глубину в графах AND/OR (см. главу 13); форсирующее дерево соответствует дереву решения AND/OR. С другой стороны, этот процесс напоминает также формирование дерева доказательства в экспертной системе (см. главу 15).

## Листинг 22.4. Реализация миниатюрного интерпретатора языка Advice Language 0

```
% Реализация миниатюрного интерпретатора языка Advice Language 0
%
% Эта программа проводит игру из заданной начальной позиции, используя знания,
% представленные на языке советов Advice Language 0
:- op(200, xfy, [:::]).
:- op(220,xfy, ..).
:- op(185, fx, if).
:- op(190, xfx, then).
:- op(180, xfy, or).
:- op(160, xfy, and).
:- op(140, fx, not).
playgame(Pos) :- % Провести игру начиная с позиции Pos
 playgame(Pos, nil). % Начать с пустого форсирующего дерева
playgame(Pos, ForcingTree) :-
 show(Pos),
 (end_of_game(Pos), % Игра окончена?
 write('End of game'), nl, !
);
 playmove(Pos, ForcingTree, Pos1, ForcingTree1), !,
 playgame(Pos1, ForcingTree1)
).
% Выполнить ход за своего игрока в соответствии с форсирующим деревом
playmove(Pos, Move..FTree1, Pos1, FTree1) :-
 side(Pos, w), % Своим является игрок, который
 % играет белыми фигурами
 legalmove(Pos, Move, Pos1),
 showmove(Move).
% Прочитать ход, сделанный чужим игроком
playmove(Pos, FTree, Pos1, FTree1) :-
 side(Pos, b),
 write('Your move: '),
 read(Move),
 (legalmove(Pos, Move, Pos1),
 subtree(FTree, Move, FTree1), ! % Перейти вниз по форсирующему дереву
);
 write('Illegal move'), nl,
 playmove(Pos, FTree, Pos1, FTree1)
).
% Если текущее форсирующее дерево является пустым, сформировать новое
playmove(Pos, nil, Pos1, FTree1) :-
 side(Pos, w),
 resetdepth(Pos, Pos0), % Позиция Pos0 - это позиция Pos
 % с глубиной 0
 strategy(Pos0, FTree), !, % Сформировать новое форсирующее дерево
 playmove(Pos0, FTree, Pos1, FTree1).

% Выбрать форсирующее поддерево, соответствующее ходу Move
subtree(FTrees, Move, FTree) :-
 member(Move ..FTree, FTrees),!.
subtree(_, nil).
strategy(Pos, ForcingTree) :- % Найти форсирующее дерево для позиции Pos
 Rule :: if Condition then AdviceList, % Получить консультацию
 % по таблице советов
 holds(Condition, Pos, _), !, % Согласовать Pos с предварительным условием
 member(AdviceName, AdviceList), % Поочередно проверить применимость
 % элементарных советов
 nl, write('Trying'), write(AdviceName),
 satisfiable(AdviceName, Pos, ForcingTree), !. % Выполнить совет AdviceName
 % в позиции Pos
satisfiable(AdviceName, Pos, FTree) :-
 advice(AdviceName, Advice), % Извлечь из таблицы один элементарный совет
 sat(Advice, Pos, Pos, FTree). % В предикате sat необходимо задавать две
 % позиции, которые передаются в предикаты сравнения
```

```

sat(Advice, Pos, RootPos, FTree) :-
 holdinggoal(Advice, HG),
 holds(HG, Pos, RootPos), % Требование достижения консервативной цели
 % соблюдается
 sat1(Advice, Pos, RootPos, FTree).
Sat1(Advice, Pos, RootPos, nil) :-
 bettergoal(Advice, BG),
 holds(BG, Pos, RootPos), !. % Требование достижения лучшей цели соблюдается
sat1(Advice, Pos, RootPos, Move..FTrees) :-
 side(Pos, w), !, % Своим является игрок, который играет белыми
 usmoveconstr(Advice, UMC),
 move(UMC, Pos, Move, Pos1), % Ход удовлетворяет ограничениям хода
 sat(Advice, Pos1, RootPos, FTrees).
Sat1(Advice, Pos, RootPos, FTrees) :-
 side(Pos, b), !, % Чужим является игрок, который играет черными
 themmoveconstr(Advice, TMC),
 bagof(Move..Pos1, move(TMC, Pos, Move, Pos1), MPlist),
 satall(Advice, MPlist, RootPos, FTrees). % Совет выполним во
 % всех позициях-преемниках
satall(_, [], _, []).
satall(Advice, [Move..Pos | MPlist], RootPos, [Move..FT | MFTs]) :-
 sat(Advice, Pos, RootPos, FT),
 satall(Advice, MPlist, RootPos, MFTs).

% Толкование понятий консервативной и лучшей целей: любая цель - это комбинация
% имен предикатов, сформированная с помощью связей AND/OR/NOT

holds(Goal1 and Goal2, Pos, RootPos) :- !,
 holds(Goal1, Pos, RootPos),
 holds(Goal2, Pos, RootPos).

holds(Goal1 or Goal2, Pos, RootPos) :- !,
 (holds(Goal1, Pos, RootPos)
 ;
 holds(Goal2, Pos, RootPos)
).
holds(not Goal, Pos, RootPos) :- !,
 not holds(Goal, Pos, RootPos).
holds(Pred, Pos, RootPos) :-
 (Cond =..[Pred, Pos] % Большинство предикатов не зависит от RootPos
 ;
 Cond =..[Pred, Pos, RootPos]),
 call(Cond).

% Интерпретация ограничений хода

move(MC1 and MC2, Pos, Move, Pos1) :- !,
 move(MC1, Pos, Move, Pos1),
 move(MC2, Pos, Move, Pos1).
move(MC1 then MC2, Pos, Move, Pos1) :- !,
 (move(MC1, Pos, Move, Pos1)
 ;
 move(MC2, Pos, Move, Pos1)
).
% Селекторы для компонентов элементарного совета

bettergoal(BG : _, BG).
holdinggoal(BG : HG : _, HG).
usmoveconstr(BG : HG : UMC : _, UMC).
themmoveconstr(BG : HG : UMC : TMC, TMC).
member(X, [X | L]).
member(X, [_ | L]) :-
 member(X, L).

```



Для простоты в программе, приведенной в листинге 22.4, предполагается, что свой игрок играет белыми фигурами, а чужой игрок — черными. Программа вызывается на выполнение с помощью процедуры

```
playgame(Pos)
```

где Pos — выбранная начальная позиция в игре, которая должна быть проведена компьютером. Если в позиции Pos очередь хода принадлежит чужому игроку, то программа считывает ход, введенный пользователем. В противном случае программа консультируется по таблице советов, прилагаемой к этой программе, формирует форсирующее дерево и делает свой ход в соответствии с этим деревом. Такие действия продолжаются до тех пор, пока не будет достигнут конец игры, как определено предикатом end\_of\_game (например, будет поставлен мат).

Форсирующее дерево — это дерево ходов, представленное в программе с помощью структуры

```
Move...[Reply1..Ftree1, Reply2..Ftree2, ...]
```

где “.” — инфиксный оператор, Move — первый ход своего игрока, Reply1, Reply2 и т.д. — возможные ответы чужого игрока, а Ftree1, Ftree2 и т.д. — форсирующие поддеревья, которые относятся к каждому из ответов чужого игрока.

## 22.6.2. Программа ведения эндшпиля “король и ладья против короля” на основе таблицы советов

Общая стратегия выигрыша в эндшпили с королем и ладьей против одинокого короля чужого игрока состоит в том, чтобы вынудить короля перейти на край доски или, в случае необходимости, в угол, а затем поставить мат в несколько ходов. Уточнение этого основного принципа описано ниже.

Следя за тем, чтобы ни при каких условиях не возникала патовая ситуация или ладья не оставалась незащищенной от нападения короля чужого игрока, повторно выполнять перечисленные ниже действия до тех пор, пока не будет поставлен мат.

1. Искать способ поставить королю чужого игрока мат в два хода.
2. Если указанное выше невозможно, то искать способ больше ограничить область на шахматной доске, в которой короля чужого игрока удерживает ладья своего игрока.
3. Если указанное выше невозможно, то искать способ продвинуть короля своего игрока ближе к королю чужого игрока.
4. Если ни один из приведенных выше элементарных советов 1, 2 или 3 не выполним, то искать способ сохранения текущих достижений в рамках элементарных советов 2 и 3 (т.е. сделать выжидательный ход).
5. Если не выполним ни один из элементарных советов 1, 2, 3 или 4, то искать способ получения позиции, в которой ладья своего игрока разделяет двух королей либо по вертикали, либо по горизонтали.

Подробная реализация этих принципов в виде таблицы советов AL0 приведена в листинге 22.5. Эта таблица может быть вызвана на выполнение интерпретатором AL0 (см. листинг 22.4). На рис. 22.4 иллюстрируется смысл некоторых из предикатов, используемых в этой таблице, и показан способ применения самой таблицы.

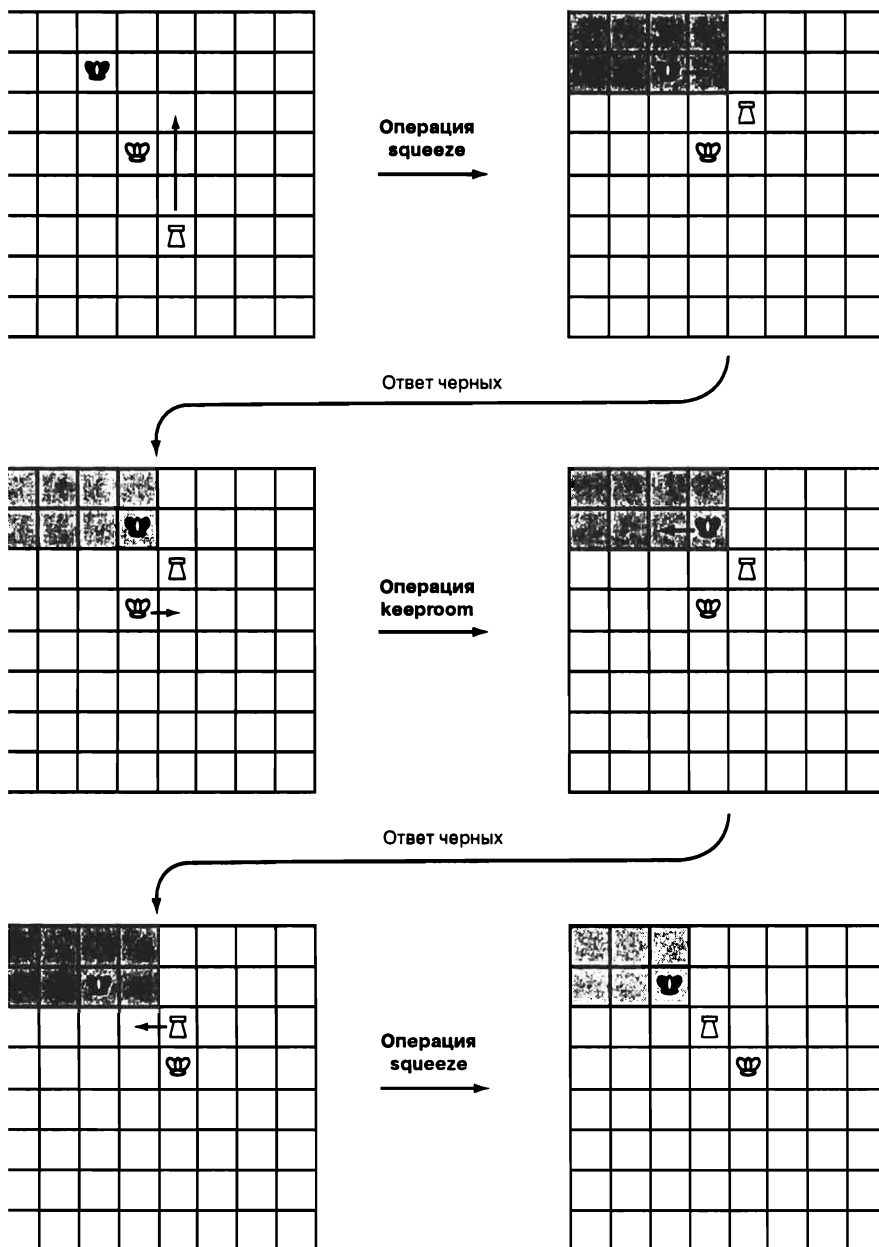


рис. 22.4. Фрагмент игры, проводимой с помощью таблицы советов, приведенной в истинке 22.5, в котором иллюстрируется метод, позволяющий зажать короля противника в угол. В этой последовательности ходов используются элементарные советы *keeproom* (выжидательный ход, после выполнения которого свободное пространство для короля противника остается неизменным) и *squeeze* (свободное пространство сужается). Область, в которой король противника удерживается ладьей своего игрока, на этом рисунке затенена. После последнего хода *squeeze* свободное пространство сужается с восьми до шести клеток

## Листинг 22.5. Таблица советов ALO для эндшпиля "король и ладья против короля". Таблица состоит из двух правил и шести элементарных советов

```
% Правила ведения эндшпиля "король и ладья против короля"
% на языке Advice Language 0
edge_rule :: if their_king_edge and kings_close
 then [mate_in_2, squeeze, approach, keeproom,
 divide_in_2, divide_in_3].
else_rule :: if true
 then [squeeze, approach, keeproom, divide_in_2, divide_in_3].
% Элементарные советы
advice(mate_in_2,
 mate :
 not rooklost and their_king_edge :
 (depth = 0) and legal then (depth = 2) and checkmove :
 (depth = 1) and legal).
advice(squeeze,
 newroomsmaller and not rookexposed and
 rookdivides and not stalemate :
 not rooklost:
 (depth = 0) and rookmove :
 nomove).
advice(approach,
 okapproachedcsquare and not rookexposed and not stalemate and
 (rookdivides or lpatt) and (roomgt2 or not our_king_edge) :
 not rooklost:
 (depth = 0) and kingdiagfirst:
 nomove).
advice(keeproom,
 themtomove and not rookexposed and rookdivides and okorndle and
 (roomgt2 or not okedge) :
 not rooklost:
 (depth = 0) and kingdiagfirst:
 nomove).
advice(dividejn_2,
 themtomove and rookdivides and not rookexposed :
 not rooklost:
 (depth < 3) and legal:
 (depth < 2) and legal).
advice(divide_in_3,
 themtomove and rookdivides and not rookexposed :
 not rooklost:
 (depth < 5) and legal :
 (depth < 4) and legal).
```

Предикаты, используемые в таблице советов, перечислены в табл. 22.2.

**Таблица 22.2. Предикаты, используемые в таблице советов**

| Предикат       | Описание                                                                                                         |
|----------------|------------------------------------------------------------------------------------------------------------------|
| Предикаты цели |                                                                                                                  |
| mate           | Королю чужого игрока поставлен мат                                                                               |
| stalemate      | Король чужого игрока поставлен в патовое положение                                                               |
| rooklost       | Король чужого игрока может взять ладью своего игрока                                                             |
| rookexposed    | Король чужого игрока может напасть на ладью своего игрока прежде, чем король своего игрока сможет защитить ладью |
| newroomsmaller | Область, в которой король чужого игрока удерживается ладьей своего игрока, уменьшилась                           |
| rookdivides    | Ладья разделяет обоих королей по вертикали или по горизонтали                                                    |

| Предикат                   | Описание                                                                                                                    |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| okapproachedcsquare        | Король своего игрока приблизился к "критической клетке" (рис. 22.5); это означает, что манхэттенское расстояние уменьшилось |
| lpatt                      | L-образный шаблон (см. рис. 22.5)                                                                                           |
| roomgt2                    | Область свободного пространства (room) для короля чужого игрока состоит больше чем из двух клеток                           |
| Предикаты ограничений хода |                                                                                                                             |
| depth = N                  | Ход, встречающийся на глубине depth = N в дереве поиска                                                                     |
| legal                      | Любой допустимый ход                                                                                                        |
| checkmove                  | Проверка хода                                                                                                               |
| rookmove                   | Ход ладьи                                                                                                                   |
| nomove                     | Цель, которая не достигается при любом ходе                                                                                 |
| kingdiagfirst              | Один из ходов короля, среди которых предпочтительными являются ходы короля по диагонали                                     |

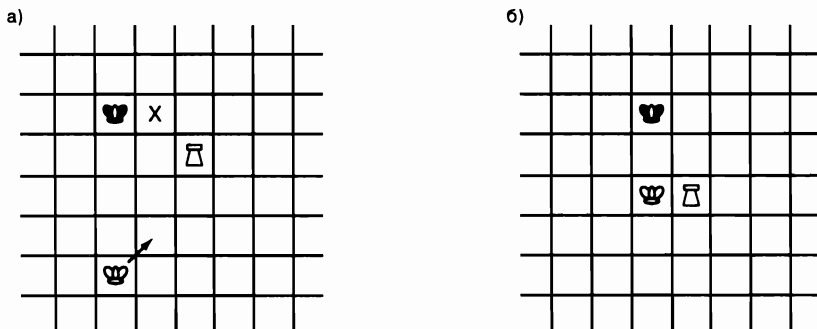


Рис. 22.5. Определение понятия L-образного шаблона: а) иллюстрация к понятию критической клетки (клетки, наиболее важной в маневрах, имеющих целью ограничить свободное пространство для короля противника, которая обозначена крестиком); белый король приближается к этой критической клетке, сделав указанный ход; б) три фигуры образуют L-образный шаблон

Параметрами этих предикатов являются либо позиции (предикаты цели), либо ходы (предикаты ограничений ходов). Предикаты цели могут иметь один или два параметра. Одним из параметров всегда является текущий узел поиска, а второй параметр (если он существует) представляет собой корневой узел дерева поиска. Второй параметр требуется в так называемых *предикатах сравнения*, которые сравнивают по некоторому признаку корневую и текущую позицию поиска. В качестве примера можно указать предикат *newroomsmaller*, который проверяет, удалось ли сузить свободное пространство для короля противника (см. рис. 22.4). Эти предикаты, а также шахматные правила для эндшпиля "король и ладья против короля" и процедура отображения шахматной доски (*show( Pos)*) приведены в программе в листинге 22.6.

#### Листинг 22.6. Библиотека предикатов для эндшпиля "король и ладья против короля"

```
% Библиотека предикатов для эндшпиля "король и ладья против короля"

% Позиция представлена с помощью функтора Side..Wx : Wy..Rx : Ry..Bx : By..Depth
```

```

% Side - соперник, имеющий право хода ('w' - белый, или 'b' - черный)
% Wx, Wy - координаты X и Y белого короля
% Rx, Ry - координаты X и Y белой ладьи
% Bx, By - координаты X и Y черного короля
% Depth - глубина позиции в дереве поиска

% Селекторные отношения

side(Side_., Side). % Side - соперник, имеющий право хода в данной позиции
wk(_..WK_., WK). % Координаты белого короля
wr(_.._..WR_., WR). % Координаты белой ладьи
bk(_.._..BK_., BK). % Координаты черного короля
depth(_.._.._..Depth, Depth). % Глубина позиции в дереве поиска
resetdepth(S..W..R..B..D, S..W..R..B..0). % Копия позиции с глубиной 0

% Некоторые отношения между клетками

n(N, N1) :- % Числовые обозначения соседних клеток должны находиться
 % в пределах размеров доски
 (N1 is N + 1;
 N1 is N - 1),
 in(N1).
in(N) :-
 N > 0, N < 9.
diagngb(X : Y, X1 : Y1) :- % Клетки, соседние по диагонали
 n(X, X1), n(Y, Y1).
verngb(X : Y, X : Y1) :- % Клетки, соседние по вертикали
 n(Y, Y1).
horngb(X : Y, X1 : Y) :- % Клетки, соседние по горизонтали
 n(X, X1).
ngb(S, S1) :- % Соседние клетки, первая диагональ
 diagngb(S, S1);
 horngb(S, S1);
 verngb(S, S1).
end_of_game(Pos) :-
 mate(Pos).

% Предикаты ограничений хода
% move(MoveConstr, Pos, Move, NewPos):
% специализированные предикаты для формирования ходов

move(depth < Max, Pos, Move, Pos1) :-
 depth(Pos, D),
 D < Max, !.

move(depth = D, Pos, Move, Pos1) :-
 depth(Pos, D), !.

move(kingdiagfirst, w..W..R..B..D, W-W1, b..W1..R..B..D1) :-
 D1 is D + 1,
 ngb(W, W1), % Предикат ngb в первую очередь вырабатывает ходы по диагонали
 not ngb(W1, B), % Король не должен становиться под шах
 W1 \== R. % Его позиция не должна совпадать с позицией ладьи

move(rookmove, w..W..Rx : Ry..B..D, Rx : Ry-R, b..W..R..B..D1) :-
 D1 is D + 1,
 coord(I), % Целое число от 1 до 8
 (R = Rx : I; R = I : Ry), % Ход по вертикали или по горизонтали
 R \== Rx : Ry, % Ход уже должен быть сделан
 not inway(Rx : Ry, W, R). % На пути нет белого короля

move(checkmove, Pos, R-Rx : Ry, Pos1) :-
 wr(Pos, R),
 bk(Pos, Bx : By),
 (Rx = Bx; Ry = By), % Ладья и черный король стоят на одной линии
 move(rookmove, Pos, R-Rx : Ry, Pos1).

```

```

move(legal, w..P, M, PI) :-
 (MC = kingdiagfirst; MC = rookmove),
 move(MC, w..P, M, PI).

move(legal, b..W..R..B..D, B-B1, w..W..R..B1..D1) :-
 D1 is D + 1,
 ngb(B, B1),
 not check(w..W..R..B1..D1).

legalmove(Pos, Move, Pos1) :-
 move(legal, Pos, Move, Pos1).

check(_..W..Rx:Ry..Bx:By.._) :-
 ngb(W, Bx : By); % Король находится слишком близко
 (Rx = Bx; Ry = By),
 Rx : Ry \== Bx : By, % Ладья не взята
 not inway(Rx : Ry, W, Bx : By).

inway(S, S1, S1) :- !.

inway(X1 : Y, X2 : Y, X3 : Y) :-
 ordered(X1, X2, X3), !.

inway(X : Y1, X : Y2, X : Y3) :-
 ordered(Y1, Y2, Y3).

ordered(N1, N2, N3) :-
 N1 < N2, N2 < N3;
 N3 < N2, N2 < N1.

coord(1). coord(2). coord(3). coord(4).
coord(5). coord(6). coord(7). coord(8).

% Целевые предикаты
true(Pos).
thentomove(b.._). % Очередь хода принадлежит чужому игроку, который играет
 % черными фигурами
mate(Pos) :-
 side(Pos, b),
 check(Pos),
 not legalmove(Pos, _, _).
stalemate(Pos) :-
 side(Pos, b),
 not check(Pos),
 not legalmove(Pos, _, _).
newroomsmaller(Pos, RootPos) :-
 room(Pos, Room),
 room(RootPos, RootRoom),
 Room < RootRoom.
rookexposed(Side..W..R..B.._) :-
 dist(W, R, D1),
 dist(B, R, D2),
 (Side = w, !, D1 > D2 + 1
 ;
 Side = b, !, D1 > D2).

okapproachedcsquare(Pos, RootPos) :-
 okcsquaremdist(Pos, D1),
 okcsquaremdist(RootPos, D2),
 D1 < D2.
okcsquaremdist(Pos, Mdist) :- % Манхэттенское расстояние между
 % белым королем WK и критической клеткой
 wk(Pos, WK),
 cs(Pos, CS), % Критическая клетка
 manhdist(WK, CS, Mdist).
rookdivides(_..Wx : Wy..Rx : Ry..Bx : By.._) :-

```

```

ordered(Wx, Rx, Bx), !;
ordered(Wy, Ry, By).
lpatt(...W..R..B..) :-
 manhdist(W, B, 2),
 manhdist(R, B, 3).
% L-образный шаблон
okorndle(...W..R.., ...W1..R1..) :-
 dist(W, R, D),
 dist(W1, R1, D1),
 D =< D1.
roomgt2(Pos) :-
 room(Pos, Room),
 Room > 2.

our_king_edge(...X : Y..) :-
 (X = 1, !; X = 8, !; Y = 1, !; Y = 8).
 % Белый король в углу
their_king_edge(...W..R..X : Y..) :-
 (X = 1, !; X = 8, !; Y = 1, !; Y = 8).
 % Черный король в углу
kings_close(Pos) :-
 wk(Pos, WK), bk(Pos, BK),
 dist(WK, BK, D),
 D < 4.
 % Расстояние между королями меньше 4
rooklost(...W..B..B..).
rooklost(b...W..R..B..) :-
 ngb(B, R),
 not ngb(W, R).
 % Ладья взята
 % Черный король нападает на ладью
 % Белый король ее не защищает
 % Расстояние в ходах королей
dist(X : Y, X1 : Y1, D) :-
 absdiff(X, X1, Dx),
 absdiff(Y, Y1, Dy),
 max(Dx, Dy, D).
absdiff(A, B, D) :-
 A > B, !, D is A - B;
 D is B - A.
max(A, B, M) :-
 A > B, !, M = A;
 M = B.
manhdist(X : Y, X1 : Y1, D) :-
 absdiff(X, X1, Dx),
 absdiff(Y, Y1, Dy),
 D is Dx + Dy.
 % Манхэттенское расстояние
room(Pos, Room) :-
 wr(Pos, Rx : Ry),
 bk(Pos, Bx : By),
 (Bx < Rx, SideX is Rx - 1; Bx > Rx, SideX is 8 - Rx),
 (By < Ry, SideY is Ry - 1; By > Ry, SideY is 8 - Ry),
 Room is SideX * SideY, !
 ;
 Room is 64.
 % Ладья находится на одной линии с черным королем
cs(...W..Rx : Ry..Bx : By.., Cx : Cy) :-
 (Bx < Rx, !, Cx is Rx - 1; Cx is Rx + 1),
 (By < Ry, !, Cy is Ry - 1; Cy is Ry + 1).
 % "Критическая клетка"

% Процедуры вывода данных

show(Pos) :-
 nl,
 coord(Y), nl,
 coord(X),
 writepiece(X : Y, Pos),
 fail.

show(Pos) :-
 side(Pos, S), depth(Pos, D),
 nl, write('Side= '), write(S),
 write('Depth= '), write(D), nl.
writepiece(Square, Pos) :-
 wk(Pos, Square), !, write('W');

```

```

wr(Pos, Square), !, write('R');
bk(Pos, Square), !, write('B');
write('.').
showmove(Move) :-
 nl, write(Move), nl.

```

---

Пример того, как играет эта программа, основанная на использовании таблицы советов, показан на рис. 22.4. Игра продолжается с последней позиции (см. рис. 22.4), как в следующем варианте (при условии, что чужой игрок делает те ходы, которые указаны в этом варианте). Здесь используется буквенно-цифровая шахматная система обозначений, согласно которой вертикали шахматной доски обозначены буквами “a”, “b”, “c” и т.д., а горизонтали — цифрами 1, 2, 3 и т.д. Первая прописная буква обозначает цвет (B — черный, W — белый), а вторая прописная буква — фигуру (K — король, R — ладья). Например, ход “BK b7” означает: передвинуть черного короля на клетку, которая находится на пересечении вертикали “b” и горизонтали 7.

```

...BK b7
WK d5 BK c7
WK c5 BK b7
WR c6 BK a7
WR b6 BK a8
WK b5 BK a7
WK c6 BK a8
WK c7 BK a7
WR c6 BK a8
WR a6 мат

```

Теперь необходимо найти ответы на некоторые вопросы. Прежде всего, является ли эта программа, основанная на использовании таблицы советов, правильной в том смысле, что она ставит мат, несмотря ни на какие оборонительные действия, если игра начинается с любой позиции эндшпиля “король и ладья против короля”? В [13] посредством формального доказательства показано, что в этом смысле таблица советов, по сути, аналогичная приведенной в листинге 22.5, является правильной.

Еще один вопрос состоит в том, действительно ли эта программа, основанная на таблице советов, является оптимальной в том отношении, что она всегда ставит мат за наименьшее количество ходов? Можно легко показать на примерах, что с этой точки зрения игра программы не является оптимальной. Как известно, в этом эндшпили оптимальные варианты (в которых оба игрока выбирают самые сильные ходы) имеют длину не больше 16 ходов. Хотя игра по рассматриваемой здесь таблице советов может оказаться довольно далекой от этого оптимума, было показано, что количество необходимых ходов при использовании этой таблицы советов все еще достаточно далеко от опасного предела, равного 50. Это важно потому, что в шахматах действует правило 50 ходов: в таких эндшпилях, как “король и ладья против короля”, сильнейшая сторона должна поставить мат за 50 ходов; в противном случае может быть объявлена ничья.

## Проект

Рассмотрите некоторые другие простые шахматные эндшпили, такие как “король и пешка против короля”, и напишите программу AL0 (вместе с соответствующими определениями предикатов) для поиска игровых продолжений в этих эндшпилях.

## Резюме

- Игры с двумя участниками соответствуют формальному определению *графов AND/OR*. Поэтому для поиска в деревьях игры могут использоваться процедуры поиска AND/OR.



- Программу непосредственного осуществления *поиска в глубину* в деревьях игры можно составить довольно легко, но она становится слишком неэффективной при ведении достаточно интересных игр. В подобных случаях более легко применимый подход состоит в использовании *принципа минимакса* в сочетании с функцией оценки и поиском с ограничением по глубине.
- Одной из эффективных реализаций принципа минимакса является *альфа-бета-алгоритм*. Эффективность альфа-бета-алгоритма зависит от порядка, в котором происходит поиск альтернатив. В наилучшем случае альфа-бета-алгоритм в действительности сокращает коэффициент ветвления дерева игры до квадратного корня этого коэффициента.
- К некоторым *усовершенствованиям* основного альфа-бета-алгоритма относятся следующие: расширение поиска до тех пор, пока не будут найдены спокойная позиция, последовательное углубление и эвристическое отсечение.
- Числовая оценка представляет собой весьма ограничительную форму применения знаний о конкретной игре. В любом подходе к ведению игры, основанном на использовании большего объема знаний, должно быть предусмотрено *применение знаний* в виде готовых образцов действий. Такой подход может быть реализован с помощью так называемых языков советов Advice Language, в которых знания представлены в виде целей и средств достижения этих целей.
- В настоящей главе рассматриваются следующие программы: реализация принципа минимакса и альфа-бета-алгоритма, интерпретатор языка Advice Language 0 и таблица советов для ведения шахматного эндшпиля “король и ладья против короля”.
- В данной главе рассматривались следующие понятия:
  - игры с двумя участниками, с полной информацией;
  - деревья игры;
  - функция оценки, принцип минимакса;
  - статические значения, зафиксированные значения;
  - альфа-бета-алгоритм;
  - последовательное углубление, эвристическое отсечение, эвристические функции поиска спокойных позиций;
  - эффект горизонта;
  - языки советов Advice Language;
  - цели, ограничения, элементарный совет, таблица советов.

## Дополнительные источники информации

Принцип минимакса, реализованный в виде альфа-бета-алгоритма, представляет собой один из подходов, наиболее часто используемых для создания игровых программ, особенно шахматных. Принцип минимакса был предложен в [144]. История разработки метода, основанного на использовании альфа-бета-алгоритма, является довольно запутанной, так как несколько исследователей открыли или реализовали этот метод или, по меньшей мере, его часть независимо друг от друга. Эта интересная история описана в [74], где также представлена более компактная формулировка альфа-бета-алгоритма с использованием принципа “neg-max” (в отличие от принципа минимакса, в котором применяются максимальные значения в множестве оценок дочерних узлов на нечетных уровнях и минимальные значения на четных уровнях, принцип “neg-max” предусматривает использование на каждом уровне взятых с обратным знаком (neg) максимальных значений оценок дочерних узлов (max)) вместо минимакса и приведены результаты математического анализа его производительности.

сти. Результаты всестороннего исследования нескольких алгоритмов на основе минимакса и их анализа приведены в [118]. В [69] приведен также обзор алгоритмов поиска. В [124] дано описание наиболее современных вариантов алгоритма альфа-бета-поиска. В этой работе рассматривается еще один интересный вопрос, касающийся принципа минимакса: "Если известно, что статическая оценка достоверна лишь в определенной степени, не следует ли из этого, что зафиксированные минимаксные значения являются более достоверными, чем сами статические значения?" В [118] приведен также сборник результатов математического анализа задач, которые относятся к данной проблеме.

Результаты исследования процесса распространения ошибок в деревьях минимакса показывают, когда и почему становится выгодно использовать метод минимаксного опережающего просмотра.

В [11], [54] и [95] приведены сборники статей по ведению на компьютерах сложных игр, особенно шахмат. Современные результаты исследований по компьютерным шахматам публикуются в серии *Advances in Computer Chess* [1] и в журнале *ICCA*.

Подход к использованию в шахматах знаний, представленных в виде образцов на языке советов *Advice Language*, был предложен Мичи (Michie) и получил дальнейшее развитие в [19], а также в [14], [16], [17]. Приведенная в данной главе программа ведения эндшпиля "король и ладья против короля", основанная на использовании таблицы советов, содержит немного измененную таблицу советов, правильность которой доказана математически в [13].

К другим интересным экспериментам в области шахмат, основанным на использовании знаний (в отличие от подходов, опирающихся на применении средств поиска), относятся [7], [123] и [168]. Создается впечатление, что в последнее время интерес к разработке шахматных программ, основанных на использовании знаний, угасает, вероятно, из-за их поражения в конкурентной борьбе с шахматными программами, которые основаны главным образом на использовании вычислительных мощностей и отыскивают решение путем перебора. В результате резкого улучшения характеристик компьютерных аппаратных средств, включая шахматные аппаратные средства специального назначения, появилась возможность просматривать до нескольких сотен миллионов позиций в секунду, поэтому отсутствие более тонких знаний компенсируется за счет грубой силы. Кульминацией подхода, основанного на использовании грубой силы, явился проигрыш одного из ведущих шахматистов мира Гарри Каспарова в матче с программой *Deep Blue* ([65]). Но этот успех в конкурентной борьбе не позволяет устранить известный недостаток программ, основанных на использовании примитивного перебора: они не способны описать свою игру в концептуальных терминах. Поэтому с точки зрения объяснения результатов, комментирования и обучения остается необходимым подход, основанный на использовании знаний. С другой стороны, создается впечатление, что игра го, если судить по тем же простым показателям успеха в конкурентной борьбе, демонстрирует необходимость использования подхода, основанного на использовании знаний. При составлении программ для этой игры решения, основанные на использовании простого перебора, оказались гораздо менее удачными, чем в шахматах, из-за значительно более высокой комбинаторной сложности.

# Метапрограммирование

*В этой главе...*

|                                                                                                                         |     |
|-------------------------------------------------------------------------------------------------------------------------|-----|
| 23.1. Метапрограммы и метаинтерпретаторы                                                                                | 559 |
| 23.2. Метаинтерпретаторы Prolog                                                                                         | 560 |
| 23.3. Обобщение на основе объяснения                                                                                    | 564 |
| 23.4. Объектно-ориентированное программирование                                                                         | 570 |
| 23.5. Программирование, управляемое шаблонами                                                                           | 576 |
| 23.6. Простая программа автоматического доказательства теорем,<br>реализованная в виде программы, управляемой шаблонами | 583 |

Благодаря наличию в языке Prolog возможностей по манипулированию символами он является мощным средством реализации других языков и принципов программирования. В данной главе показано, как можно создать метаинтерпретаторы Prolog — интерпретаторы языка Prolog на самом языке Prolog. Здесь рассматривается специальный метод компиляции программ, называемый *обобщением на основе объяснения*, который был разработан как один из подходов к машинному обучению. Кроме того, в этой главе описаны простые интерпретаторы для двух других подходов к программированию: объектно-ориентированное программирование и программирование, управляемое шаблонами.

## 23.1. Метапрограммы и метаинтерпретаторы

*Метапрограммой* называется программа, которая принимает в качестве данных другие программы. Примерами метапрограмм являются интерпретаторы и компиляторы. Особой разновидностью метапрограмм являются метаинтерпретаторы — интерпретаторы для некоторого языка, написанные на том же языке. Таким образом, метаинтерпретатором Prolog является интерпретатор языка Prolog, который сам написан на языке Prolog.

Благодаря наличию в языке Prolog возможностей по манипулированию символами он является мощным языком, который может успешно применяться для метапрограммирования. Поэтому он часто служит в качестве языка реализации для других языков. Многие специалисты отмечают, что Prolog особенно хорошо подходит для использования в качестве языка быстрой разработки прототипов в тех ситуациях, когда требуется как можно быстрее воплотить в жизнь новые идеи. Это особенно важно в тех случаях, когда возникает необходимость разработать новый язык, осуществить на практике новый принцип программирования или опробовать вновь созданную архитектуру программы. Этот язык позволяет быстро реализовывать новые идеи и проводить эксперименты. При разработке прототипов основное внимание уделяется тому, чтобы новые идеи оформлялись в виде экспериментального образца максимально быстро и с наименьшими затратами, благодаря чему можно было бы

немедленно приступить к их проверке. С другой стороны, особое значение повышению эффективности такой реализации не придается. После проверки первоначального замысла прототип может быть реализован повторно, для чего можно воспользоваться другим, более эффективным языком программирования. И даже если потребуются отказаться от прототипа на языке Prolog в пользу другого языка, все равно усилия, затраченные на разработку этого прототипа, не будут напрасными, поскольку он обычно позволяет ускорить творческий этап разработки.

В предыдущих главах этой книги уже встречались метапрограммы; в качестве примера можно указать интерпретаторы правил вывода (см. главы 15 и 16). Они обрабатывают язык правил вывода, который фактически является языком программирования, хотя программы, написанные на нем, обычно называют базами знаний из-за их специализированного содержания. Еще одним примером является интерпретатор для гипотез на языке ILP (см. главу 19). В этой главе приведены перечисленные ниже дополнительные примеры, которые позволяют показать, насколько легко могут быть написаны метапрограммы на языке Prolog.

- Создание метаинтерпретаторов Prolog.
- Применение метода обобщения на основе объяснения.
- Реализация на языке Prolog других принципов программирования, в частности, объектно-ориентированного программирования и программирования, управляемого шаблонами.

## 23.2. Метаинтерпретаторы Prolog

### 23.2.1. Простейший метаинтерпретатор Prolog

Метаинтерпретатор Prolog принимает на входе программу Prolog и цель Prolog, после чего выполняет данную цель применительно к данной программе; это означает, что метаинтерпретатор пытается доказать, что цель логически следует из этой программы. Но для того чтобы он имел определенное практическое значение, метаинтерпретатор не должен действовать полностью аналогично оригинальному интерпретатору Prolog; от него требуются некоторые дополнительные функциональные возможности, такие как формирование дерева доказательства или трассировка выполнения программ.

В этой главе для упрощения предполагается, что данная программа уже использовалась для консультации системой Prolog, которая вызывает на выполнение метаинтерпретатор. Поэтому метаинтерпретатор может быть определен как процедура `prove` с одним параметром (таковым является цель, которая должна быть достигнута) следующим образом:

```
prove(Goal)
```

Как показано ниже, простейший метаинтерпретатор Prolog является тривиальным.

```
prove (Goal) :-
 call(Goal).
```

В данном случае вся работа метаинтерпретатора была передана (с помощью предиката `call`) оригинальному интерпретатору Prolog, поэтому такой метаинтерпретатор ведет себя точно так же, как и сам интерпретатор Prolog. Безусловно, что такой вариант метаинтерпретатора не имеет никакого практического значения, поскольку он не предоставляет какие-либо дополнительные возможности. Для того чтобы обеспечить реализацию таких важных средств метаинтерпретатора, как формирование деревьев доказательства, прежде всего необходимо уменьшить “степень детализации” этого интерпретатора, чтобы он мог обрабатывать не только всю программу, но и ее компоненты. Такое уменьшение степени детализации метаинтерпретатора становится

возможным благодаря наличию следующего встроенного предиката, предусмотренного во многих реализациях Prolog:

```
clause(Head, Body)
```

Этот предикат осуществляет “выборку” любого предложения из программы, применяемой для консультации. Здесь Head — голова предложения, извлеченного из базы данных, а Body — его тело. Для единичного предложения (факта) параметр Body = true. В неединичном предложении (правиле) тело может содержать одну или несколько целей. Если оно содержит одну цель, то Body и есть эта цель. Если тело содержит несколько целей, то их выборка осуществляется в виде следующей пары:

```
Body = (FirstGoal, OtherGoals)
```

В этом терме запятая представляет собой встроенный инфиксный оператор. В стандартной системе обозначений Prolog эту пару можно заменить следующим эквивалентным термом:

```
, (FirstGoal, OtherGoals)
```

где OtherGoals может в свою очередь представлять собой пару, состоящую еще из одной цели и оставшихся целей. В вызове clause( Head, Body) первый параметр Head не должен быть переменной. Предположим, что программа, применяемая для консультации, содержит обычную процедуру member. В таком случае выборка предложений процедуры member может быть выполнена таким образом:

```
?- clause(member(X, L), Body).
X = _14
L = [_14 | _15]
Body = true;
X = _14
L = [_15 | _16]
Body = member(_14, _16)
```

В листинге 23.1 показан простой метаинтерпретатор для языка Prolog, реализованный с такой степенью детализации, которая, как показала практика, является удобной для большинства областей применения. Но следует отметить, что этот метаинтерпретатор предназначен только для так называемого “чистого” (базового) языка Prolog. Он не позволяет обрабатывать встроенные предикаты, в частности оператор отсечения. Но практическая применимость этого простого метаинтерпретатора определяется тем фактом, что он предоставляет схему, которая может быть легко модифицирована для получения других интересных эффектов. Одним из подобных широко известных расширений является создание средства трассировки для системы Prolog. Еще одна из его полезных особенностей состоит в том, что с его помощью можно предотвратить вхождение интерпретатора Prolog в бесконечные циклы путем ограничения глубины вызовов подцелей (см. упражнение 23.2).

### Листинг 23.1. Простой метаинтерпретатор Prolog

```
% Простой метаинтерпретатор Prolog

prove(true).

prove((Goal1, Goal2)) :-
 prove(Goal1),
 prove(Goal2).

prove(Goal) :-
 clause(Goal, Body),
 prove(Body).
```

**23.1.** Что произойдет при попытке вызвать на выполнение с помощью метаинтерпретатора, приведенного в листинге 23.1, сам этот метаинтерпретатор, например, следующим образом:

```
?- prove(prove(member(X, [a, b, c]))).
```

При такой попытке возникает проблема, поскольку данный метаинтерпретатор не может выполнять встроенные предикаты, такие как `clause`. Каким образом можно легко модифицировать этот метаинтерпретатор, чтобы он приобрел способность вызывать сам себя на выполнение, как в приведенном выше запросе?

**23.2.** Модифицируйте метаинтерпретатор, приведенный в листинге 23.1, ограничив глубину поиска доказательства системой Prolog. Предположим, что модифицированный метаинтерпретатор представляет собой предикат `prove( Goal, DepthLimit)`, который достигает успеха, если  $\text{DepthLimit} \geq 0$ . Каждый рекурсивный вызов уменьшает этот предел.

### 23.2.2. Трассировка с помощью метаинтерпретатора

Ниже приведен результат первой попытки расширить простой метаинтерпретатор, приведенный в листинге 23.1, чтобы создать интерпретатор, обеспечивающий трассировку.

```
prove(true) :- !.
prove((Goal1, Goal2)) :- !,
 prove(Goal1),
 prove(Goal2).
prove(Goal) :-
 write('Call: '), write(Goal), nl,
 clause(Goal, Body),
 prove(Body),
 write('Exit: '), write(Goal), nl.
```

В данном случае операторы отсечения требуются для предотвращения необходимости отображать “истинные” (`true`) и составные цели в форме `( Goal1, Goal2 )`. Эта программа трассировки имеет несколько недостатков: она не обеспечивает трассировку недостигнутых целей и не показывает результатов перебора с возвратами, когда повторно выполняется одна и та же цель. В этом отношении программа трассировки, приведенная в листинге 23.2, является более совершенной. Кроме того, для удобства чтения отображаемые цели обозначаются в ней отступами, пропорционально той глубине логического вывода, на которой они вызываются. Но и этот метаинтерпретатор все еще ограничивается чистым языком Prolog. Пример вызова рассматриваемой программы трассировки приведен ниже.

```
?- trace((member(X, [a, b]), member(X, [b, c]))).
Call: member(_0085, [a, b])
Exit: member(a, [a, b])
Call: member(a, [b, c])
Call: member(a, [c])
Call: member(a, [])
Fail: member(a, [])
Fail: member(a, [c])
Fail: member(a, [b, c])
Redo: member(a, [a, b])
Call: member(_0085, [b])
Exit: member(b, [b])
Exit: member(b, [a, b])
Call: member(b, [b, c])
Exit: member(b, [b, c])
```

## Листинг 23.2. Метаинтерпретатор Prolog для трассировки программ на чистом языке Prolog

```
% trace(Goal): выполнить цель Goal, которая определена в программе Prolog,
% и вывести информацию трассировки

trace(Goal) :-
 trace(Goal, 0).

trace(true, Depth) :- !. % Красный оператор отсечения; Depth - глубина вызова

trace((Goal1, Goal2), Depth) :- !, % Красный оператор отсечения
 trace(Goal1, Depth),
 trace(Goal2, Depth).

trace(Goal, Depth) :-
 display('Call: ', Goal, Depth),
 clause(Goal, Body),
 Depth1 is Depth + 1,
 trace(Body, Depth1),
 display('Exit: ', Goal, Depth),
 display_redo(Goal, Depth).

trace(Goal, Depth) :- % Все альтернативные варианты исчерпаны
 display('Fail: ', Goal, Depth),
 fail.

display(Message, Goal, Depth) :-
 tab(Depth), write(Message),
 write(Goal), nl.

display_redo(Goal, Depth) :-
 true % Достижение первой цели оказалось несложным
 ;
 display('Redo: ', Goal, Depth), % Затем вывести сообщение о том,
 % что начинается перебор с возвратами,
 % и вынудить систему выполнить перебор
 % с возвратами
 fail.
```

Эта программа трассировки выводит описанную ниже информацию для каждой выполняемой цели.

1. Цель, которая должна быть выполнена (Call: Goal).
2. Трассировка подцелей (с отступлениями).
3. Если цель достигается, то отображается ее окончательная конкретизация (Exit: InstantiatedGoal); если цель не достигается, то отображается результат Fail: Goal.
4. В случае перебора с возвратами к ранее достигнутому целям сообщение принимает вид Redo: InstantiatedGoal (конкретизация этой цели в предыдущем решении).

Безусловно, существует возможность продолжить доработку этого трассирующего интерпретатора в соответствии с конкретными требованиями пользователей.

### 23.2.3. Формирование деревьев доказательства

Еще одним широко известным расширением простого интерпретатора, приведенного в листинге 23.1, является формирование деревьев доказательства. Поэтому после достижения некоторой цели вырабатывается дерево ее доказательства, которое может служить для дальнейшей обработки. Как было показано в главах 15 и 16, формирование деревьев доказательства реализовано для экспертных систем на основе правил. Хотя синтаксис этих правил отличается от синтаксиса Prolog, принципы формирования дерева доказательства остаются одинаковыми. Эти принципы можно

легко реализовать в метаинтерпретаторе, приведенном в листинге 23.1. Например, предположим, что можно представить дерево доказательства в зависимости от конкретного случая, как описано ниже.

1. Для цели `true` деревом доказательства является `true`.
2. Для пары целей `(Goal1, Goal2)` деревом доказательства является пара `(Proof1, Proof2)`, состоящая из деревьев доказательства этих двух целей.
3. Для цели `Goal`, согласующейся с головой предложения, телом которого является `Body`, деревом доказательства служит `Goal <== Proof`, где `Proof` — дерево доказательства `Body`.

Эти требования можно реализовать в простом метаинтерпретаторе, приведенном в листинге 23.1, следующим образом:

```
:- op(500, xfy, <==).
prove(true, true).
prove((Goal1, Goal2), (Proof1, Proof2)) :-
 prove(Goal1, Proof1),
 prove(Goal2, Proof2).
prove(Goal, Goal <== Proof) :-
 clause(Goal, Body),
 prove(Body, Proof).
```

Такое дерево доказательства может использоваться различными способами. В главах 15 и 16 оно служило в качестве основы для выработки объяснения последовательности рассуждений в экспертной системе. В следующем разделе рассматривается еще один интересный вариант использования дерева доказательства — обобщение на основе объяснения.

## 23.3. Обобщение на основе объяснения

Идея обобщения на основе объяснения впервые была реализована в области машинного обучения, целью которого является обобщение заданных примеров в виде общих описаний рассматриваемых понятий. Одним из способов формирования таких описаний является обобщение на основе объяснения (*Explanation-Based Generalization* — *EBG*), при котором обычно используется только один пример. Малое количество примеров компенсируется за счет применения в системе *фоновых знаний*, которые, как правило, именуются *теорией проблемной области*.

Способ *EBG* основан на использовании при формировании обобщенных описаний следующей идеи: если дан некоторый экземпляр целевого понятия, то можно воспользоваться теорией проблемной области для объяснения того, как этот экземпляр фактически удовлетворяет данному понятию. Затем анализируется это объяснение и предпринимается попытка обобщить его таким образом, чтобы оно подходило не только для заданного экземпляра, но и для множества “аналогичных” экземпляров. Затем такое обобщенное объяснение становится частью описания рассматриваемого понятия и может в дальнейшем использоваться при распознавании экземпляров этого понятия. Кроме того, требуется, чтобы формируемое описание понятия было *операционным*; это означает, что оно должно быть сформулировано в терминах понятий, объявленных пользователем как операционные. Интуитивно можно себе представить, что описание понятия является операционным (определяющим некоторую операцию), если его можно (относительно) легко использовать на практике. Задача определения того, что подразумевается под словом “операционный”, полностью возлагается на пользователя.

В реализации метода *EBG* эти абстрактные идеи должны были стать более конкретными. Один из способов их реализации в логике языка *Prolog* состоит в следующем.

- Понятие реализуется в качестве предиката.
- Описание понятия представляет собой определение предиката.



- Объяснением является дерево доказательства, которое демонстрирует, как заданный экземпляр удовлетворяет целевому понятию.
- Теория проблемной области представлена в виде множества доступных предикатов, определенных как программа Prolog.

В таком случае задача обобщения на основе объяснения может быть сформулирована, как описано ниже.

## Дано

- **Теория проблемной области.** Множество предикатов, доступное для механизма обобщения на основе объяснения, включая целевой предикат, для которого должно быть сформировано операционное определение.
- **Критерии операционности.** Эти критерии определяют предикаты, которые могут использоваться в определении целевого предиката.
- **Учебный пример.** Множество фактов, описывающих конкретную ситуацию и экземпляр целевого понятия таким образом, чтобы этот экземпляр мог быть получен путем логического вывода из заданного множества фактов и теории проблемной области.

## Найти

- Обобщение учебного экземпляра и операционное определение целевого понятия; это определение состоит из достаточного условия (в терминах операционных предикатов) для данного обобщенного экземпляра, при котором удовлетворяется целевое понятие.

Метод обобщения на основе объяснения в такой постановке может рассматриваться как своего рода компиляция программы с преобразованием из одной формы в другую. Первоначальная программа определяет целевое понятие в терминах предикатов теории проблемной области. Откомпилированная программа определяет то же целевое понятие (или подпонятие) в терминах “целевого языка”, т.е. только с помощью операционных предикатов. Механизм компиляции, предусмотренный в методе EBG, является довольно необычным. Он предусматривает выполнение первоначальной программы на заданном примере, что приводит к получению дерева доказательства. Затем это дерево доказательства обобщается таким образом, что сохраняется его структура, но постоянные заменяются переменными при любой возможности. В полученном таким образом обобщенном дереве доказательства некоторые узлы указывают на операционные предикаты. Затем дерево сокращается так, что остаются только эти “операционные узлы” и корень. Полученный результат представляет собой операционное определение целевого понятия.

Это описание проще всего проиллюстрировать на примере практического применения метода EBG. В листинге 23.3 приведены два определения теорий проблемной области для EBG. Первая теория проблемной области касается вручения подарка, а вторая описывает движения лифта. Рассмотрим первую проблемную область. Допустим, что учебный экземпляр состоит в следующем:

```
% Джон вручает сам себе шоколад (разрешает себе его съесть)
gives(john, john, chocolate)
```

### Листинг 23.3. Два определения задач для обобщения на основе объяснения

```
% Для совместимости с некоторыми реализациями Prolog следующие предикаты
% определены как динамические:
```

```
:- dynamic gives/3, would_please/2, would_comfort/2, feels_sorry_for/2,
go/3, move/2, move_list/2.
```

```
% Теория проблемной области, которая относится к теме вручения подарков
```

```

gives(Person1, Person2, Gift) :-
 likes(Person1, Person2),
 would_please(Gift, Person2).

gives(Person1, Person2, Gift) :-
 feels_sorry_for(Person1, Person2),
 would_comfort(Gift, Person2).

would_please(Gift, Person) :-
 needs(Person, Gift).

would_comfort(Gift, Person) :-
 likes(Person, Gift).

feels_sorry_for(Person1, Person2) :-
 likes(Person1, Person2),
 sad(Person2).

feels_sorry_for(Person, Person) :-
 sad(Person).

% Операционные предикаты
operational(likes(_, _)).
operational(needs(_, _)).
operational(sad(_)).

% Пример проблемной ситуации
likes(john, annie).
likes(annie, john).
likes(john, chocolate).
needs(annie, tennis_racket).
sad(john).

% Еще одна теория проблемной области, применяемая для описания движений лифта
% go(Level, GoalLevel, Moves) если
% список движений Moves переводит лифт с этажа Level на этаж GoalLevel
go(Level, GoalLevel, Moves) :-
 move_list(Moves, Distance), % Список движений и пройденное расстояние
 Distance == GoalLevel - Level.

move_list([], 0).

move_list([Move1 | Moves], Distance + Distance1) :-
 move_list(Moves, Distance),
 move(Move1, Distance1).

move(up, 1).
move(down, -1).

operational(A == B).

```

**В результате выработки доказательства рассматриваемый метаинтерпретатор находит следующее доказательство обоснованности выполненного действия:**

```

gives(john, john, chocolate) <==
(feels_sorry_for(john, john) <== sad(john), % Джону стало жалко самого себя,
 % поскольку ему грустно,
 would_comfort(chocolate, john) <== likes(john, chocolate) % и Джону будет
 % приятно, поскольку он любит шоколад
)

```

Это доказательство может быть обобщено путем замены постоянных `john` и `chocolate` переменными следующим образом:

```
gives(Person, Person, Thing) <== % Человек вручает себе какую-то вещь, если
(feels_sorry_for(Person, Person) <== sad(Person), % человеку жалко себя,
 % потому что ему грустно,
 would_comfort(Thing, Person) <== likes(Person, Thing) % и человеку будет
 % приятно, поскольку ему нравится эта вещь
)
```

В листинге 23.3 предикаты `sad` (грустит) и `likes` (любит) определены как операционные. Теперь может быть получено операционное определение предиката `gives` путем удаления из дерева доказательства всех узлов, кроме узлов, указанных как “операционные”, и корневого узла. Это приводит к получению такого результата:

```
gives(Person, Person, Thing) <== % Человек вручает себе какую-то вещь,
(sad(Person), % если ему грустно
 likes(Person, Thing) % и ему нравится эта вещь
)
```

Таким образом, достаточное условие `Condition` для предиката `gives( Person, Person, Thing)` состоит в следующем:

```
Condition = (sad(Person), likes(Person, Thing))
```

Теперь это новое определение можно ввести в первоначальную программу, как показано ниже.

```
asserta((gives(Person, Person, Thing) :- Condition))
```

В результате получено следующее новое предложение, касающееся предиката `gives`, для которого требуется оценка только операционных предикатов:

```
gives(Person, Person, Thing) :-
 sad(Person),
 likes(Person, Thing).
```

Благодаря обобщению заданного экземпляра `gives( john, john, chocolate)` было сформировано определение (в операционных терминах) как один из общих случаев применения предиката `gives` для описания одного из способов самоутешения! Еще одним случаем применения такого понятия может служить результат, полученный на основании следующего примера:

```
gives(john, annie, tennis_racket). % Джон вручает Энни теннисную ракетку
```

В данном случае метод обобщения на основе объяснения приводит к получению такого предложения:

```
gives(Person1, Person2, Thing) :- % Один человек вручает другому вещь,
 likes(Person1, Person2), % если он любит этого человека,
 needs(Person2, Thing). % а последнему требуется эта вещь
```

Проблемная область с описанием движений лифта, приведенная в листинге 23.3, является немного более сложной, поэтому проведем с ней эксперименты после реализации метода `EBG` на языке `Prolog`.

Метод `EBG` можно запрограммировать как двухэтапный процесс: на первом этапе вырабатывается дерево доказательства для данного примера, а на втором это дерево доказательства обобщается и из него извлекаются “операционные узлы”. Для этой цели можно применить описанный выше метаинтерпретатор, который формирует деревья доказательства. Но фактически эти два этапа не требуются. Более удобный способ состоит в том, чтобы откорректировать простой метаинтерпретатор, приведенный в листинге 23.1, таким образом, чтобы операции обобщения чередовались с операциями доказательства заданного экземпляра. Модифицированный подобным образом метаинтерпретатор, который реализует рассматриваемый метод `EBG`, будет именоваться `ebg` и теперь будет иметь три параметра:

```
ebg(Goal, GenGoal, Condition)
```

где `Goal` — заданный пример, который должен быть доказан, `GenGoal` — обобщенная цель, `Condition` — полученное путем логического вывода достаточное условие

для GenGoal, сформулированное в терминах операционных предикатов. Такой обобщающий метаинтерпретатор приведен в листинге 23.4. Для проблемной области с описанием процесса получения подарков (см. листинг 23.3) метаинтерпретатор ebg может быть вызван следующим образом:

```
?- ebg(gives(john, john, chocolate), gives(X, Y, Z), Condition).
X = Y
Condition = (sad(X), likes(X, Z))
```

#### Листинг 23.4. Программа обобщения на основе объяснения

```
% ebg(Goal, GeneralizedGoal, SufficientCondition) если
% достаточное условие SufficientCondition, заданное в терминах операционных
% предикатов, гарантирует, что GeneralizedGoal, обобщение цели Goal,
% является истинным. Обобщение цели GeneralizedGoal не должно быть задано
% с помощью переменной

ebg(true, true, true) :- !.

ebg(Goal, GenGoal, GenGoal) :-
 operational(GenGoal),
 call(Goal).

ebg((Goal1,Goal2), (Gen1,Gen2), Cond) :- !,
 ebg(Goal1, Gen1, Cond1),
 ebg(Goal2, Gen2, Cond2),
 and(Cond1, Cond2, Cond). % Cond = (Cond1,Cond2) в упрощенном виде

ebg(Goal, GenGoal, Cond) :-
 not operational(Goal),
 clause(GenGoal, GenBody),
 copy_term((GenGoal,GenBody), (Goal,Body)), % Новая копия
 % терма (GenGoal,GenBody)
 ebg(Body, GenBody, Cond).

% and(Cond1, Cond2, Cond) если
% условие Cond представляет собой конъюнкцию условий Cond1 и Cond2
% (возможно, в упрощенном виде)

and(true, Cond, Cond) :- !. % (true and Cond) <==> Cond

and(Cond, true, Cond) :- !. % (Cond and true) <==> Cond

and(Cond1, Cond2, (Cond1, Cond2)).
```

Теперь попытаемся исследовать проблемную область с описанием движений лифта. Предположим, что цель, которая должна быть решена и обобщена, состоит в определении последовательности движений Moves, которые переводят лифт с третьего этажа на шестой, как показано ниже.

```
go(3, 6, Moves)
```

Теперь можно вызвать на выполнение программу ebg и получить результирующую обобщенную цель и ее условие следующим образом:

```
?- Goal = go(3, 6, Moves),
 GenGoal = go(Level1, Level2, GenMoves),
 ebg(Goal, GenGoal, Condition),
 asserta((GenGoal :- Condition)).
Goal = go(3, 6, [up, up, up])
GenGoal = go(Level1, Level2, [up, up, up])
Condition = (0+1 + 1 + 1:= Level2 - Level1)
Moves = [up, up, up]
...
```

Результирующее новое предложение, касающееся предиката `go`, состоит в следующем:

```
go(Level1, Level2, [up, up, up]) :-
 0 + 1 + 1 + 1 := Level2 - Level1.
```

С помощью метода `EBG` простая операция перемещения лифта вверх на три этажа была обобщена как операция перемещения вверх между любыми двумя этажами, находящимися друг от друга на расстоянии в три этажа. Чтобы решить задачу по достижению цели `go( 3, 6, Moves)`, первоначальная программа выполняет поиск среди последовательностей действий по подъему (`up`) и спуску (`down`). А с помощью вновь выведенного предложения задача перемещения вверх с одного этажа на другой на расстояние в три этажа (например, `go( 7, 10, Moves)`) решается немедленно, без какого-либо поиска.

Метаинтерпретатор `ebg`, приведенный в листинге 23.4, снова является одной из производных простого метаинтерпретатора, показанного в листинге 23.1. Этот новый метаинтерпретатор вызывает встроенную процедуру `copy_term`. Вызов этой процедуры в форме

```
copy_term(Term, Copy)
```

позволяет сформировать для заданного термина `Term` его копию `Copy` с переименованными переменными. Это удобно, если требуется сохранить терм в его первоначальном виде и в то же время обеспечить его обработку таким образом, чтобы переменные этого термина могли стать конкретизированными. При такой обработке можно использовать копию термина, притом что переменные в первоначальном терме остаются незатронутыми.

Последнее предложение в процедуре `ebg` заслуживает дополнительного пояснения. Вызов

```
clause(GenGoal, GenBody)
```

обеспечивает выборку предложения, которое может использоваться для доказательства обобщенной цели. Это означает, что метаинтерпретатор фактически предпринимает попытку доказать обобщенную цель. Но следующая строка налагает некоторое ограничение на эту операцию:

```
copy_term((GenGoal, GenBody), (Goal, Body))
```

В ней предъявляется требование, чтобы переменные `GenGoal` и `Goal` были согласованными. Согласование выполняется над копией `GenGoal`, поэтому переменные в этой общей цели остаются незатронутыми. Причина, по которой требуется такое согласование, состоит в том, что выполнение обобщенной цели должно ограничиваться такими альтернативными ветвями дерева доказательства (предложениями), которые являются применимыми для заданного примера (`Goal`). Благодаря этому выполнением обобщенной цели управляет сам пример. Подобное управление и составляет суть компиляции программы в стиле `EBG`. Без такого управления возникает вероятность того, что будет найдено такое доказательство для `GenGoal`, которое не применимо для `Goal`. В подобных случаях обобщение полностью не соответствует примеру.

После того как пример обобщен и сформулирован в терминах операционных предикатов, он может быть добавлен в виде нового предложения к программе для использования при поиске ответов на будущие подобные вопросы путем оценки только операционных предикатов. Таким образом, метод компиляции `EBG` преобразует программу в “операционный” язык. Откомпилированная программа может выполняться интерпретатором, который “знает” только данный операционный язык. Одним из возможных преимуществ этого может стать создание более эффективной программы. Ее эффективность может быть повышена в следующих двух направлениях: во-первых, обработка операционных предикатов может осуществляться проще по сравнению с другими предикатами, и, во-вторых, последовательность обработки предикатов, показанная на примере, может оказаться более подходящей по сравнению с той, которая была предусмотрена в первоначальной программе, поскольку в откомпилированном определении вообще не появляются ветви, которые приводят к неудачному

завершению. При компиляции программы с помощью метода EBG необходимо следить за тем, чтобы новые предложения не вступали в противоречие с первоначальными предложениями неконтролируемым способом.

## Упражнение

**23.3.** Может показаться, что такой же эффект компиляции, достигаемый в методе EBG, может быть получен без использования примера, просто путем замены целей в первоначальном определении понятия их подцелями, взятыми из соответствующих предложений теории проблемной области, до тех пор, пока все цели не будут сокращены до операционных подцелей. Такая процедура называется *развертыванием цели* (цель “развертывается” в подцели). Обсудите эту идею по отношению к методу EBG и покажите, что в этом методе нельзя обойтись без управления с помощью некоторого примера. Кроме того, покажите, что новые определения понятий, выработанные с помощью метода EBG, представляют собой обобщение заданных примеров и поэтому не обязательно эквивалентны первоначальной программе (определения нового понятия могут быть неполными).

## 23.4. Объектно-ориентированное программирование

В системе Prolog можно легко реализовать объектно-ориентированный стиль программирования. В данном разделе применение такого стиля демонстрируется на примере простого интерпретатора для объектно-ориентированных программ. При использовании объектно-ориентированного подхода программа рассматривается как состоящая из объектов, которые передают друг другу сообщения. Вычисление происходит в результате формирования объектом ответа на полученное сообщение. Каждый объект имеет собственную область памяти и некоторые процедуры, называемые *методами*. Кроме того, объекты могут наследовать методы от других объектов. Применяемый при этом механизм аналогичен механизму наследования в методе представления данных на основе фреймов (см. главу 15). Объект отвечает на сообщение, вызывая на выполнение один из своих методов, а сообщение определяет, какой метод должен быть выполнен. Действие по передаче сообщения объекту фактически представляет собой своего рода вызов процедуры. Реализация объектно-ориентированной программы по сути является моделированием процессов передачи сообщений между объектами и формирования объектами ответов на сообщения.

Чтобы ознакомиться с тем, как этот принцип практически осуществляется в системе Prolog, рассмотрим в качестве первого примера объектно-ориентированную программу, касающуюся геометрических фигур. Одним из объектов этой программы является прямоугольник, а в состав данного объекта могут входить процедуры для описания прямоугольника и вычисления его площади. Если объекту прямоугольника передается сообщение `area( A )`, он отвечает, вычисляя площадь прямоугольника, и переменная `A` становится конкретизированной значением этой площади. В рассматриваемой реализации объект будет представлен как отношение

`object( Object, Methods )`

где `Object` — терм, который именуется объектом (и, возможно, задает его параметры), а `Methods` — список термов Prolog, определяющих методы. Термы в этом списке имеют форму предложений Prolog, т.е. обычных фактов и правил Prolog (если не считать того, что они не оканчиваются точкой). В рассматриваемой реализации Prolog определение любого объекта может задавать целый класс объектов, таких как класс всех прямоугольников, определяемых с помощью предиката `rectangle( Length, Width )`. В таком случае конкретный прямоугольник со сторонами 4 и 3 определяется

с помощью терма `rectangle( 4, 3)`. Это означает, что в общем объект `rectangle( Length, Width)` с двумя методами, `area` и `describe`, может быть определен следующим образом:

```
object(rectangle(Length, Width),
 [(area(A) :-
 A is Length * Width),
 (describe :-
 write('Rectangle of size '),
 write(Length * Width))]).
```

В данной реализации процесс передачи сообщения объекту может быть промоделирован с помощью такой процедуры:

```
send(Object, Message)
```

Для того чтобы объект прямоугольника со сторонами 4 и 3 описал себя и вычислил свою площадь, ему достаточно передать соответствующие сообщения:

```
?- Rec1 = rectangle(4, 3),
 send(Rec1, describe),
 send(Rec1, area(Area)).
Rectangle of size 4*3
Area = 12
```

Составить программу для процедуры `send( Object, Message)` можно достаточно просто. Вначале необходимо обеспечить выборку методов объекта `Object`. Эти методы фактически определяют программу `Prolog`, которая является локальной по отношению к `Object`. Затем нужно вызвать на выполнение эту программу, задав ей в качестве цели сообщение `Message`. Программа, представленная в виде списка, состоит из компонентов в форме `Head :- Body` или просто `Head` в случае, если тело является пустым. Чтобы выполнить программу с целью `Message`, требуется найти голову предложения, которая согласуется с целью `Message`, а затем выполнить соответствующее тело предложения с помощью собственного интерпретатора `Prolog`.

Прежде чем реализовать этот замысел в программе, необходимо рассмотреть еще один существенно важный механизм объектно-ориентированного программирования — наследование методов в соответствии с отношением “*isa*” (является) между объектами. Например, квадрат “*isa*” прямоугольником. Для вычисления его площади в объекте квадрата может использоваться такая же формула, как и в объектах прямоугольников. Поэтому для квадрата не требуется его собственный метод `area`; он может унаследовать этот метод от класса прямоугольников. Для того чтобы получить такую возможность, необходимо определить объект `square` и дополнительно указать, что он также является прямоугольником.

```
object(square(Side),
 [(describe :-
 write('Square with side '),
 write(Side))]).
isa(square(Side), rectangle(Side, Side)).
```

Это дает возможность успешно выполнить следующий запрос:

```
?- send(square(5), area(Area)).
Area = 25
```

Сообщение `area( Area)` обрабатывается таким образом: вначале выполняется поиск метода `area( Area)` среди методов объекта `square( 5)`, но найти его не удается. Затем с помощью отношения `isa` успешно выполняется поиск этого метода в *суперобъекте* квадрата — в прямоугольнике `rectangle( 5, 5)`. Суперобъект имеет соответствующий метод `area`, который вызывается на выполнение.

Интерпретатор для объектно-ориентированных программ, разработанный в соответствии с замыслом, описанным в данном разделе, приведен в листинге 23.5, а в листинге 23.6 приведена законченная объектно-ориентированная программа, относящаяся к геометрическим фигурам.

### Листинг 23.5. Простой интерпретатор для объектно-ориентированных программ

```
% Интерпретатор для объектно-ориентированных программ

% send(Message, Object) если
% требуется найти методы объекта Object и вызвать на выполнение метод,
% который соответствует сообщению Message

send(Object, Message) :-
 get_methods(Object, Methods), % Найти методы объекта Object
 process(Message, Methods). % Вызвать на выполнение соответствующий метод

get_methods(Object, Methods) :-
 object(Object, Methods). % Собственные методы

get_methods(Object, Methods) :-
 isa(Object, SuperObject),
 get_methods(SuperObject, Methods). % Унаследованные методы

process(Message, [Message | _]). % Использовать факт

process(Message, [(Message :- Body) | _]) :- % Использовать правило
 call(Body).

process(Message, [_ | Methods]) :-
 process(Message, Methods).
```

### Листинг 23.6. Объектно-ориентированная программа, относящаяся к геометрическим фигурам

```
/*
 polygon([Side1, Side2, ...])
 /
 rectangle(Length, Width) reg_polygon(Side, N)
 / \
 square(Side) pentagon(Side)
*/

object(polygon(Sides),
 [(perimeter(P) :-
 sum(Sides, P))]).

object(reg_polygon(Side, N),
 [(perimeter(P) :-
 P is Side * N),
 (describe :- write('Regular polygon'))]).

object(square(Side),
 [(describe :-
 write('Square with side '),
 write(Side))]).

object(rectangle(Length, Width),
 [(area(A) :-
 A is Length * Width),
 (describe :-
 write('Rectangle of size '),
 write(Length * Width))]).

object(pentagon(Side),
 [(describe :- write('Pentagon'))]).

isa(square(Side), rectangle(Side, Side)).

isa(square(Side), reg_polygon(Side, 4)).
```



```

isa(rectangle(Length, Width), polygon([Length,Width,Length,Width])).

isa(pentagon(Side), reg_polygon(Side, 5)).

isa(reg_polygon(Side, N), polygon(L)) :-
 makelist(Side, N, L).

% makelist(Item, N, List) если
% List - это список, в котором элемент Item появляется N раз

makelist(_, 0, []).

makelist(Item, N, [Item | List]) :-
 N > 0, N1 is N - 1,
 makelist(Item, N1, List).

% sum(ListOfNumbers, Sum) если
% Sum - это сумма чисел в списке ListOfNumbers

sum([], 0).

sum([Number | Numbers], Sum) :-
 sum(Numbers, Sum1),
 Sum is Sum1 + Number.

```

---

До сих пор мы еще не упоминали о проблеме множественного наследования. Подобная проблема возникает, если отношение `isa` определяет такую решетку связей между объектами, что некоторые объекты имеют больше одного родительского объекта, как, например, объект `square` (см. листинг 23.6). Поэтому существует вероятность того, что некоторый метод может быть унаследован объектом больше чем от одного родительского объекта. В таком случае необходимо найти ответ на вопрос о том, какой из нескольких потенциально наследуемых методов должен использоваться в дочернем объекте. В программе, приведенной в листинге 23.5, предусмотрен поиск одного из применимых методов среди объектов в графе, определяемым отношением `isa`. В листинге 23.5 применяется простая стратегия поиска в глубину, но более подходящими могут оказаться некоторые другие стратегии. Например, поиск в ширину гарантирует, что будет всегда использоваться “ближайший наследуемый” метод.

В качестве дополнительной иллюстрации стиля объектно-ориентированного программирования, который поддерживается интерпретатором, приведенным в листинге 23.5, рассмотрим ситуацию, показанную на рис. 23.1. На этом рисунке изображен мир робота: определенное количество блоков, расставленных на столе в виде столбиков. На потолке установлена телекамера, которая снимает вид сверху объектов, находящихся на столе. Для простоты предположим, что все блоки представляют собой кубики со сторонами, равными 1. Блоки имеют имена `a`, `b`, ..., а камера позволяет распознавать по именам те блоки, которые не закрыты сверху, и определять их координаты по отношению к осям `x` и `y`. Предположим, что перед нами стоит задача определения местонахождения объектов, т.е. их координат `x`, `y` и `z`. С каждым блоком связана некоторая локальная информация, поэтому известно, какой блок (если он существует) находится непосредственно над ним или под ним. Поэтому координаты `x - y` произвольного блока `B` могут быть получены одним из двух способов, описанных ниже.

1. Если блок не закрыт сверху другим блоком, то он может быть обнаружен с помощью камеры, которая определит координаты `x - y` этого блока. Такая задача решается путем передачи камере сообщения `look( B, X, Y)`.
2. Если блок `B` расположен под некоторым блоком `B1`, то он находится вместе с ним в одном и том же столбике, а все блоки одного столбика имеют одинаковые координаты `x - y`. Поэтому, чтобы определить координаты `x - y` блока `B`, необходимо отправить сообщение `xy_coord( X, Y)` блоку `B1`.

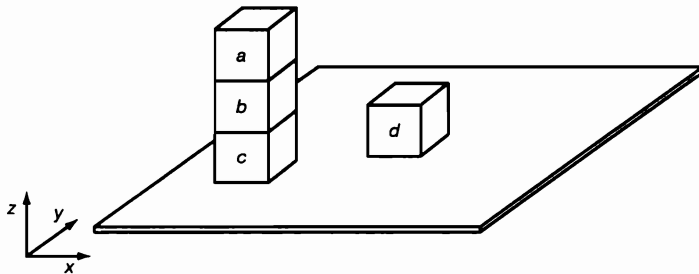


Рис. 23.1. Мир робота

Аналогичные рассуждения позволяют найти способ определения координаты  $z$  любого блока. Если некоторый блок  $B$  стоит на столе, то его координата  $z$  равна 0, а если  $B$  стоит на другом блоке  $B1$ , то достаточно передать сообщение `z_coord( Z1)` блоку  $B1$  и сложить полученную высоту  $Z1$  с высотой  $B1$ , определив тем самым значение координаты  $z$  блока  $B$ . Такие операции выполняются в программе, приведенной в листинге 23.7, где блоки  $a$ ,  $b$  и т.д. определены как экземпляры класса блоков `block( BlockName)`, от которого они наследуют методы `xy_coord` и `z_coord`.

#### Листинг 23.7. Объектно-ориентированная программа, относящаяся к миру робота

% Мир робота: стол, блоки и видеокамера

```
object(camera,
 [look(a, 1, 1), % Найти координаты x - y видимого блока
 look(d, 3, 2),
 xy_coord(2, 2), % Координаты x - y видеокамеры
 z_coord(20)]). % Координата z видеокамеры

object(block(Block),
 [(xy_coord(X, Y) :-
 send(camera, look(Block, X, Y))),
 (xy_coord(X, Y) :-
 send(Block, under(Block1)),
 send(Block1, xy_coord(X, Y))),
 (z_coord(0) :-
 send(Block, on(table))),
 (z_coord(Z) :-
 send(Block, on(Block1)),
 send(Block1, z_coord(Z1)),
 Z is Z1 + 1)]).

object(physical_object(Name),
 [(coord(X, Y, Z) :-
 send(Name, xy_coord(X, Y)),
 send(Name, z_coord(Z)))].

object(a, [on(b)]).
object(b, [under(a), on(c)]).
object(c, [under(b), on(table)]).
```

```
object(d, [on(table)])).
```

```
isa(a, block(a)).
isa(b, block(b)).
isa(c, block(c)).
isa(d, block(d)).
isa(block(Name), physical_object(Name)).
isa(camera, physical_object(camera)).
```

---

Завершим этот раздел общим комментарием, касающимся преимуществ объектно-ориентированного программирования. Рассматриваемые здесь в качестве примеров объектно-ориентированные программы могут быть реализованы в обычном стиле Prolog (фактически более компактно) без упоминания объектов и сообщений. Приведенные в этом разделе примеры иллюстрируют общие принципы, но они слишком малы для того, чтобы служить убедительным доказательством возможных преимуществ объектно-ориентированного программирования. Очевидно, что объектно-ориентированная модель является особенно удобной при написании программ для систем машинного моделирования, в которых физические компоненты действительно обмениваются своего рода сообщениями. Например, в производственном процессе в качестве сообщений могут рассматриваться потоки материалов, движущиеся от одного станка к другому. Еще одним преимуществом объектно-ориентированной модели является то, что она предоставляет удобную инфраструктуру для организации крупных программ. И память, и процедуры (методы) сосредоточиваются вокруг определений объектов, а механизм наследования предоставляет способ структуризации такой программы.

## Упражнения

- 23.4.** Определите, что происходит при интерпретации программами, приведенными в листингах 23.5 и 23.6, следующего вопроса, который предусматривает отpravку сообщения:

```
?- send(square(6), perimeter(P)).
```

Возникают ли альтернативные варианты для перебора с возвратами? Как можно модифицировать программу, приведенную в листинге 23.5, чтобы предотвратить возникновение нежелательных альтернативных вариантов?

- 23.5.** В листинге 23.5 показан интерпретатор, который является неэффективным, поскольку он затрачивает слишком много времени на поиск подходящих методов. Напишите компилятор для объектно-ориентированных программ, который компилирует определения объектов (в частности, их методы) в более эффективную программу Prolog. Такой компилятор должен, например, при компиляции программы, приведенной в листинге 23.6, вырабатывать примерно такие предложения:

```
area(square(Side), Area) :-
 Area is Side * Side.
area(rectangle(Length, Width), Area) :-
 Area is Length * Width.
```

Кроме того, переопределите процедуру `send( Object, Message)` таким образом, чтобы она использовала эти вновь сформированные предложения.

## 23.5. Программирование, управляемое шаблонами

### 23.5.1. Архитектура, управляемая шаблонами

В данном разделе под *системами, управляемыми шаблонами*, подразумевается одна из разновидностей архитектур для программных систем. Такая архитектура в большей степени подходит для решения задач определенных типов, чем системы, организованные обычным образом. К числу задач, которые естественным образом укладываются в рамки архитектуры, управляемой шаблонами, относятся многие приложения искусственного интеллекта, например экспертные системы. Основное различие между обычными системами и системами, управляемыми шаблонами, заключается в использовании разных механизмов вызова программных модулей. При обычной организации программы модули системы вызывают друг друга в соответствии с фиксированной, явной, заранее определенной схемой. В каждом модуле программы принимается решение о том, какой модуль должен быть выполнен следующим, путем явного вызова других модулей. В соответствии с этим поток выполнения является последовательным и детерминированным.

В противоположность этому при использовании организации программы, управляемой шаблонами, модули системы не вызываются явно другими модулями. Вместо этого их вызов осуществляется при обнаружении шаблонов, которые встречаются в их "среде определения данных". Поэтому такие модули и называются *модулями, управляемыми шаблонами*. Программа, управляемая шаблонами, представляет собой коллекцию модулей, управляемых шаблонами. Каждый модуль определен с помощью следующих компонентов.

1. Шаблон предварительного условия.
2. Действие, которое должно быть выполнено, если среда определения данных согласуется с этим шаблоном.

Операции вызова на выполнение модулей программы активизируются шаблонами, которые обнаруживаются в среде системы. Среда определения данных обычно именуется базой данных. Такая система условно изображена на рис. 23.2.

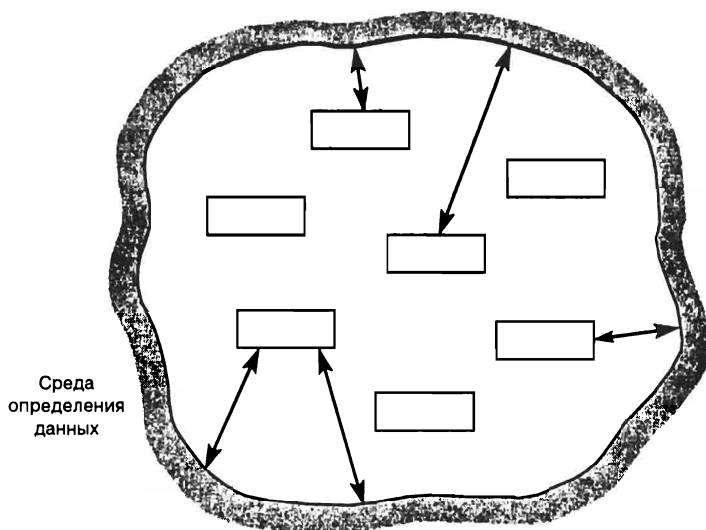


Рис. 23.2. Система, управляемая шаблонами. В виде прямоугольников показаны модули, управляемые шаблонами, а стрелки показывают, как активизируются модули при обнаружении шаблонов в данных

Изучение среды, показанной на рис. 23.2, позволяет сделать некоторые важные выводы. В этой системе не определена иерархия модулей и не даны явные указания на то, какие модули могут вызываться другими модулями. Модули скорее взаимодействуют с базой данных, а не непосредственно с другими модулями. Сама эта структура в принципе допускает возможность параллельного выполнения нескольких модулей, поскольку состояние базы данных может одновременно удовлетворять нескольким предварительным условиям и поэтому активизировать сразу несколько модулей. Следовательно, такая организация может также служить естественной моделью параллельного вычисления, в которой каждый модуль физически реализован с помощью отдельного процессора.

Архитектура, управляемая шаблонами, имеет несколько преимуществ. Одно из основных преимуществ состоит в том, что при проектировании системы не требуется заранее тщательно планировать и определять все связи между модулями. Поэтому возникает возможность проектировать и реализовывать каждый модуль относительно автономно. Благодаря этому обеспечивается весьма высокая степень модульности. Преимущество такой модульности выражается, например, в том, что удаление некоторого модуля из системы не обязательно приводит к фатальным последствиям. После их удаления система сохраняет способность решать свои задачи, но способ решения этих задач может стать иным. Это утверждение остается справедливым также в отношении добавления новых модулей и модификации существующих. Если аналогичные модификации проводятся в системах с обычной организацией, то возникает необходимость, по меньшей мере, модифицировать соответствующим образом вызовы между модулями.

Высокая степень модульности особенно желательна при создании систем со сложными базами знаний, поскольку невозможно заранее предвидеть все способы взаимодействия отдельных фрагментов знаний в этой базе. Архитектура, управляемая шаблонами, предоставляет наиболее приемлемое решение этой задачи: каждый фрагмент знаний, представленный в виде правила импликации, может рассматриваться как отдельный модуль, управляемый шаблонами.

Рассмотрим дополнительное уточнение основной схемы функционирования систем, управляемых шаблонами, с точки зрения их реализации. Изучение схемы, приведенной на рис. 23.2, позволяет сделать вывод, что для таких систем наиболее подходящей является параллельная реализация. Но предположим, что такая система должна быть реализована на обычном последовательном процессоре. В таком случае при возникновении ситуации одновременной активизации в базе данных шаблонов нескольких модулей возникает конфликт, связанный с определением того, какой из всех этих потенциально активных модулей должен быть фактически вызван на выполнение. Такое множество потенциально активных модулей называется *конфликтным множеством*. Для фактической реализации схемы (см. рис. 23.2) на последовательном процессоре требуется дополнительный программный модуль, называемый *управляющим модулем*. Управляющий модуль разрешает конфликт, выбирая и активизируя только один модуль из конфликтного множества. Процедура разрешения конфликтов может быть основана на простом правиле, которое предусматривает заранее определенное, постоянное упорядочение модулей.

Таким образом, основной жизненный цикл систем, управляемых шаблонами, состоит из трех описанных ниже этапов.

1. **Сопоставление с шаблоном.** Найти в базе данных все вхождения шаблонов условия вызова программных модулей, что влечет за собой формирование конфликтного множества.
2. **Разрешение конфликта.** Выбрать один из модулей в конфликтном множестве.
3. **Выполнение.** Вызвать на выполнение модуль, который был выбран на этапе 2.

Такая схема реализации показана на рис. 23.3.

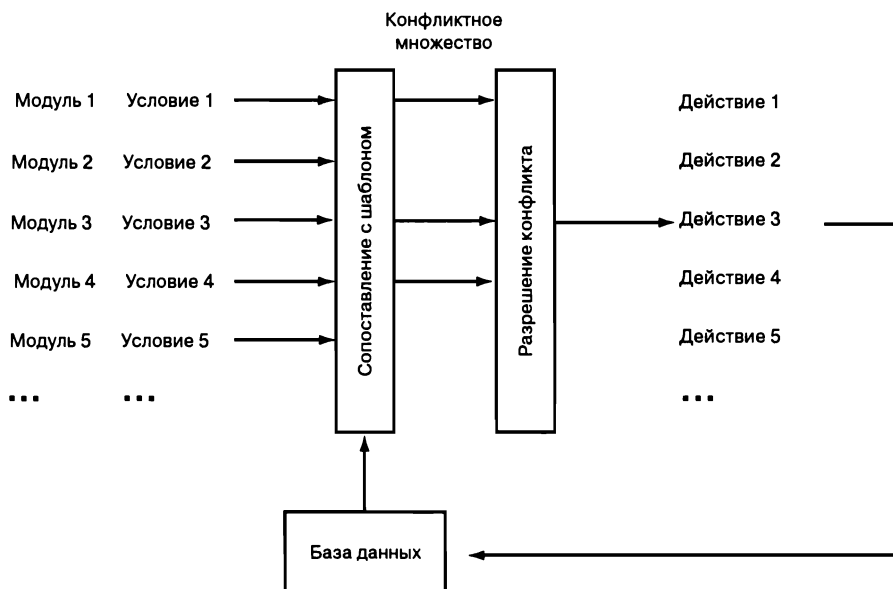


Рис. 23.3. Основной жизненный цикл систем, управляемых шаблонами. В этом примере в базе данных условию соответствуют шаблоны модулей 1, 3 и 4; для выполнения выбирается модуль 3

## 23.5.2. Программы Prolog, реализованные в виде систем, управляемых шаблонами

В качестве систем, управляемых шаблонами, могут рассматриваться сами программы Prolog. Соответствие между системой Prolog и системами, управляемыми шаблонами, можно определить без особого труда по перечисленным ниже признакам.

- В качестве модуля, управляемого шаблонами, может рассматриваться каждое предложение Prolog в программе. Условная часть этого модуля представляет собой голову предложения, а часть, определяющая действие, задается с помощью тела предложения.
- База данных системы представляет собой текущий список целей, которых пытается достичь система Prolog.
- Предложение вызывается на выполнение, если его голова согласуется с первой целью в базе данных.
- Выражение “выполнить действие модуля (тело предложения)” означает — заменить первую цель в базе данных списком целей в теле предложения (после надлежащей конкретизации переменных).
- Процесс вызова модулей является недетерминированным в том смысле, что с первой целью в базе данных могут согласовываться головы нескольких предложений, но только одна из них в принципе может быть выполнена. Такой недетерминированный вызов фактически реализован в системе Prolog с помощью перебора с возвратами.

### 23.5.3. Пример разработки программ, управляемых шаблонами

Системы, управляемые шаблонами, могут также рассматриваться как особый стиль написания программ и размышления над задачами, который называется *программированием, управляемым шаблонами*.

Чтобы показать справедливость этого утверждения, рассмотрим простейшую задачу составления программы: вычислить наибольший общий делитель  $D$  двух целых чисел  $A$  и  $B$ . Классический алгоритм Евклида, применяемый для решения этой задачи, может быть записан, как показано ниже.

Чтобы вычислить наибольший общий делитель  $D$  целых чисел  $A$  и  $B$ , необходимо осуществить описанные ниже действия.

До тех пор, пока  $A$  и  $B$  не станут равными, повторно выполнять (в цикле) такую операцию:

- если  $A > B$ , то заменить  $A$  разностью  $A - B$ , в противном случае заменить  $B$  разностью  $B - A$ .

После останова этого цикла  $A$  и  $B$  становятся равными; теперь наибольший общий делитель  $D$  равен  $A$  (или  $B$ ).

Тот же самый процесс можно определить с помощью двух модулей, управляемых шаблонами, как показано ниже.

#### Модуль 1

- **Условие.** В базе данных есть такие два числа,  $X$  и  $Y$ , что  $X > Y$ .
- **Действие.** Заменить в базе данных  $X$  разностью  $X - Y$ .

#### Модуль 2

- **Условие.** В базе данных есть число  $X$ .
- **Действие.** Вывести  $X$  на внешнее устройство и остановиться.

Каждый раз, когда выполняется условие модуля 1, выполняется и условие модуля 2, поэтому возникает конфликт. Этот конфликт разрешается с помощью простого правила управления: модуль 1 всегда имеет более высокий приоритет, чем модуль 2. Первоначально база данных содержит два разных числа,  $A$  и  $B$ , а после завершения работы — только одинаковые числа.

Приятным сюрпризом является то, что рассматриваемая программа, управляемая шаблонами, фактически решает более общую задачу — вычисление наибольшего общего делителя для любого множества целых чисел. Если в базе данных хранится несколько целых чисел, система выводит наибольший общий делитель для всех них. На рис. 23.4 показана возможная последовательность изменений в базе данных, прежде чем будет получен результат, при том условии, что база данных первоначально содержит четыре числа: 25, 10, 15, 30. Обратите внимание на то, что предварительное условие модуля может быть удовлетворено в нескольких местах базы данных.

В данной главе рассматривается интерпретатор для простого языка определения систем, управляемых шаблонами, и на реальных примерах иллюстрируется одно из направлений программирования, управляемого шаблонами.





```
[num X, num Y, X > Y] --->
[NewX is X - Y, replace(num X, num NewX)].

[num X] ---> [write(X), stop].
```

% Начальное состояние базы данных

```
num 25.
num 10.
num 15.
num 30.
```

---

Простейший способ реализации такого языка, управляемого шаблонами, состоит в использовании собственного встроенного механизма поддержки базы данных системы Prolog. Добавление объекта в базу данных и удаление объекта может осуществляться с помощью следующих встроенных процедур:

```
assertz(Object)
retract(Object)
```

Замена одного объекта другим также осуществляется достаточно просто:

```
replace(Object1, Object2) :-
 retract(Object1), !,
 assertz(Object2).
```

В этом предложении оператор отсечения используется для того, чтобы предикат retract не мог удалить из базы данных (в результате перебора с возвратами) больше одного объекта.

Небольшой интерпретатор для программ, управляемых шаблонами, разработанный в соответствии с этим замыслом, приведен в листинге 23.9. Возможно, этот интерпретатор в определенных отношениях является слишком упрощенным. В частности, правило разрешения конфликтов в этом интерпретаторе является чрезвычайно простым и жестким, поскольку согласно этому правилу всегда выполняется первый потенциально активный модуль, управляемый шаблонами (в порядке их местонахождения в программе). Поэтому управление порядком выполнения со стороны программиста сводится лишь к упорядочению модулей. Первоначальное состояние базы данных для этого интерпретатора должно быть сформировано путем вставки фактов Prolog, возможно, в результате получения консультаций из файла. Затем выполнение программы активизируется в результате вызова следующей цели:

```
?- run.
```

---

### Листинг 23.9. Небольшой интерпретатор для программ, управляемых шаблонами

---

```
% Небольшой интерпретатор для программ, управляемых шаблонами.
% Манипуляции с базой данных системы осуществляются с помощью
% предикатов assert/retract

:- op(800, xfx, --->).

% run: выполнять порождающие правила, заданные в форме Condition ---> Action,
% до тех пор, пока не будет активизировано действие 'stop'

run :-
 Condition ---> Action, % Порождающее правило
 test(Condition), % Предварительное условие выполнено?
 execute(Action).

% test([Condition1, Condition2, ...]) если результаты проверки всех условий
% являются истинными

test([]). % Пустое условие
```

```

test([First|Rest]) :- % Проверить конъюнкцию условий
 call(First),
 test(Rest).

% execute([Action1, Action2, ...]): выполнить список действий

execute([stop]) :- !. % Прекратить выполнение

execute([]) :- % Пустое действие (цикл выполнения завершен)
 run. % Перейти к следующему циклу выполнения

execute([First | Rest]) :-
 call(First),
 execute(Rest).

replace(A, B) :- % Заменить в базе данных предложение A предложением B
 retract(A), !, % Извлечь только один экземпляр
 assert(B).

```

---

## 23.5.5. Возможные усовершенствования

Рассматриваемый в данном разделе простой интерпретатор для программ, управляемых шаблонами, вполне позволяет проиллюстрировать некоторые идеи программирования, управляемого шаблонами. Но для использования в более сложных приложениях он должен быть доработан в определенных отношениях. Ниже приведены некоторые критические замечания и указания по его усовершенствованию.

В данном интерпретаторе разрешение конфликтов сводится к использованию постоянной, заранее определенной последовательности модулей. Но часто требуются более гибкие схемы. Для обеспечения более сложного управления необходимо организовать поиск всех потенциально активных модулей и передачу их в специальный, программируемый пользователем модуль управления.

Если база данных велика и в программе содержится много модулей, управляемых шаблонами, то процесс сопоставления с шаблонами может стать чрезвычайно неэффективным. В этом отношении эффективность может быть повышена за счет более продуманной организации базы данных. Для этого может потребоваться индексация информации в базе данных, сегментирование информации на подбазы или сегментирование множества управляемых шаблонами модулей на подмножества. Преимущество сегментирования состоит в том, что в любой конкретный момент времени доступ предоставляется только к подмножеству базы данных или к определенной части модулей, поэтому сопоставление с шаблоном ограничивается только этим подмножеством или этой частью. Безусловно, в таком случае требуется более развитый механизм управления, который позволил бы обеспечить переход от одного из таких подмножеств к другому по принципу активизации подмножеств или перевода их из активного состояния в пассивное. Для этого могут использоваться своего рода метаправила.

К сожалению, рассматриваемый интерпретатор в том виде, в каком он реализован в программе, исключает возможность применять какие-либо методы перебора с возвратами, поскольку в нем предусмотрен способ непосредственного манипулирования базой данных с помощью предикатов `assertz` и `retract`. Поэтому мы не имеем возможности изучать альтернативные пути выполнения. Указанный недостаток можно устранить, используя другую реализацию базы данных, в которой не применяются предикаты `assertz` и `retract` языка Prolog. Один из способов решения такой задачи может предусматривать представление всего состояния базы данных с помощью термина Prolog, передаваемого в качестве параметра в процедуру `run`. Наиболее простая возможность реализации такого решения состоит в организации этого термина как списка объектов в базе данных. В подобном случае верхний уровень интерпретатора может выглядеть следующим образом:

```

run(State) :-
 Condition ---> Action,

```

```
test(Condition, State),
execute(Action, State).
```

Затем процедура `execute` вычисляет новое состояние и вызывает процедуру `run` с этим новым состоянием.

## Проект

Реализуйте интерпретатор для программ, управляемых шаблонами, в котором база данных не сопровождается в виде собственной внутренней базы данных Prolog (т.е. с помощью предикатов `assertz` и `retract`), а для ее сопровождения используется параметр процедуры, в соответствии с приведенным выше замечанием. В таком случае новый интерпретатор будет обеспечивать автоматический перебор с возвратами. Попробуйте спроектировать представление базы данных, которое обеспечивало бы эффективное сопоставление с шаблонами.

## 23.6. Простая программа автоматического доказательства теорем, реализованная в виде программы, управляемой шаблонами

Рассмотрим реализацию простой программы автоматического доказательства теорем в виде системы, управляемой шаблонами. Эта программа автоматического доказательства будет основана на принципе резолюции; он представляет собой один из широко применяемых методов автоматического доказательства теорем. В этом описании мы ограничимся доказательством теорем простой логики высказываний лишь для иллюстрации используемого принципа, хотя описанный механизм резолюции может быть легко расширен для обработки выражений исчисления предикатов первого порядка (логических формул, которые содержат переменные). Дело в том, что сама основная система Prolog представляет собой особую разновидность программы автоматического доказательства теорем на основе принципа резолюции. Задачу доказательства теоремы можно определить следующим образом: дана некоторая формула; необходимо показать, что она является теоремой. Это означает, что формула всегда является истинной, независимо от интерпретации символов, которые встречаются в этой формуле. Например, формула

$p \vee \sim p$

читается как “ $p$  или не  $p$ ” и всегда остается истинной, независимо от значения символа  $p$ . В качестве логических операторов будут использоваться следующие символы.

- $\sim$ . Отрицание, читается как “not” (не).
- $\&$ . Конъюнкция, читается как “and” (и).
- $\vee$ . Дизъюнкция, читается как “or” (или).
- $\Rightarrow$ . Импликация, читается как “implies” (следует из).

Для этих операторов определен такой приоритет, что “not” всегда связывает сильнее всех, за ним следует “and”, затем “or” и после этого “implies”.

Метод резолюции предусматривает использование такой последовательности доказательства: вначале к предполагаемой теореме применяется операция отрицания, а затем предпринимается попытка показать, что формула, полученная в результате отрицания, содержит противоречие. Если формула, полученная в результате отрицания, действительно является противоречивой, то первоначальная формула должна представлять собой тавтологию (т.е. быть истинной при любой интерпретации). Поэтому общая идея состоит в следующем: если продемонстрировано, что формула, по-

лученная в результате отрицания, содержит противоречие, это равносильно доказательству, что первоначальная формула является теоремой (всегда остается истинной). Процесс, направленный на обнаружение противоречия, состоит из последовательности этапов резолюции.

Проиллюстрируем этот принцип на простом примере. Предположим, что перед нами стоит задача доказать, что следующая формула исчисления высказываний является теоремой:

$$(a \Rightarrow b) \ \& \ (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$$

Эта формула читается следующим образом: “если  $b$  следует из  $a$  и  $c$  следует из  $b$ , то  $c$  следует из  $a$ ”.

Прежде чем можно будет начать процесс резолюции, необходимо перевести отрицаемую, предполагаемую теорему в более приемлемую форму, которая может использоваться в процессе резолюции. Такой подходящей формой является конъюнктивная нормальная форма, которая состоит из дизъюнктивных термов, соединенных знаками операции конъюнкции, и выглядит примерно так:

$$(p_1 \vee p_2 \vee \dots) \ \& \ (q_1 \vee q_2 \vee \dots) \ \& \ (r_1 \vee r_2 \vee \dots) \ \& \ \dots$$

В этой формуле все символы  $p$ ,  $q$  и  $r$  представляют собой простые высказывания или их отрицания. Эта форма называется также *формой представления в виде предложений* (под *предложением* здесь подразумевается конструкция, аналогичная простому грамматическому предложению в составе сложного), и каждый из ее дизъюнктов рассматривается как предложение. Поэтому составной терм  $(p_1 \vee p_2 \vee \dots)$  также является предложением. В эту форму может быть преобразована любая пропозициональная формула (формула исчисления высказываний). Для данной теоремы, рассматриваемой в качестве примера, такое преобразование может быть выполнено, как описано ниже. Сама эта теорема имеет следующий вид:

$$(a \Rightarrow b) \ \& \ (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$$

Отрицание данной теоремы выглядит таким образом:

$$\sim((a \Rightarrow b) \ \& \ (b \Rightarrow c) \Rightarrow (a \Rightarrow c))$$

Ниже перечислены известные правила эквивалентности, которые могут применяться при преобразовании этой формулы в конъюнктивную нормальную форму.

1. Выражение  $x \Rightarrow y$  эквивалентно  $\sim x \vee y$ .
2. Выражение  $\sim(x \vee y)$  эквивалентно  $\sim x \ \& \ \sim y$ .
3. Выражение  $\sim(x \ \& \ y)$  эквивалентно  $\sim x \vee \sim y$ .
4. Выражение  $\sim(\sim x)$  эквивалентно  $x$ .

Применив правило 1 к рассматриваемой формуле, получим следующее:

$$\sim(\sim((a \Rightarrow b) \ \& \ (b \Rightarrow c)) \vee (a \Rightarrow c))$$

С помощью правил 2 и 4 преобразуем формулу в такую форму:

$$(a \Rightarrow b) \ \& \ (b \Rightarrow c) \ \& \ \sim(a \Rightarrow c)$$

Несколько раз воспользовавшись правилом 1, получим:

$$(\sim a \vee b) \ \& \ (\sim b \vee c) \ \& \ \sim(\sim a \vee c)$$

Применив правило 2, наконец, получим требуемую форму представления в виде предложений:

$$(\sim a \vee b) \ \& \ (\sim b \vee c) \ \& \ a \ \& \ \sim c$$

Это предложение состоит из четырех термов:  $(\sim a \vee b)$ ,  $(\sim b \vee c)$ ,  $a$ ,  $\sim c$ . Теперь можно приступить к выполнению процесса резолюции.

Основной этап резолюции может быть выполнен в любой момент, когда имеются такие два предложения, что в одном из них встречается высказывание  $p$ , а в другом — высказывание  $\sim p$ . Допустим, что два таких предложения имеют следующий вид:

$$p \vee Y \text{ и } \sim p \vee Z$$

где  $p$  — высказывание, а  $Y$  и  $Z$  — формулы исчисления высказываний. В таком случае этап резолюции, выполненный над этими двумя предложениями, приводит к получению третьего предложения:

$Y \vee Z$

Можно показать, что это предложение логически следует из двух первоначальных предложений. Поэтому, добавив выражение  $(Y \vee Z)$  к рассматриваемой формуле, мы не изменим истинность этой формулы. Таким образом, в процессе резолюции вырабатываются новые предложения. Если же в конечном итоге будет получено “пустое предложение” (которое обычно обозначается как “nil”), это означает, что обнаружено противоречие. Пустое предложение nil формируется из двух предложений, имеющих следующую форму:

$x$  и  $\sim x$

Эта форма, безусловно, свидетельствует о противоречии.

На рис. 23.5 показан процесс резолюции, который начинается с ввода отрицания предполагаемой теоремы и оканчивается пустым предложением.

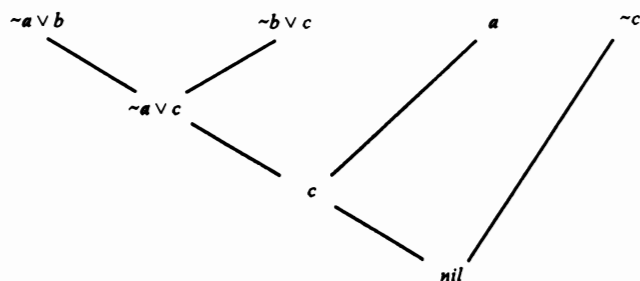


Рис. 23.5. Доказательство теоремы  $(a \Rightarrow b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$  по методу резолюции. Верхняя строка представляет собой отрицание данной теоремы в форме исчисления высказываний. Пустое предложение в нижней части свидетельствует о том, что отрицание этой теоремы приводит к противоречию

В листинге 23.10 показано, как можно реализовать этот процесс резолюции с помощью программы, управляемой шаблонами. Эта программа оперирует с предложениями, внесенными в базу данных. Весь ход осуществления принципа резолюции может быть сформулирован в виде процесса, управляемого шаблонами, как показано ниже.

Если

имеются два предложения,  $C1$  и  $C2$ , такие, что  $P$  представляет собой (дизъюнктивное) подвыражение  $C1$ , а  $\sim P$  — подвыражение  $C2$ ,

то

удалить  $P$  из  $C1$  (получив  $CA$ ), удалить  $\sim P$  из  $C2$  (получив  $CB$ ) и ввести в базу данных новое предложение:  $CA \vee CB$ .

**Листинг 23.10. Управляемая шаблонами простая программа для доказательства теорем с помощью метода резолюции**

```
% Следующая директива требуется для некоторых версий Prolog
:- dynamic clause/1, done/3.
```

```

% Порождающие правила для доказательства теоремы по методу резолюций

% Взаимоисключающие предложения
[clause(X), clause(~X)] --->
[write('Contradiction found'), stop].

% Удалить истинное предложение
[clause(C), in(P, C), in(~P, C)] --->
[retract(C)].

% Упростить предложение
[clause(C), delete(P, C, C1), in(P, C1)] --->
[replace(clause(C), clause(C1))].

% Этап резолюции, частный случай
[clause(P), clause(C), delete(~P, C, C1), not done(P, C, P)] --->
[assert(clause(C1)), assert(done(P, C, P))].

% Этап резолюции, частный случай
[clause(~P), clause(C), delete(P, C, C1), not done(~P, C, P)] --->
[assert(clause(C1)), assert(done(~P, C, P))].

% Этап резолюции, общий случай
[clause(C1), delete(P, C1, CA),
 clause(C2), delete(~P, C2, CB), not done(C1,C2,P)] --->
[assert(clause(CA v CB)), assert(done(C1, C2, P))].

% Последнее правило: процесс резолюции зашел в тупик
[] ---> [write('Not contradiction'), stop].

% delete(P, E, E1) если
% удаление дизъюнктивного подвыражения P из E приводит к получению E1
delete(X, X v Y, Y).

delete(X, Y v X, Y).

delete(X, Y v Z, Y v Z1) :-
 delete(X, Z, Z1).

delete(X, Y v Z, Y1 v Z) :-
 delete(X, Y, Y1).

% in(P, E) если P - дизъюнктивное подвыражение в E
in(X, X).

in(X, Y) :-
 delete(X, Y, _).

```

После оформления соответствующей конструкции на языке, управляемом шаблонами, получим следующее правило:

```

[clause(C1), delete(P, C1, CA),
 clause(C2), delete(~ P, C2, CB)] --->
[assertz(clause(CA v CB))].

```

Это правило требует небольшой доработки в целях предотвращения возможности повторного выполнения действий над некоторыми предложениями, что может привести к созданию новых копий уже существующих предложений. Программа, приведенная в листинге 23.10, регистрирует в базе данных операции, которые уже были выполнены, вводя в нее такой факт:

```
done(C1, C2, P)
```

В таком случае условные части правил позволяют распознавать и предотвращать подобные повторяющиеся действия.

Правила, приведенные в листинге 23.10, позволяют также учесть некоторые частные случаи, для которых могло бы иначе потребоваться явное представление пускового предложения. Кроме того, имеются два правила, которые упрощают предложения при любой возможности. Одно из этих правил распознает истинные предложения, такие как

```
a ∨ b ∨ ~a
```

и удаляет их из базы данных, поскольку они не могут применяться для обнаружения противоречия. Еще одно правило удаляет избыточные подвыражения. Например, это правило позволяет упростить следующее предложение:

```
a ∨ b ∨ a
```

преобразовав его в предложение  $a \vee b$ .

Остается нерешенным вопрос о том, как преобразовать заданную формулу исчисления высказываний в форму представления в виде предложений. Эта задача является несложной, и ее выполняет программа, приведенная в листинге 23.11. Процедура

```
translate(Formula)
```

преобразует формулу в множество предложений C1, C2 и т.д. и вводит эти предложения в базу данных, как показано ниже.

```
clause(C1).
clause(C2).
...
```

### Листинг 23.11. Программа преобразования формулы исчисления высказываний в множество предложений (вводимых в базу данных)

```
% Трансляция пропозициональной формулы в предложения (введенные в базу данных)

:- op(100, fy, ~). % Отрицание
:- op(110, xfy, &). % Конъюнкция
:- op(120, xfy, v). % Дизъюнкция
:- op(130, xfy, =>). % Импликация

% translate(Formula): транслитировать пропозициональную формулу в предложения и
% ввести в базу данных каждое полученное в результате предложение C
% как clause(C)

translate(F & G) :- % Транслитировать конъюнктивную формулу
!, % Красный оператор отсечения
 translate(F),
 translate(G).

translate(Formula) :- % Этап преобразования, выполняемый
 transform(Formula, NewFormula), % над формулой Formula
!, % Красный оператор отсечения
 translate(NewFormula).

translate(Formula) :- % Дальнейшие преобразования невозможны
 assert(clause(Formula)).

% Правила преобразования для пропозициональных формул
```

```

% transform(Formula1, Formula2) если
% Formula2 эквивалентна формуле Formula1, но ближе к форме представления
% в виде предложений

transform(~(~X), X). % Устранить двойное отрицание

transform(X => Y, ~X v Y). % Устранить импликацию

transform(~ (X & Y), ~X v ~Y). % Закон де Моргана

transform(~ (X v Y), ~X & ~Y). % Закон де Моргана

transform(X & Y v Z, (X v Z) & (Y v Z)). % Распределительный закон

transform(X v Y & Z, (X v Y) & (X v Z)). % Распределительный закон

transform(X v Y, X1 v Y) :-
 transform(X, X1). % Преобразовать подвыражение

transform(X v Y, X v Y1) :-
 transform(Y, Y1). % Преобразовать подвыражение

transform(~ X, ~ X1) :-
 transform(X, X1). % Преобразовать подвыражение

```

---

Теперь программа автоматического доказательства теорем, управляемая шаблонами, может быть вызвана на выполнение с помощью цели `run`. Итак, чтобы доказать предполагаемую теорему с помощью этих программ, необходимо преобразовать отрицание этой теоремы в форму представления в виде предложений и приступить к выполнению процесса резолюции. Применительно к теореме, рассматриваемой в качестве примера, это можно выполнить с помощью следующего вопроса:

```
?- translate(~((a=>b) & (b=>c) => (a=>c)), run.
```

Программа в ответ сообщит "Contradiction found" (Обнаружено противоречие), а это означает, что первоначальная формула является теоремой.

## Резюме

- В *метапрограммах* другие программы рассматриваются как данные. Метаинтерпретатор Prolog представляет собой интерпретатор для языка Prolog на языке Prolog.
- Можно легко написать *метаинтерпретаторы Prolog*, которые вырабатывают трассировки выполнения, формируют деревья доказательства и предоставляют другие дополнительные возможности.
- *Обобщение на основе объяснения* — это специальный метод компиляции программ. Он может рассматриваться как символическое выполнение программы, управляемое с помощью конкретного примера. Обобщение на основе объяснения представляет собой один из подходов к машинному обучению.
- *Объектно-ориентированная программа* состоит из объектов, которые передают друг другу сообщения. Объекты отвечают на сообщения, вызывая на выполнение свои методы. Методы могут быть также унаследованы от других объектов.
- *Программа, управляемая шаблонами*, представляет собой коллекцию управляемых шаблонами модулей, которые активизируются при обнаружении шаблонов, имеющих в "базе данных". Как системы, управляемые шаблонами, могут рассматриваться и сами программы Prolog. Наиболее естественной была бы параллельная реализация систем, управляемых шаблонами, а при последовательной реализации необходимо обеспечить разрешение конфликтов между модулями в конфликтном множестве.



- В данной главе реализован простой интерпретатор для программ, управляемых шаблонами, и применен для автоматического доказательства теорем логики высказываний по методу резолюции.
- В этой главе рассматривались следующие понятия:
  - метапрограммы, метаинтерпретаторы;
  - обобщение на основе объяснения;
  - объектно-ориентированное программирование;
  - объекты, методы, сообщения;
  - наследование методов;
  - системы, управляемые шаблонами, архитектура, управляемая шаблонами;
  - программирование, управляемое шаблонами;
  - модуль, управляемый шаблонами;
  - конфликтное множество, разрешение конфликтов;
  - доказательство теорем на основе резолюции, принцип резолюции.

## Дополнительные источники информации

Разработка метаинтерпретаторов Prolog является одним из традиционных направлений программирования на языке Prolog. Некоторые другие интересные примеры метаинтерпретаторов Prolog, кроме приведенных в данной главе, можно найти в [87], [147] и [154].

Идея обобщения на основе объяснения была впервые предложена для использования в области машинного обучения. Формулировки, применяемые в этой главе, соответствуют предложенным в [106]. Рассматриваемая здесь программа EBG аналогична приведенной в [72]. Эта программа представляет собой превосходную иллюстрацию того, насколько изящно может быть реализован на языке Prolog сложный символический метод. С помощью данной программы Кедар-Кабелли (Kedar-Cabelli) и Маккарти (McCarty) преобразовали многие страницы первоначального запутанного изложения метода EBG в сжатое, кристально ясное и непосредственно применимое описание.

Рассматриваемый в этой главе интерпретатор для объектно-ориентированных программ аналогичен описанному в [152]. В [107] представлен язык Prolog++, который является одной из эффективных реализаций объектно-ориентированной среды на базе Prolog.

Одной из классических книг по управляемым шаблонами системам является [165]. Многие направления дальнейшего развития базовой архитектуры, управляемой шаблонами, объединены под общим названием методологии “классной доски”. В [45] находится полезный сборник статей по методологии “классной доски”. Приведенное в данной главе в качестве иллюстрации приложение из области программирования, управляемого шаблонами, представляет собой простой пример автоматического доказательства теорем. Основы автоматического доказательства теорем в логике предикатов описаны во многих общих книгах по искусственному интеллекту, таких как [58], [60], [126] и [133].

# Некоторые различия между реализациями Prolog

Синтаксис Prolog, принятый в этой книге, соответствует традициям эдинбургской версии Prolog, которая была принята в большинстве реализаций Prolog, а также в стандарте ISO языка Prolog. Как правило, во многих реализациях Prolog предусмотрен целый ряд дополнительных средств. В целом в программах данной книги используется подмножество средств, предусмотренных в типичных реализациях и включенных в стандарт ISO. Тем не менее существуют небольшие различия между разными версиями Prolog, поэтому при выполнении программ, приведенных в этой книге, в конкретной версии Prolog может потребоваться внесение небольших изменений. В настоящем приложении описаны такие наиболее вероятные различия.

## Динамические и статические предикаты

В стандарте ISO и некоторых реализациях Prolog проводится различие между статическими и динамическими предикатами. В процессе компиляции статических предикатов создается более эффективный код по сравнению с динамическими предикатами. С другой стороны, допускается манипулирование с помощью предикатов `assert(a/z)` и `retract`, а также выборка с применением конструкции `clause(Head, Body)` только динамических предикатов. Предикат считается статическим, если в программе не указано иное с помощью объявления в следующей форме:

```
:- dynamic PredicateName(PredicateArity).
```

Например:

```
:- dynamic member(2).
```

Предикаты, введенные только с помощью `assert(a/z)`, по умолчанию рассматриваются как динамические.

## Предикаты `assert` и `retract`

В стандарте ISO предусмотрены только предикаты `asserta` и `assertz`, но не `assert`. Тем не менее практически во всех реализациях предусмотрен также предикат `assert`. Если же предикат `assert` не доступен, то вместо него можно использовать `assertz`. Могут быть также предусмотрены встроенные предикаты `retractall( Clause)` или `abolish( PredicateName/Arity)` для удаления всех предложений, которые относятся к указанному предикату.

## Неопределенные предикаты

В некоторых версиях Prolog попытки вызова предиката, вообще не определенного в программе, приводят к неудачному завершению, а в других версиях Prolog в таких случаях предусматривается вывод сообщения об ошибке. В версиях Prolog последне-

го типа можно обеспечить неудачное завершение неопределенных предикатов (без сообщений об ошибках) с помощью встроенного предиката `unknown( _, fail)`.

## Отрицание как недостижение успеха — операторы `not` и `\+`

В данной книге для обозначения операции отрицания как недостижения успеха применялась конструкция `not Goal`. Но во многих версиях Prolog (и в стандарте) предусмотрено использование другого обозначения (возможно, менее привлекательного) следующим образом:

```
\+Goal
```

Такое обозначение позволяет подчеркнуть, что данная операция определяет не настоящее логическое отрицание, а отрицание, определенное как недостижение цели. Для совместимости с другими версиями Prolog оператор `“not”` необходимо заменить оператором `“\+”` или (чтобы уменьшить для себя объем работы) ввести в программу `not` как предикат, определяемый пользователем (подробно об этом — в приложении Б).

## Предикат `name( Atom, CodeList)`

Этот предикат предусмотрен во многих реализациях, но не включен в стандарт (вместо этого задан предикат `atom_codes/2`). В разных версиях Prolog имеются небольшие различия, связанные с выполнением предиката `name` в частных случаях, например, когда первым параметром является число.

## Загрузка программ с помощью предикатов `consult` и `reconsult`

В различных реализациях применяются немного разные процедуры загрузки программ с помощью предикатов `consult` и `reconsult`. Различия обнаруживаются, если программы загружаются из нескольких файлов, а один и тот же предикат определен больше чем в одном файле (один из способов обработки состоит в том, что новые предложения, относящиеся к тому же предикату, могут просто добавляться к старым предложениям, а другой способ предусматривает загрузку только предложений из самого последнего файла и уничтожение предыдущих предложений, относящихся к тому же предикату).

## Модули

В некоторых реализациях Prolog программа может быть разделена на модули таким образом, чтобы имена предикатов были локальными по отношению к модулю, только если они не объявлены специально как видимые из других модулей. Это удобно при написании больших программ, если в них применяются разные предикаты с одинаковыми именами, а параметры этих предикатов имеют разный смысл в различных модулях.

# Некоторые часто используемые предикаты

Во многих программах в этой книге используются некоторые основные предикаты, такие как `member/2` и `conc/3`. Чтобы не было ненужного повторения, определения таких предикатов обычно не включаются в листинги этих программ. Но перед вызовом такой программы на выполнение в систему Prolog должны быть также загружены эти предикаты. Такое действие проще всего можно выполнить, используя для консультации (или компиляции) файл, подобный приведенному в данном приложении, который содержит определения указанных предикатов. В листинге Б.1 даны определения некоторых предикатов, которые могут уже быть включены в состав встроенных предикатов в зависимости от реализации Prolog. Например, в этот листинг включено также определение операции отрицания как недостижения цели, которое записывается в виде `not Goal`, для совместимости с такими версиями Prolog, в которых вместо этого используется обозначение `\+Goal`. При загрузке в систему Prolog определения предикатов, которые уже являются встроенными, система Prolog обычно выводит предупреждающее сообщение и игнорирует новое определение.

### Листинг Б.1. Определения часто используемых предикатов

```
% Файл frequent.pl: библиотека часто используемых предикатов
% Отрицание как недостижение цели
% Обычно эта операция реализуется в виде встроенного предиката, а для
% ее обозначения часто применяется префиксный оператор '\+', например
% в форме \+likes(mary,snakes)
% Приведенное ниже определение предназначено исключительно для обеспечения
% совместимости различных реализаций Prolog
:- op(900, fy, not).
not Goal :-
 Goal, !, fail
 ;
 true.

% once(Goal): предикат, который формирует только одно решение для цели Goal
% (в качестве него берется первое же полученное решение)
% Этот предикат может быть уже предоставлен в конкретной реализации
% как встроенный
once(Goal) :-
 Goal, !.

% member(X, List): предикат, который определяет принадлежность элемента X
% к списку List
member(X, [X | _]). % X - голова списка
member(X, [_ | Rest]) :-
 member(X, Rest). % X находится в теле списка

% conc(L1, L2, L3): список L3 - конкатенация списков L1 и L2
conc([],L,L).
```

```

conc([X | L1], L2, [X | L3]) :-
 conc(L1, L2, L3).

% del(X, L0, L): список L соответствует списку L0 с удаленным элементом X
% Примечание. Происходит удаление только одного экземпляра элемента X
% Вызов предиката на выполнение завершается неудачей, если элемент X
% отсутствует в списке L0

del(X, [X | Rest], Rest). % Удалить элемент X, если он составляет
 % голову списка

del(X, [Y | Rest0], [Y | Rest]) :-
 del(X, Rest0, Rest). % Удалить элемент X из хвоста списка

% subset(Set, Subset): предикат, который позволяет проверить, действительно ли
% список Set содержит все элементы списка Subset (проверить принадлежность
% подмножества к множеству)
% Примечание. Элементы списка Subset находятся в списке Set в том же порядке,
% что и в списке Subset

subset([], []).

subset([First | Rest], [First | Sub]) :- % Оставить элемент First
 % в подмножестве
 subset(Rest, Sub).

subset([First | Rest], Sub) :- % Удалить элемент First
 subset(Rest, Sub).

% set_difference(Set1, Set2, Set3): список Set3 определяется как разность
% множеств, представленных в виде списков Set1 и Set2
% Обычно этот предикат используется в такой форме: списки Set1 и Set2 задаются
% как входные параметры, а Set3 - как выходной

set_difference([], _, []).

set_difference([X | S1], S2, S3) :-
 member(X, S2), !, % Элемент X принадлежит к множеству S2
 set_difference(S1, S2, S3).

set_difference([X | S1], S2, [X | S3]) :- % Элемент X не принадлежит
 % к множеству S2
 set_difference(S1, S2, S3).

% length(List, Length): предикат, позволяющий определить длину Length
% списка List
% Примечание. Предикат length/2 может быть уже включен в состав встроенных
% предикатов конкретной версии Prolog
% В приведенном ниже определении используется хвостовая рекурсия
% Этот предикат может использоваться для эффективной выработки списков
% заданной длины

length(L, N) :-
 length(L, 0, N).

length([], N, N).

length([_ | L], N0, N) :-
 N1 is N0 + 1,
 length(L, N1, N).

% max(X, Y, Max): элемент Max - наибольший среди X и Y

max(X, Y, Max) :-
 X>=Y, !, Max = X
 ;

```

```

 Max = Y.

% min(X, Y, Min): элемент Min - наименьший среди X и Y

min(X, Y, Min) :-
 X <= Y, !, Min = X
 ;
 Min = Y.

% copy_term(T1, T2): терм T2 равен T1, за исключением того, что переменным
% присвоены другие имена
% Этот предикат может быть уже предоставлен в конкретной реализации как
% встроенный
% В соответствии с приведенным ниже определением процедуры вызов
% предиката copy_term происходит таким образом, что T2 согласуется с T1

copy_term(Term, Copy) :-
 asserta(term_to_copy(Term)),
 retract(term_to_copy(Copy)), !.
.....

```

## Глава 1

- 1.1.      а)    no  
            б)    X = pat  
            в)    X = bob  
            г)    X = bob, Y = pat
- 1.2.      а)    ?- parent( X, pat ).  
            б)    ?- parent( liz, X ).  
            в)    ?- parent( Y, pat ), parent( X, Y ).
- 1.3.      а)    happy(X) :-  
                  parent( X, Y ).  
            б)    hastwochildren( X ) :-  
                  parent( X, Y ),  
                  sister( Z, Y ).
- 1.4.      grandchild( X, Z ) :-  
                  parent( Y, X ),  
                  parent( Z, Y ).
- 1.5.      aunt( X, Y ) :-  
                  parent( Z, Y ),  
                  sister( X, Z ).
- 1.6.      Да, это — правильное определение.
- 1.7.      а)    Возврат не происходит.  
            б)    Возврат не происходит.  
            в)    Возврат не происходит.  
            г)    Происходит возврат.

## Глава 2

- 2.1.      а)    Переменная.  
            б)    Атом.  
            в)    Атом.  
            г)    Переменная.  
            д)    Атом.  
            е)    Структура.  
            ж)    Число.  
            з)    Синтаксически неправильное выражение.  
            и)    Структура.

- к) Синтаксически неправильное выражение.
- 2.3. а) `A = 1, B = 2`  
 б) `no`  
 в) `no`  
 г) `D = 2, E = 2`  
 д) `P1 = point(-1,0)`  
`P2 = point(1,0)`  
`P3 = point(0,Y)`
- Такая конкретизация определяет семейство треугольников, у которых две вершины находятся в точках 1 и -1 на оси x, а третья — в произвольной точке на оси y.
- 2.4. `seg( point(5,Y1), point(5,Y2))`
- 2.5. `regular( rectangle( point(X1,Y1), point(X2,Y1),`  
`point(X2,Y3), point(X1,Y3))).`  
 % Здесь предполагается, что первая точка представляет собой  
 % левую нижнюю вершину прямоугольника.
- 2.6. а) `A = two`  
 б) `no`  
 в) `C = one`  
 г) `D = s(s(1));`  
`D = s(s(s(s(s(1)))))`
- 2.7. `relatives( X, Y ) :-`  
`predecessor( X, Y)`  
`;`  
`predecessor( Y, X)`  
`;`  
`predecessor( Z, X),`  
`predecessor( Z, Y)`  
`;`  
`predecessor( X, Z),`  
`predecessor( Y, Z).`
- 2.8. `translate( 1, one).`  
`translate( 2, two).`  
`translate( 3, three).`
- 2.9. В случае, показанном в листинге 2.1, система Prolog выполняет немного больше работы.
- 2.10. В соответствии с определением операции согласования, приведенным в разделе 2.2, данное согласование будет выполнено успешно. Параметр X принимает вид своего рода циклической структуры, одним из компонентов которой становится сам этот параметр X.

## Глава 3

- 3.1. а) `conc( L1, [_,_,_], L)`  
 б) `conc( [_,_,_ | L2], [_,_,_], L)`
- 3.2. а) `last( Item, List) :-`  
`conc( _, [Item], List).`  
 б) `last( Item, [Item]).`  
`last( Item, [First | Rest]) :-`  
`last( Item, Rest).`
- 3.3. `evenlength([]).`  
`Evenlength( [First | Rest]) :-`  
`oddlength( Rest).`



- ```

oddlength( [First | Rest]) :-
    evenlength( Rest).
3.4.      reverse( [],[]).
          reverse( [First | Rest], Reversed) :-
              reverse( Rest, ReversedRest),
              conc( ReversedRest, [First], Reversed).
3.5.      % Эту задачу можно легко решить с помощью предиката reverse
          palindrome( List) :-
              reverse( List, List).
          % Еще одно решение, в котором не используется
          % предикат reverse
          palindromel( []).
          palindromel( [_]).
          palindromel( List) :-
              conc( [First | Middle], [First], List),
              palindromel( Middle).
3.6.      shift( [First | Rest], Shifted) :-
              conc( Rest, [First], Shifted).
3.7.      translate( [],[]).
          translate( [Head | Tail], [Head1 | Tail1]) :-
              means( Head, Head1),
              translate( Tail, Tail1).
3.8.      % В приведенном ниже решении предполагается, что порядок
          % элементов в списке Subset является таким же, как и в Set
          subset( [],[]).
          % Оставить элемент First в списке Subset
          subset( [First | Rest], [First | Sub]) :-
              subset( Rest, Sub).
          subset( [First | Rest], Sub) :- % Удалить элемент First
              subset( Rest, Sub).
3.9.      dividelist( [],[],[]). % Разделение пустого списка приводит
          % к получению двух пустых списков
          dividelist( [X], [X], []). % Разделение
          % одноэлементного списка
          dividelist( [X, Y | List], [X | List1], [Y | List2]) :-
              dividelist( List, List1, List2).
3.10.     canget( state( _,_,_,has), []). % Возможные действия
          % отсутствуют
          canget( State, [Action | Actions]) :-
              move( State, Action, NewState), % Первое действие
              canget( NewState, Actions). % Оставшиеся действия
3.11.     flatten( [Head | Tail], FlatList) :- % Линеаризация
          % непустого списка
              flatten( Head, FlatHead),
              flatten( Tail, FlatTail),
              conc( FlatHead, FlatTail, FlatList).
          flatten( [],[]). % Линеаризация пустого списка
          flatten( X, [ X]).
          % Примечание. При использовании перебора с возвратами эта
          % программа вырабатывает бессмысленные результаты
3.12.     Term1 = plays( jimmy, and( football, squash))
          Term2 = plays( susan, and( tennis,
              and( basketball, volleyball)))
3.13.     :- op( 300, xfx, was).
          :- op( 200, xfx, of).
          :- op( 100, fx, the).
3.14.     a)   A = 1 + 0

```

```

б)   B = 1 + 1 + 0
в)   C = 1 + 1 + 1 + 1 + 0
г)   D = 1 + 1 + 0 + 1;
      D = 1 + 0 + 1 + 1;
      D = 0 + 1 + 1 + 1 % Дальнейшие попытки выполнить перебор
                           % с возвратами вызывают возникновение бесконечного цикла

3.15. :- op( 100, xfx, in).
      :- op( 300, fx, concatenating).
      :- op( 200, xfx, gives).
      :- op( 100, xfx, and).
      :- op( 300, fx, deleting).
      :- op( 100, xfx, from).
      % Принадлежность к списку
      Item in [Item | List].
      Item in [First | Rest] :-
          Item in Rest.
      % Конкатенация списков
      concatenating [] and List gives List.
      Concatenating [X | L1] and L2 gives [X | L3] :-
          concatenating L1 and L2 gives L3.
      % Удаление элемента из списка
      deleting Item from [Item | Rest] gives Rest.
      deleting Item from [First | Rest] gives [First | NewRest] :-
          deleting Item from Rest gives NewRest.

3.16. max( X, Y, X) :-
      X >= Y.
      max( X, Y, Y) :-
      X < Y.

3.17. maxlist( [ X], X). % Наибольший элемент
      % в одноэлементном списке
      maxlist( [X, Y | Rest], Max) :- % По меньшей мере
      % два элемента в списке
          maxlist( [Y | Rest], MaxRest),
          max( X, MaxRest, Max). % Элемент Max - наибольший среди X
      % и MaxRest

3.18. sumlist([], 0).
      sumlist( [First | Rest], Sum) :-
          sumlist( Rest, SumRest),
          Sum is First + SumRest.

3.19. ordered( [X]). % Одноэлементный список
      % является упорядоченным
      ordered( [X, Y | Rest]) :-
          X=<Y,
          ordered( [Y | Rest]).

3.20. subsum([], 0, []).
      subsum( [N | List], Sum, [N | Sub]) :- % Элемент N
      % принадлежит к подмножеству
          Sum1 is Sum - N,
          subsum( List, Sum1, Sub).
      subsum( [N | List], Sum, Sub) :- % Элемент N не
      % принадлежит к подмножеству
          subsum( List, Sum, Sub).

3.21. between( N1, N2, N1) :-
      N1 =<= N2.
      between( N1, N2, X) :-
          N1 < N2,
          NewN1 is N1 + 1,
          between( NewN1, N2, X).

```

```

3.22.      :- op( 900, fx, if).
            :- op( 800, xfx, then).
            :- op( 700, xfx, else).
            :- op( 600, xfx, :=).
            if Val1 > Val2 then Var := Val3 else Anything :-
                Val1 > Val2,
                Var = Val3.
            if Val1 > Val2 then Anything else Var := Val4 :-
                Val1 = <Val2,
                Var = Val4.

```

Глава 4

- 4.1. a) `?- family( person( _,Name,_,_), _, []).`
 б) `?- child( person( Name, SecondName, _, works( _,_))).`
 в) `?- family(person(_,Name,_,unemployed),
 person(_,_,_,works(_,_)), _).`
 г) `?- family(Husband, Wife, Children),
 dateofbirth(Husband, date(_,_, Year1)),
 dateofbirth(Wife, date(_,_,Year2)),
 (Year1 - Year2 >= 15
 ;
 Year2 - Year1 >= 15
),
 member(Child, Children).`
- 4.2. `twins(Child1, Child2) :-
 family(_,_, Children),
 del(Child1, Children, OtherChildren), % Удалить
 % объект Child1
 member(Child2, OtherChildren),
 dateofbirth(Child1, Date),
 dateofbirth(Child2, Date).`
- 4.3. `nth_member(1, [X | L], X). % X - первый элемент
 % списка [X | L]
 nth_member(N, [Y | L], X) :- % X - n-й элемент
 % списка is [Y | L]
 N1 is N - 1,
 nth_member(N1, L, X).`
- 4.4. Входная строка сокращается после прохождения каждого цикла, отличного от скрытого, а поскольку она имеет конечную длину, то не может сокращаться бесконечно долго.
- 4.5. `accepts(State, [], _) :-
 final(State).
 accepts(State, [X | Rest], MaxMoves) :-
 MaxMoves > 0,
 trans(State, X, Statel),
 NewMax is MaxMoves - 1,
 accepts(Statel, Rest, NewMax).
 Accepts(State, String, MaxMoves) :-
 MaxMoves > 0,
 silent(State, Statel),
 NewMax is MaxMoves - 1,
 accepts(Statel, String, NewMax).`
- 4.6. Такой порядок определен в цели `member( Y, [1,2,3,4,5,6,7,8]).`
- 4.7. а) `jump(X/Y, X1/Y1) :- % Конь переходит из клетки
 % X/Y в клетку X1/Y1
 (
 dxу(Dx, Dy) % Расстояния между позициями коня`

- ```

 % в направлениях x и y
 ;
 dxy(Dy, Dx) % или в направлениях y и x
),
 X1 is X + Dx,
 inboard(X1), % X1 находится в пределах шахматной доски
 Y1 is Y + Dy,
 inboard(Y1). % Y1 находится в пределах шахматной доски
 dxy(2, 1). % 2 клетки вправо, 1 вперед
 dxy(2, -1). % 2 клетки вправо, 1 назад
 dxy(-2, 1). % 2 клетки влево, 1 вперед
 dxy(-2, -1). % 2 клетки влево, 1 назад
 inboard(Coord) :- % Координаты находятся в пределах
 % шахматной доски
 0 < Coord,
 Coord < 9.

```
- б) knightpath( [Square]). % Конь стоит на клетке Square  
 knightpath( [S1,S2 | Rest]) :-  
     jump( S1, S2),  
     knightpath( [S2 | Rest]).
- в) ?- knightpath( [2/1,R,5/4,S,X/8]).

## Глава 5

- 5.1. а) X = 1;  
       X = 2;
- б) X = 1;  
    Y = 1;  
    X = 1;  
    Y = 2;  
    X = 2;  
    Y = 1;  
    X = 2;  
    Y = 2;
- в) X = 1;  
    Y = 1;  
    X = 1;  
    Y = 2;
- 5.2. Предполагается, что вызов процедуры `class` происходит с неконкретизированным вторым параметром.
- ```

class( Number, positive) :-
    Number > 0, !.
class( 0, zero) :- !.
class( Number, negative).

```
- 5.3. `split( [],[],[]).`
`split( [X | L], [X | L1], L2) :-`
 X >= 0, !,
 split(L, L1, L2).
`split( [X | L], L1, [X | L2]) :-`
 split(L, L1, L2).
- 5.4. `member( Item, Candidates), not member( Item, RuledOut)`
- 5.5. `set_difference( [],_, []).`
`set_difference( [X | L1], L2, L) :-`
 member(X, L2), !,
 set_difference(L1, L2, L).
`set_difference( [X | L1], L2, [X | L]) :-`
 set_difference(L1, L2, L).

```

5.6.      unifiable( [], _, []).
          unifiable( [First | Rest], Term, List) :-
              not( First = Term), !,
              unifiable( Rest, Term, List).
          unifiable( [First | Rest], Term, [First | List]) :-
              unifiable( Rest, Term, List).

```

Глава 6

```

6.1.      findterm( Term) :- % Предполагается, что текущим входным
                              % потоком является файл f
                              read( Term), !, % Текущий терм в файле f согласуется
                              % с переменной Term?
                              write( Term) % В случае положительного ответа вывести его
                              % на внешнее устройство
                              ;
                              findterm( Term). % В противном случае перейти к обработке
                              % остальной части файла

6.2.      findallterms( Term) :-
                              read( CurrentTerm), % Предполагается, что терм CurrentTerm
                                                      % не является переменной
                              process( CurrentTerm, Term).
          process( end_of_file, _) :- !.
          process( CurrentTerm, Term) :-
              ( not( CurrentTerm = Term), ! % Термы не согласуются
              ;
              write( CurrentTerm), nl % В противном случае вывести
              % текущий терм на внешнее устройство
              ),
              findallterms( Term). % Перейти к обработке остальной
              % части файла

6.4.      starts( Atom, Character) :-
              name( Character, [Code]),
              name( Atom, [Code | _]).

6.5.      plural( Noun, Nouns) :-
              name( Noun, CodeList),
              name( s, CodeS),
              conc( CodeList, CodeS, NewCodeList),
              name( Nouns, NewCodeList).

```

Глава 7

```

7.2.      add_to_tail( Item, List) :-
              var( List), !, % Переменная List представляет собой
                              % пустой список
              List = [Item | Tail].
          add_to_tail( Item, [_ | Tail]) :-
              add_to_tail( Item, Tail).
          member( X, List) :-
              var( List), !, % Переменная List представляет собой
                              % пустой список,
              fail. % поэтому X не может быть его элементом
          member( X, [X | Tail]).
          member( X, [_ | Tail]) :-
              member( X, Tail).

7.5.      % subsumes( Term1, Term2):
          % терм Term1 является более общим, чем Term2, например
          % subsumes( t(X,a,f(Y)), t(A,a,f(g(B))))
          % Предполагается, что Term1 и Term2 не содержат

```

```

% одинаковую переменную
% В следующей процедуре обобщаемые переменные
% конкретизируются значениями термов в форме
% literally( SubsumedTerm)
subsumes( Atom1, Atom2) :-
    atomic( Atom1), !,
    Atom1 == Atom2.
subsumes( Var, Term) :-
    var( Var), !, % Переменная обобщает любые другие термы
    Var = literally( Term). % Перейти к обработке остальных
                           % вхождений Var
subsumes( literally( Term1), Term2) :- !, % Еще одно
                                           % вхождение Term2
    Term1 == Term2.
subsumes( Term1, Term2) :- % Term1 - не переменная
    nonvar( Term2),
    Term1 =.. [Fun | Args1],
    Term2 =.. [Fun | Args2],
    subsumes_list( Args1, Args2).
subsumes_list( [], []).
subsumes_list( [First1 | Rest1], [First2 | Rest2]) :-
    subsumes( First1, First2),
    subsumes_list( Rest1, Rest2).
7.6. a) ?- retract( product( X, Y, Z)), fail.
      б) ?- retract( product( X, Y, 0)), fail.
7.7. copy_term( Term, Copy) :-
      asserta( term_to_copy( Term)),
      retract( term_to_copy( Copy)).
7.9. copy_term( Term, Copy) :-
      bagof( X, X = Term, [Copy]).

```

Глава 8

```

8.2. add_at_end( L1 - [Item | Z2], Item, L1 - Z2).
8.3. reverse( A - Z, L - L) :- % Результатом становится
                                % пустой список,
    A == Z, !. % если A - Z представляет собой пустой список
reverse( [X | L] - Z, RL - RZ) :- % Непустой список
    reverse( L - Z, RL - [X | RZ]).
8.6. % Программа решения задачи с восемью ферзями
sol(Ylist) :-
    functor( Du, u, 15), % Множество восходящих диагоналей
    functor( Dv, v, 15), % Множество нисходящих диагоналей
    sol( Ylist,
    [1,2,3,4,5,6,7,8], % Множество координат X
    [1,2,3,4,5,6,7,8], % Множество координат Y
    Du, Dv).
sol( [], [], [], _, _).
sol( [Y | Ys], [X | XL], YL0, Du, Dv) :-
    del( Y, YL0, YL), % Выбрать координату Y
    U is X+Y-1,
    arg( U, Du, X), % Восходящая диагональ свободна
    V is X-Y+8,
    arg( V, Dv, X), % Нисходящая диагональ свободна
    sol( Ys, XL, YL, Du, Dv).
del( X, [X | L], L).
del( X, [Y | L0], [Y | L]) :-
    del( X, L0, L).

```

Глава 9

- 9.4. `% mergesort(List, SortedList): применение алгоритма
% сортировки-слияния
mergesort([], []).
mergesort([X], [X]).
mergesort(List, SortedList) :-
 divide(List, List1, List2), % Разделить на списки
 % примерно равной длины
 mergesort(List1, Sorted1),
 mergesort(List2, Sorted2),
 merge(Sorted1, Sorted2, SortedList). % Выполнить слияние
 % отсортированных списков

divide([], [], []).
divide([X], [X], []).
divide([X, Y | L], [X | L1], [Y | L2]) :- % Поместить X, Y
 % в отдельные списки
 divide(L, L1, L2).
% merge(List1, List2, List3): см. раздел 8.3.1`
- 9.5. a) `binarytree(nil).
binarytree(t(Left, Root, Right)) :-
 binarytree(Left),
 binarytree(Right).`
- 9.6. `height(nil, 0).
height(t(Left, Root, Right), H) :-
 height(Left, LH),
 height(Right, RH),
 max(LH, RH, MH),
 H is 1 + MH.
max(A, B, A) :-
 A >= B, !.
max(A, B, B).`
- 9.7. `linearize(nil, []).
linearize(t(Left, Root, Right), List) :-
 linearize(Left, List1),
 linearize(Right, List2),
 conc(List1, [Root | List2], List).`
- 9.8. `maxelement(t(_, Root, nil), Root) :- !. % Root - это элемент,
 % крайний справа
maxelement(t(_, _, Right), Max) :- % Правое поддереву
 % не пусто
 maxelement(Right, Max).`
- 9.9. `in(Item, t(_, Item, _), [Item]).
in(Item, t(Left, Root, _), [Root | Path]) :-
 gt(Root, Item),
 in(Item, Left, Path).
in(Item, t(_, Root, Right), [Root | Path]) :-
 gt(Item, Root),
 in(Item, Right, Path).`
- 9.10. `% Процедура отображения схемы бинарного дерева, в которой
% корень находится вверху, а листья - внизу
% В этой программе предполагается, что имя каждого узла
% состоит только из одного символа
show(Tree) :-
 dolevels(Tree, 0, more). % Пройти все уровни, начиная
 % от корня
 dolevels(Tree, Level, alldone) :- !. % Ниже уровня Level
 % узлов больше нет
 dolevels(Tree, Level, more) :- % Пройти все уровни,
 % начиная от уровня Level`

```

    traverse( Tree, Level, 0, Continue), nl, % Вывести на
        % внешнее устройство информацию об
        % узлах, находящихся на уровне Level
    NextLevel is Level + 1,
    dolevels( Tree, NextLevel, Continue). % Обработка более
        % низких уровней
traverse( nil, _, _).
traverse( t( Left, X, _Right), Level, Xdepth, Continue) :-
    NextDepth is Xdepth + 1,
    traverse( Left, Level, NextDepth, Continue), % Обработать
% левое поддерево
    ( Level = Xdepth, !, % Узел X находится на уровне Level?
    write( X), Continue = more % Вывести информацию об узле
% и отметить, что работа не окончена
    ;
    write(' ') % В противном случае оставить свободное место
    ),
    traverse( Right, Level, NextDepth, Continue). % Обработать
        % правое поддерево

```

Глава 10

10.1.

```

in( Item, l( Item)). % Элемент находится в листе
in( Item, n2( T1, M, T2)) :- % Узел имеет два поддерева
    gt( M, Item), !, % Элемент Item не находится во втором
        % поддереве
    in( Item, T1) % Выполнить поиск в первом поддереве
    ;
    in( Item, T2). % В противном случае выполнить поиск
        % во втором поддереве
in( Item, n3( T1, M2, T2, M3, T3)) :- % Узел имеет три
        % поддерева
    gt( M2, Item), !, % Элемент Item не находится во втором
        % или в третьем поддереве
    in( Item, T1) % Выполнить поиск в первом поддереве
    ;
    gt( M3, Item), !, % Элемент не находится в третьем
        % поддереве
    in( Item, T2) % Выполнить поиск во втором поддереве
    ;
    in( Item, T3). % Выполнить поиск в третьем поддереве

```

10.3.

```

avl(Tree) :-
    avl( Tree, Height). % Tree - это AVL-дерево
        % с высотой Height
avl( nil, 0). % Пустое дерево - это AVL-дерево, которое
        % имеет высоту 0
avl( t( Left, Root, Right), H) :-
    avl( Left, HL),
    avl( Right, HR),
    ( HL is HR; HL is HR + 1; HL is HR - 1), % Поддеревья
        % имеют примерно равную высоту
    max1( HL, HR, H).
max1( U, V, M) :- % M - это наибольшее значение среди U и V,
        % увеличенное на 1
    U > V, !, M is U + 1
    ;
    M is V + 1.

```

10.4.

Корневым элементом первоначально становится 5, затем 8 и, наконец, снова 5.

Глава 11

- 11.1.
- ```
depthfirst1([Node | Path], [Node | Path]) :-
 goal(Node).
depthfirst1([Node | Path], Solution) :-
 s(Node, Node1),
 not member(Node1, Path),
 depthfirst1([Node1, Node | Path], Solution).
```
- 11.3.
- % Процедура поиска с итеративным углублением, в которой  
% глубина перестает увеличиваться, если отсутствует путь  
% на текущую глубину
- ```
iterative_deepening( Start, Solution ) :-  
    id_path( Start, Node, [], Solution ),  
    goal( Node ).  
% path( First, Last, Path ): переменная Path представляет  
% собой список узлов от First до Last  
path( First, First, [First] ).  
path( First, Last, [First, Second | Rest] ) :-  
    s( First, Second ),  
    path( Second, Last, [Second | Rest] ).  
% Процедура формирования пути с итеративным углублением  
% id_path( First, Last, Template, Path ): переменная Path  
% представляет собой путь от узла First до узла Last, длина  
% которого не превышает длину применяемого в качестве  
% образца списка Template. Альтернативные пути  
% вырабатываются в порядке возрастания длины  
id_path( First, Last, Template, Path ) :-  
    Path = Template,  
    path( First, Last, Path )  
    ;  
    copy_term( Template, P ),  
    path( First, _, P ), !, % По меньшей мере один путь  
    % с длиной, равной длине списка Template  
    id_path( First, Last, [_ | Template], Path ). % Путь  
    % с длиной, превышающей длину списка Template
```
- 11.6.
- Поиск в ширину — 15 узлов; итеративное углубление — 26 узлов.
- ```
N(b, 0) = 1
N(b, d) = N(b, d - 1) + (bd+1 - 1) / (b - 1) при d > 0
```
- 11.8.
- ```
solve( StartSet, Solution ) :- % StartSet - это список  
    % начальных узлов  
    bagof( [Node], member( Node, StartSet ), CandidatePaths ),  
    breadthfirst( CandidatePaths, Solution ).
```
- 11.9.
- Поиск в обратном направлении является более выгодным, если коэффициент ветвления в обратном направлении меньше, чем в прямом. Поиск в обратном направлении применим, только если целевой узел явно определен.
- % Предположим, что origs(Node1, Node2) - первоначальное
% отношение с определением пространства состояний
% Определите новое отношение s следующим образом:
- ```
s(Node1, Node2) :-
 origs(Node2, Node1).
```
- 11.10.
- % Состояниями для двунаправленного поиска являются пары  
% узлов StartNode-EndNode из первоначального пространства  
% состояний, обозначающие начало и цель поиска
- ```
s( Start - End, NewStart - NewEnd ) :-  
    origs( Start, NewStart ), % Шаг в прямом направлении  
    origs( NewEnd, End ). % Шаг в обратном направлении  
    % goal( Start - End ) для двунаправленного поиска  
goal( Start - Start ). % Начало совпадает с концом  
goal( Start - End ) :-  
    origs( Start, End ). % До конца остался один шаг
```

- 11.11. `find1` — поиск в глубину; `find2` — итеративное углубление (предикат `conc(Path, _, _)` формирует шаблоны списков все возрастающей длины, которые вынуждают предикат `find1` перейти в режим итеративного углубления); `find3` — поиск в обратном направлении.
- 11.12. Двухнаправленный поиск с итеративным углублением с обоих концов.

Глава 12

- 12.2. Это — неправильное утверждение; $h < h^*$ является достаточным, но не необходимым условием определения допустимости.

- 12.3. $h(n) = \max\{h_1(n), h_2(n), h_3(n)\}$

- 12.7.

```
% Спецификация головоломки "игра в восемь"
% для алгоритма IDA*
s( Depth:State, NewDepth:NewState) :-
    s( State, NewState, _),
    NewDepth is Depth + 1.
f( Depth:[Empty | Tiles], F) :-
    goal( [Empty0 | Tiles0]),
    totdist( Tiles, Tiles0, Dist),
    F is Depth + Dist. % В качестве эвристической функции
                    % используется суммарное расстояние
% Начальные состояния для алгоритма IDA* задаются
% в форме 0:State, где State - начальное состояние
% для алгоритма A* (см. листинг 12.2)
showpos( Depth:State) :-
    showpos( State).
```

Если задано начальное состояние $[1/2, 3/3, 3/1, 1/3, 3/2, 1/1, 2/3, 2/1, 2/2]$, то алгоритм A^* (в котором используется эвристическая функция, не требующая обеспечения допустимости) позволяет быстро найти решение с длиной 38, а алгоритм IDA* (вызванный на выполнение с использованием в качестве эвристической функции суммарного расстояния) требует больше времени, но находит оптимальное решение с длиной 24.

- 12.9. Порядок формирования узлов: a, b, c, f, g, j, k, d, e, f, g, j, l, k, m, j, l.
Последовательность обновлений $F(b)$ и $F(c)$: $F(b) = 2, F(c) = 1, F(c) = 3, F(b) = 5$.

Глава 13

- 13.4. Формируются три дерева решения, которые имеют стоимости 8, 10 и 11. Интервалы значений таковы: $0 \leq h(c) < 9, 0 \leq h(f) < 5$.

Глава 14

- 14.1. Будет получен такой же результат, независимо от выбранной последовательности.
- 14.6. Диод, подключенный последовательно с резистором R_5 , в направлении слева направо, не должен повлиять на напряжение в клемме T51. Диод, подключенный в противоположном направлении, влияет на напряжение этой клеммы.

Глава 15

- 15.2. Ошибка не возникает, если $p(A|B) = 1$ или $p(B|A) = 1$. Ошибка становится наибольшей (0,5), если $p(A) = p(B) = 0.5$ и $p(A|B) = 0$.
- 15.4. $A = 6, B = 5, C = 25$.

Глава 17

17.6.

Нет. Причина этого состоит в том, что одно действие позволяет достичь больше одной цели одновременно.

Глава 18

18.1.

```
Для всего множества примеров: I = 2.2925
Ires(size) = 1.5422, Gain(size) = 0.7503
Info(size) = 0.9799, GainRatio(size) = 0.7503/0.9799
        = 0.7657
Ires(holes) = 0.9675, Gain(holes) = 1.324
Info(holes) = 1.5546, GainRatio(holes) = 0.8517
```

18.2.

```
Gain = I - Ires
I = - (p(D) log p(D) + p(~D) log p(~D))
  = - ( 0.25 log 0.25 + 0.75 log 0.75) = 0.8113
Для вычисления Ires требуется p(S), p(~S), p(D|S), p(D|~S)
p(S) = p(S|D) p(D) + p(S|~D) p(~D) = 0.75 * 0.25
      + 1/6 * 0.75 = 0.3125
С использованием формулы Байеса:
p(D|S) = p(D) p(S|D) / p(S) = 0.25 * 0.75 / 0.3125 = 0.6
p(~D|S) = 0.4
p(D|~S) = p(D)*p(~S|D) / p(~S) = 0.25 * 0.25 / (1-0.3125)
        = 0.09090
Ires = p(S)*I(D|S) + p(~S)*I(D|~S) = 0.6056
В приведенной выше формуле I(D|S) обозначает количество
информации о болезни, полученное с учетом наличия
указанного симптома.
Gain = 0.2057
GainRatio = Gain(S) / I(S) = 0.2057/0.8960 == 0.2296
```

18.5.

```
% prunedtree( Tree, PrunedTree): переменная PrunedTree
% представляет собой дерево Tree, отсечение поддеревьев
% которого выполнено оптимальным образом по отношению
% к оценочному значению ошибки классификации
% Предполагается, что деревья являются бинарными:
% Tree = leaf( Node, ClassFrequencyList) или
% Tree = tree( Root, LeftSubtree, RightSubtree)
prunedtree( Tree, PrunedTree) :-
    prune( Tree, PrunedTree, Error, FrequencyList).
% prune( Tree, PrunedTree, Error, FrequencyList):
%   переменная PrunedTree представляет собой дерево Tree,
%   отсечение поддеревьев которого выполнено оптимальным
%   образом по отношению к оценочному значению ошибки
%   классификации, а FrequencyList - это список частот
%   классов в корне дерева Tree
prune( leaf( Node, FreqList),
leaf( Node, FreqList), Error, FreqList) :-
    static_error( FreqList, Error).
prune( tree( Root, Left, Right),
PrunedT, Error, FreqList) :-
    prune( Left, Left1, LeftError, LeftFreq),
    prune( Right, Right1, RightError, RightFreq),
    sumlists( LeftFreq, RightFreq, FreqList), % Добавить
                                           % соответствующие элементы
    static_error( FreqList, StaticErr),
    sum( LeftFreq, N1),
    sum( RightFreq, N2),
    BackedErr is ( N1 * LeftError + N2 * RightError)
                / ( N1 + N2),
    decide( StaticErr, BackedErr, Root, FreqList, Left1,
            Right1, Error, PrunedT).
% Предикат, вырабатывающий решение о том, должно ли быть
```

```

% выполнено отсечение или нет:
decide( StatErr, BackErr, Root, FreqL, _, _, StatErr,
leaf( Root, FreqL)) :-
    StatErr = < BackErr,!. % Статическая ошибка меньше:
                        % выполнить отсечение поддеревьев
% В противном случае не выполнять отсечение:
decide( _, BackErr, Root, _, Left, Right, BackErr,
tree( Root, Left, Right)).
% static_error( ClassFrequencyList, Error): оценочное
% значение ошибки классификации
static_error( FreqList, Error) :-
    max( FreqList, Max), % Максимальное число
                        % в списке FreqList
    sum( FreqList, All), % Сумма чисел в списке FreqList
    number_of_classes( NumClasses),
    Error is ( All - Max + NumClasses - 1)
        / ( All + NumClasses).

sum([],0).
sum( [Number | Numbers], Sum) :-
    sum( Numbers, Sum1),
    Sum is Sum1 + Number.
max( [X], X).
max( [X,Y | List], Max) :-
    X > Y, !, max( [X | List], Max)
;
    max( [Y | List], Max).
sumlists([], [], []).
sumlists( [X1 | L1], [X2 | L2], [X3 | L3]) :-
    X3 is X1 + X2,
    sumlists( L1, L2, L3).
% Произвольное дерево
treel( tree( a, % Корень
    tree( b, leaf( e, [3,2]), leaf( f, [1,0])), % Левое
                        % поддерево
    tree( c, tree( d, leaf( g, [1,1]), leaf( h, [0,1])),
        leaf( i, [1,0])))).
number_of_classes( 2).
% Контрольный вопрос:
% ?- treel( Tree), prunetree( Tree, PrunedTree).

```

Глава 19

- 19.2. Семь этапов.
- 19.3. Количество выработанных гипотез равно 373179, а количество уточненных гипотез — 66518.
- 19.4. Гипотеза $\{C_0, C_1\}$ является более общей, чем $\{C_0, C_2\}$. Предложение C_1 не обеспечивает тэта-классификацию предложения C_2 .

Глава 20

- 20.1. `qmult( pos, pos, pos).`
`qmult( pos, zero, zero).`
`qmult( pos, neg, neg).`
`...`
- 20.2. `resistor( pos, pos).`
`resistor( zero, zero).`
`resistor( neg, neg).`
`diode( zero, pos).`
`diode( zero, zero).`
`diode( neg, zero).`
- 20.3. а) Первое состояние: $X = \text{zero/inc}$, $y = \text{zero/inc}$

Второе состояние: $X = \text{zero}..inf/inc$, $Y = \text{zero}..inf/inc$
 Третье состояние: $X = \text{zero}..inf/std$, $Y = \text{zero}..inf/inc$ или
 $X = \text{zero}..inf/std$, $Y = \text{zero}..inf/std$ или
 $X = \text{zero}..inf/inc$, $Y = \text{zero}..inf/std$

- б) Ответ совпадает с ответом к упражнению а), за исключением того, что третьим состоянием может быть только следующее:
 $X = \text{zero}..inf/std$, $Y = \text{zero}..inf/std$

20.4.

При $T=t0$: $X = \text{zero}/inc$, $Y = \text{zero}/inc$, $Z = \text{zero}/inc$
 При $T=t1$: $X = \text{zero}..inf/inc$, $Y = \text{zero}..inf/inc$,
 $Z = \text{zero}..landz/inc$
 При $T=t2$: $X = \text{zero}..inf/inc$, $Y = \text{zero}..inf/inc$,
 $Z = \text{landz}/inc$ или
 $X = \text{zero}..inf/std$, $Y = \text{zero}..inf/std$,
 $Z = \text{zero}..landz/std$ или
 $X = \text{zero}..inf/std$, $Y = \text{zero}..inf/std$,
 $Z = \text{landz}/std$

Примечание. Эти результаты могут быть также получены с помощью программы машинного моделирования, приведенной в листинге 20.2, и следующей модели, которая соответствует данному упражнению:

```
landmarks( x, [minf,zero,inf]).
landmarks( y, [minf,zero,inf]).
landmarks( z, [minf,zero,landz,inf]).
correspond( x:zero, y:zero).
legalstate( fX,Y,Z) :-
    mplus( X, Y),
    sum( X, Y, Z).
```

```
initial( [x:zero/inc, y:Y0, z:Z0]).
```

Необходимый для этого запрос приведен ниже.

```
?- initial(S), simulate(S,Beh,3).
```

20.5.

```
% Модель сообщающихся сосудов
% Здесь levA0 и levB0 - начальные уровни в контейнерах A
% и B, fAB0 - начальный поток из A в B, fBA0 начальный
% поток из B в A
landmarks( level, [ zero, levB0, levA0, inf]).
landmarks( levdiff, [minf, zero, inf]).
landmarks( flow, [minf, fBA0, zero, fAB0, inf]).
correspond( levdiff:zero, flow:zero).
correspond( flow:fAB0, flow:fBA0, flow:zero).
legalstate( [ LevA, LevB, FlowAB, FlowBA] ):-
    deriv( LevA, FlowBA),
    deriv( LevB, FlowAB),
    sum( FlowAB, FlowBA, flow:zero/std), % FlowBA = -FlowAB
    DiffAB = levdiff:_,
    sum( LevB, DiffAB, LevA), % DiffAB = LevA - LevB
    mplus( DiffAB, FlowAB).
initial( [ level:levA0/dec, level:levB0/inc,
          flow:fAB0/dec, flow:fBA0/inc]).
```

20.7.

```
legal_trans( State1, State2) :-
    system_trans( State1, State2),
    State1 \== State2, % Качественно иное следующее состояние
    not (point_state( State1), % Не допускается, чтобы
        % мгновенными были
        point_state( State2)), % и состояние State1,
        % и состояние State2
    legalstate( State2). % Допустимое в соответствии с моделью
point_state( State) :-
    member( _:Qmag/Dir, State),
    not (Qmag = _.._), % Qmag - это отметка, а не интервал
    Dir \== std. % Направление Dir отлично от устойчивого
```

Глава 21

21.1.

```
s( [a | List], Rest) :-  
    s( List, [b | Rest]).
```

21.3. Это модифицированное определение является менее эффективным и может привести к возникновению бесконечного цикла.

Глава 23

23.1. Введите в программу метаинтерпретатора следующее предложение:

```
prove( clause( Head, Body)) :-  
    clause( Head, Body).
```

23.3. Варианты возникают в связи с тем, что предикат `square(6)` наследует метод `perimeter` от нескольких объектов. Такое множественное наследование можно предотвратить, введя оператор отсечения в процедуру `send`

Список литературы

1. Advances in Computer Chess Series: Clarke M.R.B. (ed.). Vols 1-2. Edinburgh University Press; Clarke M.R.B. (ed.). Vol. 3. Pergamon Press; Beal D.F. (ed.). Vol. 4. Pergamon Press; Beal D.F. (ed.). Vol. 5. North-Holland; Beal D.F. (ed.). Vol. 6. Ellis Horwood.
2. Aho A.V., Hopcroft J.E., Ullman J.D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.
3. Aho A.V., Hopcroft J.E., Ullman J.D. (1983). *Data Structures and Algorithms*. Addison-Wesley.
4. AIJ volume 76 (1995). *Artificial Intelligence*, Vol. 76. Special issue on Planning and Scheduling.
5. Allen J.F. (1995). *Natural Language Understanding*. Redwood City, CA: Benjamin/Cummings.
6. Allen J., Hendler J., Tate A. (eds) (1990). *Readings in Planning*. San Mateo, CA: Morgan Kaufmann.
7. Berliner H.J. (1977). A representation and some mechanisms for a problem solving chess program. In: Clarke M.R.B. (ed.). *Advances in Computer Chess 1*. Edinburgh University Press.
8. Bobrow D.G. (ed.) (1984). *Artificial Intelligence Journal*, Vol. 24 (Special Volume on Qualitative Reasoning about Physical Systems). Этот сборник статей вышел также под названием: *Qualitative Reasoning about Physical Systems*. Cambridge, MA: MIT Press 1985.
9. Bowen D.L. (1981). *DECsystem-10 Prolog User's Manual*. University of Edinburgh: Department of Artificial Intelligence.
10. Brachman R.J., Levesque H.J. (eds) (1985). *Readings in Knowledge Representation*. Los Altos, CA: Morgan Kaufmann.
11. Bramer M.A. (ed.) (1983). *Computer Game Playing: Theory and Practice*. Chichester: Ellis Horwood and John Wiley.
12. Bratko I. (1990). *Prolog Programming for Artificial Intelligence*. Addison-Wesley.
13. Bratko I. (1978). Proving correctness of strategies in the AL1 assertional language. *Information Processing Letters*, 7: 223-230.
14. Bratko I. (1982). Knowledge-based problem solving in AL3. In: Hayes J., Michie D., Pao J.H. (eds). *Machine Intelligence 10*. Ellis Horwood (сокращенную версию можно также найти в [11]).
15. Bratko I. (1982). Knowledge-based problem-solving in AL3. In: Hayes J.E., Michie D., Pao Y.H. (eds). *Machine Intelligence 10*. Ellis Horwood.
16. Bratko I. (1984). Advice and planning in chess end-games. In: Amarel S., Elithorn A., Banerji R. (eds). *Artificial and Human Intelligence*. North-Holland.
17. Bratko I. (1985). Symbolic derivation of chess patterns. In: Steels L., Campbell J.A. (eds). *Progress in Artificial Intelligence*. Chichester: Ellis Horwood and John Wiley.
18. Bratko I. (1999). Refining complete hypotheses in ILP. In: Džeroski S., Flach P. (eds). *Inductive Logic Programming. Proc. ILP-99*. LNAI 1634, Springer.

19. Bratko I., Michie D. (1980). An advice program for a complex chess programming task. *Computer Journal*, 23: 353–359.
20. Bratko I., Možetič, I., Lavrač, N. (1989). *KARDIO: a Study in Deep and Qualitative Knowledge for Expert Systems*. Cambridge, MA: MIT Press.
21. Bratko I., Muggleton S., Karalič A. (1998). Applications of inductive logic programming. In: Michalski R.S., Bratko I., Kubat M. (eds). *Machine Learning and Data Mining: Methods and Applications*. Chichester: Wiley.
22. Bratko I., Muggleton S., Varsek A. (1991). Learning qualitative models of dynamic systems. In: Brazdil P. (ed.). *Proc. Inductive Logic Programming ILP-91*, Viana do Castelo, Portugal. См. также: Muggleton S. (ed.). *Inductive Logic Programming*. London: Academic Press 1992.
23. Breiman L., Friedman J.H., Olshen R.A., Stone C.J. (1984). *Classification and Regression Trees*. Belmont, CA: Wadsworth Int. Group.
24. Buchanan B.C., Shortliffe E.H. (eds) (1984). *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley.
25. Castillo E., Gutierrez J.M., Hadi A.S. (1996). *Expert Systems and Probabilistic Network Models*. Berlin: Springer-Verlag.
26. Cestnik B. (1990). Estimating probabilities: a crucial task in machine learning. *Proc. ECAI 90*, Stockholm.
27. Cestnik B., Bratko I. (1991). On estimating probabilities in tree pruning. *Proc. European Conf. on Machine Learning*, Porto, Portugal. Berlin: Springer-Verlag.
28. Cestnik B., Kononenko I., Bratko I. (1987). ASISTANT 86: a knowledge elicitation tool for sophisticated users. In: Bratko I., Lavrač N. (eds). *Progress in Machine Learning*. Wilmslow, England: Sigma Press; распространяется издательством Wiley.
29. Chapman D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32: 333–377.
30. Clark P. (1985). *Towards an Improved Domain Representation for Planning*. Edinburgh University: Department of Artificial Intelligence, MSc Thesis.
31. Clark P., Niblett T. (1989). The CN2 induction algorithm. *Machine Learning*, 3, 262–284.
32. Clocksin W.F., Mellish C.S. (1987). *Programming in Prolog, second edition*. Berlin: Springer-Verlag.
33. Coelho H., Cotta J.C. (1988). *Prolog by Example*. Berlin: Springer-Verlag.
34. Coffman E.G., Denning P.J. (1973). *Operating Systems Theory*. Prentice Hall.
35. Cohen J. (1990). Constraint logic programming languages. *Communications of the ACM*, 33: 52–68.
36. Coiera E. (1989). Generating qualitative models from example behaviours. *DCS Report No. 8901*, School of Computer Sc. and Eng., Univ. of New South Wales, Sydney, Australia.
37. Cormen T.H., Leiserson C.E., Rivest R.L. (1990). *Introduction to Algorithms (second edition 2000)*. MIT Press.
38. Covington M.A. (1994). *Natural Language Processing for Prolog Programmers*. Englewood Cliffs, NJ: Prentice Hall.
39. Davis E. (1990). *Representations of Commonsense Knowledge*. San Mateo, CA: Morgan Kaufmann.
40. de Kleer J., Brown J.S. (1984). Qualitative physics based on confluences. *Artificial Intelligence Journal*, 24: 7–83.

41. de Kleer J., Williams B.C. (eds) (1991). *Artificial Intelligence Journal*, Vol. 51 (Special Issue on Qualitative Reasoning about Physical Systems II).
42. De Raedt L. (ed.) (1996). *Advances in Inductive Logic Programming*. Amsterdam: IOS Press.
43. Deransart P., Ed-Bdali A., Ceroni L. (1996). *Prolog: The Standard*. Berlin: Springer-Verlag.
44. Doran J., Michie D. (1966). Experiments with the graph traverser program. *Proc. Royal Society of London*, 294(A): 235–259.
45. Englemore R., Morgan T. (eds) (1988). *Blackboard Systems*. Reading, MA: Addison-Wesley.
46. Ernst G.W., Newell A. (1969). *GPS: A Case Study in Generality and Problem Solving*. New York: Academic Press.
47. Esposito F., Malerba D., Semeraro G. (1997). A comparative analysis of methods for pruning decision trees. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 19: 416–491.
48. Fallings B., Struss P. (eds) (1992). *Recent Advances in Qualitative Physics*. Cambridge, MA: MIT Press.
49. Fikes R.E., Nilsson N.J. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2: 189–208.
50. Fikes R.E., Nilsson N.J. (1993). STRIPS, a retrospective. *Artificial Intelligence*, 59: 227–232.
51. Flach P. (1994). *Simply Logical: Intelligent Reasoning by Example*. Chichester, UK: Wiley.
52. Forbus K., Falkenhainer B.C. (1992). Self-explanatory simulations: integrating qualitative and quantitative knowledge. См. [48].
53. Forbus K.D. (1984). Qualitative process theory. *Artificial Intelligence*, 24: 85–168.
54. Frey P.W. (ed.) (1983). *Chess Skill in Man and Machine (second edition)*. Berlin: Springer-Verlag.
55. Garey M.R., Johnson D.S. (1979). *Computers and Intractability*. W.H. Freeman.
56. Gaschnig J. (1979). *Performance measurement and analysis of certain search algorithms*. Carnegie-Mellon University: Computer Science Department. Technical Report CMU-CS-79-124 (PhD Thesis).
57. Gazdar G., Mellish C. (1989). *Natural Language Processing in Prolog*. Harlow: Addison-Wesley.
58. Genesereth M.R., Nilsson N.J. (1987). *Logical Foundation of Artificial Intelligence*. Palo Alto, CA: Morgan Kaufmann.
59. Gillies D. (1996). *Artificial Intelligence and Scientific Method*. Oxford University Press.
60. Ginsberg M. (1993). *Essentials of Artificial Intelligence*. San Francisco, CA: Morgan Kaufmann.
61. Gonnet G.H., Baeza-Yates R. (1991). *Handbook of Algorithms and Data Structures in Pascal and C (second edition)*. Addison-Wesley.
62. Hammond P. (1984). Micro-PROLOG for Expert Systems. In: Clark K.L., McCabe, F.G. (eds). *Micro-PROLOG: Programming in Logic*. Englewood Cliffs, NJ: Prentice Hall.
63. Hart P.E., Nilsson N.J., Raphael B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Sciences and Cybernetics*, SSC-4(2): 100–107.

64. Hau D.T., Coiera E.W. (1997). Learning qualitative models of dynamic systems. *Machine Learning Journal*, 26: 177–211.
65. Hsu F.-H., Anantharaman T.S., Campbell M.S., Nowatzky A. (1990). A grandmaster chess machine. *Scientific American*, 263: 44–50.
66. Jackson P. (1999). *Introduction to Expert Systems, third edition*. Harlow: Addison-Wesley.
67. Jaffar J., Maher M. (1994). Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20: 503–581.
68. Jensen F.V. (1996). *An Introduction to Bayesian Networks*. Berlin: Springer-Verlag.
69. Kaindl H. (1990). Tree searching algorithms. См. [95].
70. Kanal L., Kumar V. (eds) (1988). *Search in Artificial Intelligence*. Springer-Verlag.
71. Kearns M.J., Vazirani U.V. (1994). *An Introduction to Computational Learning Theory*. Cambridge, MA: MIT Press.
72. Kedar-Cabelli S.T., McCarty, L.T. (1987). Explanation-based generalization as resolution theorem proving. In: *Proc. 4th Int. Machine Learning Workshop*, Irvine, CA: Morgan Kaufmann.
73. Kingston J.H. (1998). *Algorithms and Data Structures (second edition)*. Addison-Wesley.
74. Knuth D.E., Moore R.W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, 6: 293–326.
75. Kodratoff Y., Michalski R.S. (1990). *Machine Learning: An Artificial Intelligence Approach, Vol. III*. Morgan Kaufmann.
76. Kolodner J.L. (1993). *Case-Based Reasoning*. San Francisco, CA: Morgan Kaufmann.
77. Kononenko I., Lavrač N. (1988). *Prolog through Examples: A Practical Programming Guide*. Wilmslow, UK: Sigma Press.
78. Korf R.E. (1985). Depth-first iterative deepening: an optimal admissible tree search. *Artificial Intelligence*, 27: 97–109.
79. Korf R.E. (1993). Linear-space best-first search. *Artificial Intelligence*, 62: 41–78.
80. Kowalski R. (1979). *Logic for Problem Solving*. North-Holland.
81. Kowalski R. (1980). *Logic for Problem Solving*. North-Holland.
82. Kowalski R., Sergot M. (1986). A logic-based calculus of events. *New Generation Computing*, 4: 67–95.
83. Kraan I.E., Richards B.L., Kuipers B.J. (1991). Automatic abduction of qualitative models. *Proc. 15th Int. Workshop on Qualitative Reasoning about Physical Systems*.
84. Kuipers B.J. (1986). Qualitative simulation. *Artificial Intelligence Journal*, 29, 289–338 (См. также: [167]).
85. Kuipers B.J. (1994). *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*. Cambridge, MA: MIT Press.
86. Lavrač N., Džeroski S. (1994). *Inductive Logic Programming: Techniques and Applications*. Chichester: Ellis Horwood.
87. Le T.V. (1993). *Techniques of Prolog Programming*. John Wiley & Sons.
88. Lenat D.B. (1982). AM: discovery in mathematics as heuristic search. In: Davis R., Lenat D.B. (eds). *Knowledge-Based Systems in Artificial Intelligence*. McGraw-Hill.
89. Lloyd J.W. (1991). *Foundations of Logic Programming, second edition*. Berlin: Springer-Verlag.

90. Luger G.F., Stubblefield W.A. (1998). *Artificial Intelligence, third edition*. Harlow: Addison-Wesley.
91. Mackworth A.K. (1992). Constraint satisfaction. In: Shapiro S.C. (ed.) *Encyclopedia of Artificial Intelligence, second edition*. New York: Wiley.
92. Makarović A. (1991). *Parsimony in Model-Based Reasoning*. Enschede: Twente University, PhD Thesis, ISBN 90-9004255-5.
93. Markus C. (1986). *Prolog Programming*. Addison-Wesley.
94. Marriott K., Stuckey P.J. (1998). *Programming with Constraints: an Introduction*. Cambridge, MA: The MIT Press.
95. Marsland A.T., Schaeffer J. (eds) (1990). *Computers, Chess and Cognition*. Berlin: Springer-Verlag.
96. Michalski R.S. (1983). A theory and methodology of inductive learning. In: Michalski R.S., Carbonell J.G., Mitchell T.M. (eds). *Machine Learning: An Artificial Intelligence Approach*. Tioga Publishing Company.
97. Michalski R.S., Bratko I., Kubat M. (eds) (1998). *Machine Learning and Data Mining: Methods and Applications*. Wiley.
98. Michalski R.S., Carbonell J.G., Mitchell T.M. (eds) (1983). *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA: Tioga Publishing Company.
99. Michalski R.S., Carbonell J.G., Mitchell T.M. (eds) (1986). *Machine Learning: An Artificial Intelligence Approach, Volume II*. Los Altos, CA: Morgan Kaufmann.
100. Michie D. (1986). The superarticulacy phenomenon in the context of software manufacture. In: *Proc. of the Royal Society, London, A405:189-212*. См. также: Michie D., Bratko L. *Expert Systems: Automating Knowledge Acquisition*. Harlow, England: Addison-Wesley.
101. Michie D. (ed.) (1979). *Expert Systems in the Microelectronic Age*. Edinburgh University Press.
102. Michie D., Ross R. (1970). Experiments with the adaptive graph traverser. *Machine Intelligence*, 5: 301–308.
103. Minsky M. (1975). A framework for representing knowledge. In: Winston P. (ed.). *The Psychology of Computer Vision*. McGraw-Hill.
104. Mitchell T.M. (1982). Generalization as search. *Artificial Intelligence*, 18: 203–226.
105. Mitchell T.M. (1997). *Machine Learning*. McGraw-Hill.
106. Mitchell T.M., Keller R.M., Kedar-Cabelli S.T. (1986). Explanation-based generalisation: a unifying view. *Machine Learning*, 1: 47–80.
107. Moss C. (1994). *Prolog++: The Power of Object-Oriented and Logic Programming*. Harlow: Addison-Wesley.
108. Možetič I. (1987a). Learning of qualitative models. In: Bratko I., Lavrač N. (eds). *Progress in Machine Learning*. Wilmslow, UK: Sigma Press.
109. Možetič I. (1987b). The role of abstractions in learning qualitative models. *Proc. Fourth Int. Workshop on Machine Learning*, Irvine, CA: Morgan Kaufmann.
110. Muggleton S. (1991). Inductive logic programming. *New Generation Computing*, 8: 295–318.
111. Muggleton S. (1995). Inverse entailment and Progol. *New Generation Computing*, 13: 245–286.
112. Muggleton S. (ed.) (1992). *Inductive Logic Programming*. London: Academic Press.
113. Newell A., Shaw J.C., Simon H.A. (1960). Report on a general problem-solving program for a computer. *Information Processing: Proc. Int. Conf. on Information Processing*. Paris: UNESCO.

114. Niblett T., Bratko I. (1986). Learning decision rules in noisy domains. In: Bramer M.A. (ed.). *Research and Development in Expert Systems III*. Cambridge University Press.
115. Nilsson N.J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill.
116. Nilsson N.J. (1980). *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga; also Berlin: Springer-Verlag.
117. O'Keefe, R. A. (1990). *The Craft of Prolog*. Cambridge, MA: MIT Press.
118. Pearl J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley.
119. Pearl J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.
120. Pereira F.C.N., Shieber S.M. (1987). *Prolog and Natural-Language Analysis*. Menlo Park, CA: CSLI — Center for the Study of Language and Information.
121. Pereira F.C.N., Warren D.H.D. (1980). Definite clause grammars for language analysis - a survey of the formalism and comparison with augmented transition networks. *Artificial Intelligence*, 13: 231-278.
122. Pereira L.M., Pereira F., Warren D.H.D. (1978). *User's Guide to DECsystem-10 Prolog*. University of Edinburgh: Department of Artificial Intelligence.
123. Pitrat J. (1977). A chess combination program which uses plans. *Artificial Intelligence*, 8: 275-321.
124. Platt A., Schaeffer J., Pijls W., de Bruin A. (1996). Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87: 255-293.
125. Plotkin G. (1969). A note on inductive generalisation. In: Meltzer B., Michie D. (eds). *Machine Intelligence*, 5. Edinburgh University Press.
126. Poole D., Mackworth A., Gaebel R. (1998). *Computational Intelligence: A Logical Approach*. Oxford University Press.
127. Quinlan J.R. (1979). Discovering rules by induction from large collections of examples. In: Michie D. (ed.). *Expert Systems in the Microelectronic Age*. Edinburgh University Press.
128. Quinlan J.R. (1986). Induction of decision trees. *Machine Learning*, 1: 81-106.
129. Quinlan J.R. (1990). Learning logical definitions from relations. *Machine Learning*, 5: 239-266.
130. Reiter J. (1980). *AL/X: An Expert System Using Plausible Inference*. Oxford: Intelligent Terminals Ltd.
131. Robinson A.J. (1965). A machine-oriented logic based on the resolution principle. *JACM*, 12: 23-41.
132. Ross P. (1989). *Advanced Prolog Techniques and Examples*. Harlow: Addison-Wesley.
133. Russell S.J., Norvig P. (1995). *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, NJ: Prentice Hall.
134. Sacerdoti E.D. (1977). *A Structure for Plans and Behavior*. New York: Elsevier.
135. Sacks E.P., Doyle J. (1991). *Prolegomena to any future qualitative physics*. New Jersey: Princeton University, Report CS-TR-314-91. См. также: *Computational Intelligence Journal*.
136. Sammut C., Banerji R. (1986). Learning concepts by asking questions. In: Michalski R.S., Carbonell J., Mitchell T. (eds). *Machine Learning: An Artificial Intelligence Approach, Volume II*. San Mateo, CA: Morgan Kaufmann.

•

137. Samuel A.L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3: 211–229. См. также: Feigenbaum E.A., Feldman L (eds). *Computers and Thought*. McGraw-Hill, 1963.
138. Say A.C.C. (1998a). L'Hopital's filter for QSIM. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20: 1–8.
139. Say A.C.C. (1998b). Improved infinity filtering in qualitative simulation. *Proc. Qualitative Reasoning Workshop 98*, Menlo Park, CA: AAAI Press.
140. Say A.C.C., Kuru S. (1993). Improved filtering for the QSIM algorithm. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 15: 967–971.
141. Say A.C.C., Kuru S. (1996). Qualitative system identification: deriving structure from behavior. *Artificial Intelligence*, 83: 75–141.
142. Shafer G., Pearl J. (eds) (1990). *Readings in Uncertain Reasoning*. San Mateo, CA: Morgan Kaufmann.
143. Shanahan M. (1997). *Solving the Frame Problem: Mathematical Investigation of the Common Sense Law of Inertia*. MIT Press, Cambridge, MA.
144. Shannon C.E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, 41: 256–275.
145. Shapiro A. (1987). *Structured Induction in Expert Systems*. Glasgow: Turing Institute Press, совместно с Addison-Wesley.
146. Shapiro E. (1983). *Algorithmic Program Debugging*. Cambridge, MA: MIT Press.
147. Shoham Y. (1994). *Artificial Intelligence Techniques in Prolog*. San Francisco, CA: Morgan Kaufmann.
148. Shortliffe E. (1976). *Computer-based Medical Consultations: MYCIN*. Elsevier.
149. Shrager J., Langley P. (1990). *Computational Models of Scientific Discovery and Theory Formation*. San Mateo, CA: Morgan Kaufmann.
150. *SICStus Prolog User's Manual* (1999). Stockholm: Swedish Institute of Computer Science. <http://www.sics.se/sicstus.html>.
151. Slagle J.R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. In: Feigenbaum E., Feldman J. (eds). *Computers and Thought*. McGraw-Hill.
152. Stabler E.P. (1986). Object-oriented programming in Prolog. *AI Expert*. (October 1986): 46–57.
153. Sterling L (1990). *The Practice of Prolog*. Cambridge, MA: MIT Press.
154. Sterling L., Shapiro E. (1994). *The Art of Prolog, second edition*. Cambridge, MA: MIT Press.
155. Sutton R.S., Barto A.G. (1998). *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.
156. Szpakowicz S. (1987). Logic grammars. *BYTE*. (August 1987): 185–195.
157. Tate A. (1977). Generating project networks. *Proc. IJCAI 77*. Cambridge, MA.
158. Touretzky D.S. (1986). *The Mathematics of Inheritance Systems*. Los Altos, CA: Morgan Kaufmann.
159. van Emden M. (1981). *Logic Programming Newsletter*, 2.
160. van Emden M. (1982). Red and green cuts. *Logic Programming Newsletter*, 2.
161. Van Hentenryck P. (1989). *Constraint Satisfaction in Logic Programming*. Cambridge, MA: MIT Press.
162. Varsek A. (1991). Qualitative model evolution. *Proc. IJCAI-91*, Sydney 1991.

163. Waldinger R.J. (1977). Achieving several goals simultaneously. In: Elcock E.W., Michie D. (eds). *Machine Intelligence 8*. Chichester: Ellis Horwood. Distributed by Wiley.
164. Warren D.H.D. (1974). *WARPLAN: A System for Generating Plans*. University of Edinburgh: Department of Computational Logic, Memo 76.
165. Waterman D.A., Hayes-Roth F. (eds) (1978). *Pattern-Directed Inference Systems*. London: Academic Press.
166. Weld D. (1994). An introduction to least commitment planning. *AI Magazine*, 15: 27–61.
167. Weld D.S., de Kleer J. (1990). *Readings in Qualitative Reasoning about Physical Systems*, San Mateo, CA: Morgan Kaufmann.
168. Wilkins D.E. (1980). Using patterns and plans in chess. *Artificial Intelligence*, 14: 165–203.
169. Winston P.H. (1975). Learning structural descriptions from examples. In: Winston P.H. (ed.). *The Psychology of Computer Vision*. McGraw-Hill.
170. Winston P.H. (1984). *Artificial Intelligence, second edition*. Reading, MA: Addison-Wesley.
171. Winston P.H. (1992). *Artificial Intelligence, third edition*. Addison-Wesley.
172. Wirth N. (1976). *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ: Prentice Hall.

Предметный указатель

2

2-3-дерево, 215

A

AL0

Advice Language 0, 545

AVL-дерево, 215; 221

AVL-словарь, 222

B

BK

Background Knowledge, 447

BNF

Backus-Naur Form, 510

В-дерево, 225

C

CLP

Constraint Logic Programming, 324

Colmerauer, 18

D

DCG

Definite Clause Grammar, 510

DC-грамматика, 510

de Morgan, 89

E

EBG

Explanation-Based Generalization, 564

F

F-значение, 269

f-значение, 249

H

HYPER

HYPothesis refinER, 446

I

ILP

Inductive Logic Programming, 446

K

Kowalski, 18

L

LSS

Least Specific Specialization, 462

M

Michie, 22

m-оценка, 436

Q

QDE

Qualitative Differential Equation, 487

R

Robinson, 73

V

van Emden, 18

W

Warren, 18

Web-узел

сопровождающий, 21

Winograd, 15

А

Абстрагирование
 возрастающих временных
 последовательностей, 480
 данных, 102
 производной по времени, 480
 функций, 480
 числовых данных, 480
 Абстракция
 качественная, 480
 качественная дифференциального
 уравнения, 506
 Автомат
 конечный недетерминированный, 103
 Агент, 341
 Аксиома, 39
 Алгоритм
 A*, 255
 AO*, 289; 299
 QSIM, 502
 RBFS, 270
 альфа-бета-отсечения, 538
 Евклида, 579
 обеспечения совместимости, 303
 обеспечения совместимости дуг, 304
 охвата, 422
 поиска, 232
 поиска допустимый, 255
 сортировки, 193
 сортировки quicksort, 194
 эвристический, 255
 Альфа-бета-алгоритм, 532; 537
 Альфа-бета-интервал, 540
 Альфа-бета-отсечение, 538
 Альфа-бета-процедура, 540
 Анализ
 данных интеллектуальный, 409
 синтаксический, 512
 целей и средств, 387; 388
 Арность
 предиката, 176
 функтора, 50
 Атом, 29; 46
 end_of_file, 138
 nil, 198
 stop, 139; 165
 Атрибут
 двухзначный, 430
 наиболее информативный, 427

Б

База
 данных, 98

знаний, 327; 358; 560

Библиотека
 предикатов, 552

В

Вакансия, 185
 ван Эмден, 18
 Вариант
 качественного поведения, 497
 основной, 535
 поведения фиктивный, 505
 предложения, 57
 Вектор
 значений атрибутов, 419
 переменных, 341
 Величина
 качественная относительная, 497
 Вероятность
 апостериорная, 343
 априорная, 343
 возможного события, 345
 конъюнкции, 345
 невозможного события, 345
 отрицания, 345
 субъективная, 337; 341
 условная, 343
 Версия
 Common Lisp, 15
 SICStus Prolog, 306
 программы predecessor, 69
 Вертикальность, 54
 Взрыв
 комбинаторный, 243
 Виноград, 15
 Внесение
 предложения в базу данных, 161
 Возврат
 списка целей в предшествующее
 состояние, 395
 Вопрос, 27; 34; 57
 "How?", 336
 "Why?", 336
 "для чего", 336
 "как", 336
 диагностический, 486
 прогностический, 486
 управленческий, 486
 Время
 завершения, 302
 завершения расписания, 261
 простоя, 261
 Вхождение
 подвыражения в выражение, 157
 Вывод

данных в файл, 136
 дерева решения логический, 426
 логический обратный, 331
 логический прямой, 331; 333
 списка, 141
 управляемый данными, 335
 управляемый целями, 335
 Выдвижение
 гипотезы, 331
 Выделение
 подписка, 189
 Вызов
 операции на выполнение, 92
 Выполнение
 целей, 58
 Выражение
 арифметическое, 92
 Высказывание, 330; 359
 логическое, 522
 Высказывания
 независимые, 344
 несовместимые, 344
 Высота
 дерева, 201
 Вычисление
 куба числа, 139
 максимума, 125
 символьное, 47
 Вычитание
 множеств, 131
 Выявление
 знаний, 330

Г

Генератор
 вариантов плана, 393
 Гипотеза
 более конкретная, 458
 более общая, 458
 достоверная, 447
 конкретная, 411
 общая, 411
 охватывающая пример, 463
 полная, 447
 совместимая с данными, 411
 эвристическая, 248
 Глагол
 переходный, 524
 Глубина
 усовершенствования, 456
 Го, 532
 Голова
 предложения, 31; 34
 списка, 76

Головоломка
 "игра в восемь", 230
 Горизонтальность, 55
 Градус
 Фаренгейта, 307
 Цельсия, 307
 Грамматика, 510
 определенных предложений, 510
 Грань
 открытая, 384
 Граф, 192
 AND/OR, 277; 331
 ориентированный, 208
 переходов, 103
 решения, 280
 связный, 211
 семейных отношений, 449
 усовершенствования, 450

Д

Данные
 входные, 143
 зашумленные, 433
 де Морган, 89
 Действие, 384
 climb, 63; 64
 grasp, 63; 64
 push, 63; 64
 walk, 63
 допустимое, 63; 229
 неконкретизированное, 403
 Декомпозиция
 задачи на подзадачи, 277
 списка, 80
 терма, 156
 Делитель
 наибольший общий, 93; 579
 Дерево, 49; 192; 218
 AND/OR, 284; 533
 l, 218
 n2, 218
 n3, 218
 nil, 218
 бинарное, 77; 197
 двоично-троичное, 215
 доказательств, 336; 367
 игры, 532; 534
 одноэлементное, 201
 остовное, 209
 поиска, 291; 534
 полностью сбалансированное, 215
 пустое, 198; 218
 решения, 280; 411
 с двумя поддеревьями, 218

- с одним узлом, 218
- с тремя поддеревьями, 218
- сбалансированное, 201; 215
- семейных отношений, 26
- синтаксического анализа, 516
- упорядоченное слева направо, 199
- форсирующее, 544
- Деятель, 524
- Диагноз, 342
- Диагональ
 - восходящая, 113
 - нисходящая, 113
- Дизъюнкт
 - хорновский, 73
- Дизъюнкция
 - целей, 57
- Диод, 317
- Директива, 88
 - dynamic, 162
 - ор, 88
- Добавление
 - объекта в список, 78
 - фонового литерала к предложению, 461
 - элемента в список, 82
- Доказательство
 - теорем автоматическое, 583
- Документированность, 170
- Дополнение
 - числа нулями, 153
- Достижение
 - цели, 39
- Дружественность, 169
- Дуга, 208
 - совместимая, 303
 - совместимая с определяемым ограничением, 303

Е

Европа, 18

З

- Зависимость
 - контекстная, 515
 - конъюнктивная, 330
- Загрузка
 - программы, 591
- Задача
 - диагностическая, 482
 - планирования, 383
 - поиска кратчайшего маршрута, 281
 - с восемью ферзями, 111
 - с обезьяной и бананом, 62

- с ханойской башней, 282
- составления расписания, 304; 310
- удовлетворения ограничений, 302
- Заклучение
 - правила, 31
- Закон
 - Кирхгоффа, 483
 - Ома, 483
 - фундаментальный, 483
- Замена
 - переменных структурированными терминами, 451
- Запись
 - в операторной форме, 87
 - операторная, 332
 - списковая, 78
- Запрос
 - ttyflush, 141
 - к пользователю, 357
- Запятая, 34; 306
- Защита
 - целей, 392
- Знак
 - конъюнкции, 34
 - операции, 88
- Знания
 - априорные, 344
 - вероятностные, 329
 - категорические, 329
 - типовые, 543
 - фоновые, 446; 564
- Значение
 - декларативное, 42; 45; 56
 - минимаксное, 535
 - программ Prolog, 56
 - программы, 45
 - процедурное, 42; 45; 56; 58
 - семантическое, 523
 - смысловое, 523
 - статическое, 535

И

- Игра
 - с двумя участниками, 283
 - с одним участником, 63
 - с полной информацией, 283
- Игрок
 - MAX, 535
 - MIN, 535
 - свой, 284; 533
 - чужой, 284; 533
- Идентичность
 - двух термов, 160
- Иерархия

понятий, 417
 Изучение
 языка программирования, 16
 Имя
 отношения, 26
 переменной, 47
 функтора, 50
 Индекс
 Gini, 430
 Индикатор, 251; 297
 успеха/неудачи, 58
 Интеллект
 искусственный, 16; 255
 Интерпретатор
 байесовских сетей доверия, 346
 гипотез, 453
 для объектно-ориентированных
 программ, 571
 для правил, 333
 качественных дифференциальных
 уравнений, 493
 командный, 328
 командный экспертной системы, 328;
 376
 обеспечивающий трассировку, 562
 правил вывода, 560
 правил обратного логического вывода,
 333
 правил прямого логического вывода,
 334
 правил с оценками достоверности, 338
 языка Advice Language, 543
 языка ALO, 546
 Интерпретация
 логическая, 522
 осторожная, 454
 причинно-следственная, 343
 смысловая, 522
 Интерфейс
 пользовательский, 99; 327
 Инфимум, 308
 Информатика, 19
 Информация
 вступительная, 177
 входная, 335
 заключительная, 177
 о повторных вызовах, 177
 остаточная, 428
 производная, 335
 Исключение
 взаимное вариантов, 143
 одного примера, 441
 Источник
 питания, 317
 Исчисление

вероятностей, 341
 Итерация, 183

К

Камера
 телевизионная, 406
 Карта
 дорожная, 249
 Европы, 179
 Квадрат, 52; 156
 Квантор, 73
 всеобщности, 34; 134
 существования, 134; 525
 Классификация
 объектов данных, 46
 с разбивкой по категориям, 127
 Клетка
 исходная, 257
 критическая, 552
 Ковальски, 18
 Код
 ASCII, 143
 Количество
 посторонних включений, 428
 Колмероз, 18
 Комментарий, 38; 175
 Комментирование, 175
 Компиляция
 файла с программами, 146
 Компоновка
 программы, 32; 174
 Компьютер
 пятого поколения, 18
 Конкатенация
 списков, 78; 181
 Конкретизация, 52
 менее общая, 53
 наиболее общая, 53
 переменных, 31; 58
 Конкретность
 гипотезы, 419
 Конструкция, 516
 глагольная, 516
 именная, 516
 Контейнер, 489
 Контроль
 текущий, 335
 Конъюнкция
 условий, 32
 целей, 29; 57
 Копия
 резервная, 290
 Корень
 дерева, 49

Коэффициент
веса, 463
ветвления эффективный, 540
достоверности, 337
приращения информации, 429
Критерий
аналогичности, 410
независимости, 281
операционности, 565
оптимизации, 281
успеха обучения, 440
Кэширование, 163; 178; 188

Л

Лес, 463
Литерал
фонный, 450
Логика
математическая, 19; 73
предикатов, 446
предикатов первого порядка, 73

М

Манипулирование
геометрическими фигурами, 156
символами, 143
фотокамерой, 385
Маршрут, 108
кратчайший, 249
Массив
моделируемый, 185
Машина
логического вывода, 327
Мера
доверия, 337
Метаинтерпретатор, 559
ebg, 568
Метапрограмма, 559
Метапрограммирование, 559
Метка
null, 104
в графе переходов, 103
Метод, 570
area, 571
EBG, 567
IDA*, 266
RBFS, 269
вакансий, 185
ветвей и границ, 315
дихотомизации атрибутов, 429
изображения дерева, 206
обеспечения совместимости, 303
обратного логического вывода, 357

поиска в бинарном словаре, 199
предотвращения циклов, 68
раздваивания атрибутов, 429
резольюции, 583
Механизм
приводной, 521
распознавания, 512
регрессии целей, 396
согласования, 62
Мир
блоков, 15
в котором существует обезьяна, 63
Мичи, 22
Множество
всех подмножеств заданного
множества, 167
испытательное, 441
конфликтное, 577
обучающее, 441
отсекаемое, 438
растущее, 438
универсальное, 410
фактов, 98
Моделирование
в ограничениях, 317
массива, 185
машинное числовое, 479
теоретическое качественное, 481
числовое, 317
Модель
ванны, 490
движения с ускорением, 501
качественная, 478
количественная, 478
числовая, 481
электрической схемы, 318
Модификация
программы, 174
процедурного значения, 45
Модифицируемость, 170; 328
Модуль, 591
управляющий, 577
Модульность, 328

Н

Набор
обучающих данных, 442
переменных, 55
Надежность, 170
Назначение
процессора текущее, 262
Накопление
смысловых значений, 527
Наращиваемость, 328

Наследование, 350
множественное, 573
Неравенство
линейное, 306
Ним, 534
Ничья, 284; 533

О

Область
определения, 303; 341; 496
определения имен переменных, 48
определения конечная, 321
применения языка Prolog, 178
рабочая, 492
Обнаружение
скрытых закономерностей, 409
Обобщение, 172
на основе объяснения, 559; 564
первоначальной проблемы, 112
Обозначение
“f”, 88
“x”, 88
“y”, 88
списка с вертикальной чертой, 78
Обучение
в результате открытия, 408
индуктивное, 409
инкрементное, 422
машинное, 446
на основе атрибутов и значений, 447
на примерах, 408
пакетное, 422
реляционное, 447
Общность
гипотезы, 419
Объединение
представлений, 72
символов в атом, 144
Объект, 571
геометрический, 49
данных, 45
структурированный, 45; 48
Объявление
типа данных, 45
Объяснение, 364
последовательности рассуждений, 336
предпосылок, 336
Ограничение, 488
deriv, 497
mplus, 497
QDE, 497
sum, 497
арифметическое, 321
бинарное, 303

длины доказательств, 452
дополнительное, 309
качественное, 490
на ресурсы, 313
предшествования, 263; 302
равенства между термами, 305
суммирования, 497
хода, 545
числовое, 306
Ограничения
своего хода, 543
чужого хода, 543
Округление, 47
Окружность, 52; 156
Операнд
операции is, 92
Оператор, 88
“~”, 166
“\==”, 160
“==”, 160
“--->”, 285
“юнив”, 156
not, 165; 359
not префиксный, 130
инфиксный, 156
инфиксный “<=”, 336
инфиксный via, 299
отсечения, 121; 164; 174
отсечения зеленый, 133; 174
отсечения красный, 133; 174
проверки на равенство, 159
сравнения, 159
усовершенствования, 450
Операция
comp, 240
is, 92
SCANNING, 59
арифметическая, 92
вставки, 202; 217
записи, 138
инфиксная, 87; 88
конкатенации, 195
получения консультации, 147
постфиксная, 88
префиксная, 88
проверки на равенство, 93
разбиения списка, 195
с базой данных, 161
с термами, 52
со списком, 78
согласования, 54; 524
сопоставления, 93
сравнения, 93
суммирования, 484

- увеличения геометрической фигуры, 156
- удаления, 203
- удаления листа, 203
- чтения, 138
- Описание
 - достоверное, 422
 - качественное, 478
 - на основе атрибутов и значений, 412
 - охватывающее объект, 422
 - полное, 422
 - понятия операционное, 564
 - реляционное, 412
- Определение
 - декларативного значения, 57
 - длины списка, 94
 - знака операции, 88
 - объекта по структурным свойствам, 100
 - операции, 88; 90
 - отношения, 43
 - пространства планирования, 385
 - рекурсивное, 37
 - реляционное, 448
 - смысла, 520
 - смысла естественного языка, 521
- Оптимизация
 - линейная, 307
 - последнего вызова, 182; 183
 - хвостовой рекурсии, 183
- Опция
 - разметки “ff”, 324
- Организация
 - табличная, 175
- Отладка
 - программы, 176
- Отметка, 488
- Отношение, 26; 29
 - a_kind_of, 351
 - accepts, 105; 108
 - actor, 524
 - add, 127; 202
 - add_at_end, 190
 - add23, 217; 221
 - addavl, 221
 - addleaf, 202
 - addroot, 205
 - adds, 387
 - adjacent, 210
 - alphabet, 539
 - AND, 279
 - askable, 371
 - attack, 131
 - aunt, 34
 - avl, 224

- born, 93
- breadthfirst, 238
- canget, 86
- class, 127
- combine, 221
- conc, 79; 111
- concat, 182
- copy_term, 164
- count, 151; 160
- del, 82
- deptime, 108
- different, 33
- digitsum, 154
- dividelist, 86
- enlarge, 156
- expand, 297
- fact, 334
- female, 30
- final, 104
- flatten, 86
- flight, 108
- forwardfib, 188
- goal, 257
- grandchild, 34
- grandparent, 28
- gt, 192; 200
- h, 257
- hamiltonian, 210
- happy, 34
- hastwochildren, 34
- impossible, 396
- in, 198; 221
- ins, 217
- ins2, 221
- instance_of, 351
- isa, 349
- jump, 119
- knightpath, 120
- length, 95; 101
- length1, 95
- linearize, 201
- male, 30
- max, 95; 126
- maxelement, 201
- member, 79; 179; 180; 210
- member1, 81
- minimax, 536
- mother, 32
- move, 64; 66; 513
- moves, 536
- newbounds, 540
- ngb, 179
- no_attack, 131
- noattack, 113; 234
- not, 130

nqueens, 173
 nthchild, 103
 object, 570
 offspring, 30
 OR, 279
 parent, 28
 path, 209; 210
 path1, 210
 permutation, 84; 111
 powerset, 167
 predecessor, 35; 68; 457
 qsum, 483
 regular, 56
 reverse, 85; 190
 route, 108
 s, 256
 safe, 115
 set_difference, 131
 shift, 85
 silent, 104
 sister, 33
 sol, 118
 solve, 232; 286
 sort, 192
 split, 196
 starts, 146
 sublist, 83
 subset, 86
 substitute, 157
 subsumes, 159
 sum, 152
 sum1, 153
 timetable, 107
 trans, 104
 transfer, 108
 transition, 108
 translate, 86
 бинарное, 30
 обобщения, 458
 одноместное, 30
 предшествования, 161; 313
 принадлежности, 349
 селективное, 102
 унарное, 30
 упорядочения, 192; 199
 Отображение
 дерева, 206
 Отрезок
 вертикальный, 54; 56
 горизонтальный, 55
 прямой, 49
 Отрицание
 вследствие недостижения цели, 121
 как недостижение цели, 129
 Отсечение

последующее, 433
 предварительное, 433
 с минимальной ошибкой, 434
 частей деревьев, 433
 эвристическое, 541
 Отступ, 32
 Оценка
 лапласовская, 437
 оптимистическая, 263
 статическая, 541
 точности гипотезы, 440
 упорядоченности, 259
 эвристическая, 247; 260
 Ошибка
 зафиксированная, 435
 классификации, 434
 статическая, 435
 числовая, 47

П

Палиндром, 85
 Пара
 разностная, 181
 Параметр
 Answer, 452
 Solved, 253
 входной, 176
 выходной, 176
 накапливающий, 182; 183
 селективного отношения, 102
 Перебор
 с возвратами, 62; 121; 163
 с возвратами автоматический, 121
 Переменная, 29
 анонимная, 47
 Переполнение
 стека, 309
 Перестановка
 элементов в списке, 84
 Переупорядочение, 179
 предложений, 68
 целей, 68
 Переход
 недетерминированный, 103
 скрытый, 104
 Переходы
 между качественными состояниями,
 499
 План
 вставки новой пленки, 385
 Планирование, 302; 383
 авиаперелетов, 107
 в мире блоков, 385
 с частичным упорядочением, 404

- Планировщик, 383
 - маршрутов, 109
 - нелинейный, 404
 - с регрессией целей, 397
 - с частичным упорядочением, 404
- Победа, 284; 533
- Поверхность
 - верхняя открытая, 384
- Подготовка
 - фотокамеры к работе, 385
- Поддереву, 49
- Подсписок, 83
- Подстановка
 - 0, 459
- Подсчет
 - количества элементов в списке, 94
- Подтверждение
 - предложения, 161
- Подцель, 32
- Позиция
 - вставки в базу данных, 163
 - выигрышная, 284
 - заключительная, 532
 - на шахматной доске, 111
 - своего хода, 284
 - текущая, 138
 - чужого хода, 284
- Поиск
 - в глубину, 232
 - в глубину с итеративным углублением, 237
 - в глубину с ограничением по глубине, 287
 - в графе AND/OR, 285
 - в пространстве состояний, 383
 - в ширину, 238
 - жадный, 425
 - маршрута, 108
 - наибольшего из двух чисел, 125
 - остовного дерева графа, 212
 - ответов, 39
 - по заданному критерию, 247; 249; 460
 - по принципу "подъема к вершине", 425
 - поглощающий, 425
 - эвристический, 245; 247
 - элемента в списке, 80
- Получение
 - консультации из файла, 146
 - ответа на вопрос, 58
- Понятие
 - операционное, 564
- Понятность
 - гипотезы, 440
- Поражение, 284; 533
- Порядок
 - расположения целей, 67
- Последовательность
 - доказательства, 40
- Постижимость
 - гипотезы, 440
- Поток
 - входной, 137
 - входной текущий, 137
 - выходной, 137
 - выходной текущий, 137
- Правила
 - игры, 63
 - именования файлов, 137
 - распространения значений, 536
 - суммирования, 152
- Правило, 31; 34
 - "if-then", 328
 - "условие-действие", 419
 - в базе знаний, 330
 - вывода, 328; 544
 - минимизации, 290
 - непротиворечивое, 422
 - порождающее, 328
 - рекурсивное, 37
 - решения задач эвристическое, 71
 - согласования термов, 53
- Правильность, 169
- Предикат
 - accepts, 105
 - achieves, 395
 - actor, 524
 - add1, 469
 - ako, 474
 - all_different, 322
 - assert, 161
 - asserta, 161; 163
 - assertz, 161
 - atom, 151
 - backliteral, 450
 - best_search, 468
 - binarytree, 201
 - call, 165
 - canget, 64
 - checkmove, 552
 - choose_attribute, 431
 - circuit_a, 319
 - circuit_b, 319
 - circuit1, 485
 - complete, 456
 - compose, 524
 - consistent, 456
 - consult, 163
 - depth_first, 456
 - deriv, 497

dictionary, 201
 domain, 321
 driver, 144
 end_of_game, 549
 eval, 468
 even, 470
 evenlength, 85
 ex, 449
 father, 16
 female, 446
 fib, 186; 308
 getsentence, 144
 gives, 567
 goal, 256
 grandfather, 16
 ground, 159
 has_daughter, 447; 450
 indomain, 321
 induce, 456
 induce_hyp, 468
 init_counts, 469
 insert_sorted, 474
 intrans_verb, 524
 is_true, 336
 iter_deep, 456
 kingdiagfirst, 552
 labeling, 322
 legal, 552
 length, 95
 likes, 522; 567
 lpatt, 552
 male, 446
 mate, 551
 max_clause_length, 469
 max_clauses, 461; 469
 max_proof_length, 469
 maxlist, 95
 moves, 536
 mplus, 497
 name, 159
 newroomsmaller, 551
 nex, 450
 nl, 139
 nomove, 552
 nonvar, 154
 nospy, 177
 not, 130
 notrace, 177
 odd, 470
 oddlength, 85
 okapproachedcsquare, 552
 once, 164
 order, 144
 ordered, 96
 paints, 522

palindrome, 85
 parent, 16; 446
 path, 471
 prec, 311
 prec_constr, 312
 precedence_constr, 312
 prolog_predicate, 450
 prove, 452; 468
 rectangle, 570
 refine, 455; 468
 refine_hyp, 456; 468
 resource, 311; 316
 retract, 161
 rookdivides, 551
 rookexposed, 551
 rooklost, 551
 rookmove, 552
 roomgt2, 552
 sad, 567
 satisfy, 431
 schedule, 313
 see, 137
 select, 395
 show_counts, 469
 show_hyp, 469
 solution, 111
 spy, 177
 stalemate, 551
 start_clause, 469
 start_hyp, 456
 start_hyps, 468
 subsum, 96
 sum, 497
 sumlist, 95
 support/2, 474
 tasks, 311
 taxi, 144
 terminallost, 533
 terminalwon, 533
 touch/2, 474
 trace, 177
 transition, 499
 unifiable, 132
 verb_phrase, 524
 встроенный, 95; 136; 149
 встроенный "=", 156
 встроенный arg, 156; 158; 184
 встроенный atom, 150
 встроенный atomic, 150
 встроенный bagof, 165
 встроенный call, 158
 встроенный clause, 561
 встроенный compile, 147
 встроенный compound, 150
 встроенный copy_term, 164

- встроенный findall, 165
- встроенный float, 150
- встроенный functor, 156; 158; 184
- встроенный get, 138
- встроенный get0, 138
- встроенный inf, 308
- встроенный integer, 150
- встроенный is, 307
- встроенный maximize, 307
- встроенный minimize, 307
- встроенный name, 144
- встроенный nonvar, 150
- встроенный number, 150
- встроенный put, 138
- встроенный read, 138; 139
- встроенный setof, 165; 180
- встроенный sup, 308
- встроенный var, 150
- встроенный write, 138; 139
- встроенный форматирования вывода, 139
- детерминированный, 139
- динамический, 162; 590
- металогический, 154
- недетерминированный, 162
- неопределенный, 590
- определения принадлежности к списку, 460
- разметки, 323
- сравнения, 552
- стандартный, 148
- статический, 590
- фоновый, 446
- целевой, 447
- Предложение, 26; 29; 34
 - неудовлетворяемое, 462
 - относительное, 527
 - хорновское, 73
- Предохранитель
 - плавкий, 359
- Предположение
 - о гладкости, 491
 - о замкнутости мира, 121; 133
- Предпосылка, 384
 - действия, 64
- Представление
 - в виде графа AND/OR, 277
 - графа, 208
 - декларативное, 72
 - древовидное, 49
 - информации о семье, 99
 - множества, 197
 - наборов данных, 215
 - понятия в виде множества, 409
 - процедурное, 72
 - списка внешнее, 76
 - списка внутреннее, 77
 - формальное, 411
 - целевое, 522
 - числа с плавающей точкой, 47
- Представления
 - физические обыденные, 479
- Преемник, 231
- Преобразование
 - в процедуру с хвостовой рекурсией, 183
 - обычного списка в разностный, 196
 - переменной в фоновый терм, 461
 - текста в список атомов, 145
- Префикс
 - "-", 176
 - "+", 176
- Прибор
 - электрический, 359
- Приглашение, 140
 - к вводу информации, 357
- Пример
 - отрицательный, 446
 - положительный, 446
 - учебный, 565
- Принцип
 - минимакса, 532; 534
 - обобщения, 119
 - резолюции, 73
- Приобретение
 - знаний, 409
- Приоритет
 - знака операции, 87
 - операнда, 88
- Причина
 - коренная, 343
- Проблема
 - планирования, 261
 - удовлетворения ограничений, 301
- Проверка
 - k-кратная перекрестная, 441
 - вхождения, 73
 - на неидентичность, 160
 - на неравенство, 306
 - на равенство, 306
 - принадлежности к множеству, 86; 198
 - принадлежности к списку, 78
 - типа терма, 149
- Прогноз, 342
- Программа, 58
 - addleaf, 202
 - ARCHES, 419
 - conc, 82
 - convert, 307
 - family, 37

- HYPER, 446; 448
- length, 94
- MINIHYPER, 448; 454
- schedule, 312
- ведения игры, 541
- интерпретируемая, 147
- качественного машинного моделирования, 493
- конкатенации списков, 448
- обобщения на основе объяснения, 568
- обучающаяся, 420
- откомпилированная, 147
- планирования путешествий, 108
- решения числовых ребусов, 154
- свернутая, 521
- сортировки списков, 448
- составления расписания, 313
- целенаправленная, 15
- шахматная, 540
- эмулятора конечного автомата, 105
- Программирование
 - автоматическое, 446
 - в ограничениях, 301
 - декларативное, 16
 - логическое в ограничениях, 301
 - логическое индуктивное, 446
 - объектно-ориентированное, 570
 - рекурсивное, 37
 - управляемое шаблонами, 579
 - целевое, 15
- Продление
 - поиска, 541
- Продукция, 328
- Прозрачность
 - системы, 328
- Пространство
 - планирования, 384
 - поиска, 534
 - состояний, 228; 277
 - состояний бесконечное, 235
- Процедура, 37; 38
 - actor, 524
 - add_to_tail, 155
 - adds, 384
 - andor, 298
 - assign_processors, 315
 - bars, 141
 - between, 96
 - bubblesort, 193
 - can, 384
 - choose_attribute, 432
 - collect, 190
 - combine, 298
 - compute, 158
 - continue, 254; 298
 - count, 151
 - cube, 139
 - deletes, 384
 - depth_first_iterative_deepening, 237
 - depthfirst1, 237
 - depthfirst2, 236
 - dosquares, 165
 - ebg, 569
 - enlarge обобщенная, 157
 - execute, 58; 59
 - expand, 251; 253
 - expandlist, 297
 - expert, 366; 377
 - explore, 366
 - fact, 332
 - fib/2, 308
 - fib2, 187
 - find1, 241
 - find2, 241
 - find3, 241
 - forwardfib, 188
 - gen, 118
 - getletters, 145
 - getreply, 369
 - getrest, 144
 - getsentence, 144
 - height, 201
 - impurity1, 432
 - in, 198
 - induce_trees, 431
 - InduceOneRule, 422
 - insertsort, 193
 - is_true, 333
 - learn, 426
 - makelist, 180
 - maketable, 163
 - maplist, 172
 - max, 190
 - merge, 175
 - move, 513
 - numbervars, 376
 - obtain, 158
 - path, 236
 - permutation, 84
 - plan, 396
 - playgame, 549
 - plural, 146
 - present, 366; 376
 - processfile, 142
 - prove, 560
 - prunetree, 439
 - quicksort, 194
 - quicksort2, 196
 - reverse, 184
 - satisfied, 403

schedule, 312
 search, 146
 send, 571
 show, 206; 287; 298
 show2, 299
 showfile, 142
 simplify, 155
 sol, 118
 solution, 112
 solve, 234; 287
 split, 129
 squeeze, 143
 sub1, 189
 sub2, 189
 sub3, 189
 substitute, 159
 succlist, 254
 sumlist, 183
 useranswer, 366; 369
 value, 352
 write, 139
 writelist, 141
 writelist2, 141
 writenode, 299
 встроенная, 92
 встроенная asserta, 187
 встроенная copy_term, 569
 поиска, 450
 поиска в глубину в графе AND/OR, 285
 прямого логического вывода, 334
 рекурсивная, 182
 решения задач в ограничениях, 307
 с хвостовой рекурсией, 183
 управляющая, 377
 формирования рассуждений, 331
 целенаправленная, 15
 Процесс
 согласования термов, 54
 Процессор, 261
 Прочтение
 декларативное, 56
 процедурное, 56
 Прямоугольник, 52; 56; 156
 Псевдофайл, 137
 Псевдоцель, 123
 Пункт
 ключевой, 298
 Путь
 ациклический, 209
 гамильтонов, 210
 с минимальной стоимостью, 211

Р

Равенство
 буквальное, 160
 линейное, 306
 Разбиение
 атома на символы, 144
 списка, 84
 Развертывание
 цели, 570
 Развитие
 языков программирования, 15
 Раздваивание
 атрибута, 429
 Разделение
 знаний и алгоритмов, 328
 Размер
 гипотезы, 463
 дерева, 292
 Разметка, 322
 Разработка
 программ Prolog, 171
 Раскраска
 карты, 179
 Распечатка
 дерева, 206
 Расписание
 авиаперелетов, 108
 допустимое, 261
 пустое, 262
 частично составленное, 262
 Распознавание
 типа объекта, 45
 Распределение
 ресурсов по процессам, 302
 Расстояние
 манхэттенское, 259
 отступа, 207
 суммарное, 259
 Рассуждения
 качественные, 478
 функциональные, 482
 Реализация
 отношения findall, 167
 Ребро
 ориентированное, 208
 Ребус
 числовой, 151
 Регрессия
 целей, 395
 Режим
 интерактивный, 147
 итеративного углубления, 393
 Резистор, 51; 317
 Результат, 384

Рекурсия, 171
 хвостовая, 182
Решатель
 задач, 15
Робинсон, 73
Робот
 подвижный, 256
Ряд
 Фибоначчи, 186

С

Свойство
 магическое, 104
 оптимальных расписаний, 261
Связь, 26; 349; 384
 AND, 278; 280
 OR, 280
Селектор, 102
Семантика
 Prolog процедурная, 65
Семейство
 треугольников, 56
Сеть
 байесовская, 340; 342
 доверия байесовская, 340
 логического вывода, 330
 ограничений, 303
 семантическая, 349
 электрическая, 359
Символ, 138
 “::=”, 512
 “..>”, 512
 “null”, 426
 непечатаемый, 138
 непробельный, 143
 нетерминальный, 511
 подчеркивания, 47; 99
 пустой, 104
 терминальный, 511
Синтаксис
 DEC-10, 20
 эдинбургский, 20
Синтез
 структурный, 482
Система
 AL/X, 329
 AL3, 329
 CLP, 305
 CLP(B), 306
 CLP(FD), 306
 CLP(Q), 306; 310
 CLP(R), 306
 CLP(Z), 306
 MYCIN, 329

Shrdlu, 15
 обозначений DCG, 520
 обозначений инфиксная, 51
 основанная на знаниях, 326
 управляемая шаблонами, 576
 экспертная, 326
Ситуация
 игры первоначальная, 532
 проблемная, 228
Сканирование
 программы, 59
Слияние
 списков, 196
Словарь
 бинарный, 199
 бинарный несбалансированный, 216
 двоично-троичный, 215; 216
Слово
 ключевое, 146
 определяющее, 525
 определяющее “а”, 525
 определяющее “every”, 526
Сложность
 вычислительная, 440
 деревьев игр в шахматах, 534
 комбинаторная, 425; 448
 логарифмическая, 201
Слот, 350; 524
Случай
 граничный, 172
 общий, 172
Смысл
 движения, 519
 имени собственного, 522
 непереходного глагола, 522
 фразы, 517
Событие, 341
Совет, 543
 элементарный, 543
 элементарный выполнимый, 544
Совместимость, 303
Согласование, 45; 52; 73
 как аналог унификации, 73
 переменных, 461
Соглашение
 стилистическое, 174
Соединение
 между двумя клеммами, 318
 между тремя клеммами, 318
 резисторов, 51
 резисторов параллельное, 51
Создание
 форсирующего дерева, 546
Сообщение
 “More core needed”, 69

- “Stack overflow”, 69
 - Сортировка
 - вставкой, 193
 - пузырьковая, 193
 - списка, 192
 - Составление
 - расписания, 262
 - Состояние, 262
 - конечное, 104; 492
 - мира обезьяны, 63
 - мира обезьяны начальное, 63
 - начальное, 63; 262
 - переменной качественное, 497
 - решения, 292
 - системы качественное, 497
 - целевое, 230; 262
 - Состояние-преемник, 262
 - Сосуды
 - сообщающиеся, 501
 - Сохранение
 - достигнутой цели, 391
 - полученных решений, 163
 - Спецификатор
 - типа, 88
 - Спецификация
 - байесовской сети, 343; 348
 - Список, 76; 192
 - добавления, 384
 - непустой, 76
 - поддеревьев, 294
 - пустой, 76
 - разностный, 181; 512
 - связей, 384
 - советов, 546
 - удаления, 384
 - целей, 57; 58
 - Способ
 - обновления текущей гипотезы, 417
 - самоутешения, 567
 - Сравнение
 - арифметическое, 306
 - числовых значений, 93
 - Средства
 - ведения диалога, 333
 - отладки, 176
 - удовлетворения ограничений, 301
 - управления, 164
 - Стандарт
 - ISO, 18
 - ISO/IEC 13211-1, 44
 - Prolog, 148
 - языка Prolog, 18
 - Стиль
 - программирования, 173
 - Стоимость
 - игры, 535
 - пути, 211
 - решения, 231; 262
 - Стратегия
 - SLD, 73
 - Структура, 48
 - данных, 98; 192
 - древовидная, 50; 197
 - программ CLP(FD), 322
 - рекурсивная, 172
 - с описанием семьи, 102
 - экспертной системы, 327
 - Структурирование
 - данных, 48
 - Суперобъект, 571
 - Супремум, 308
 - Сущность, 349
 - Схема
 - графическая, 173
 - движения, 64
 - комбинирования достоверностей, 338
 - отношения, 32
 - электрическая, 51; 317; 482
 - Сцена
 - действия, 405
 - Счетчик
 - ответов, 374
 - США, 18
- Т
- Таблица
 - product, 164
 - произведений целых чисел, 163
 - советов, 544
 - фактов, 163
 - Тело
 - предложения, 31; 34
 - Теорема, 39
 - Байеса, 344
 - допустимости, 255; 256; 294
 - эквивалентности де Моргана, 89
 - Теория
 - вероятностей, 339
 - информации, 429
 - проблемной области, 564; 565
 - Терм, 49
 - rectangle, 56
 - идентичный, 53
 - составной, 192
 - Терминал
 - пользовательский, 137
 - Тип
 - операции, 88
 - цели askable, 369

Точка, 49
 в двумерном пространстве, 49
 в трехмерном пространстве, 49
 Точка с запятой, 57; 175; 306
 Точность
 гипотезы, 413
 классификации, 434; 440
 предсказания, 413
 Трассировка, 177; 364; 562
 выполнения, 60
 избирательная, 177
 поиска, 291
 цели, 177
 Треугольник, 49; 156
 Турнир
 шахматный, 540
 Тэта-классификация, 458

У

Углубление
 итеративное, 232; 236; 287
 последовательное, 542
 Удаление
 объекта из списка, 78
 элемента из списка, 82
 Удобство для чтения, 169
 Удовлетворение
 ограничений, 301
 Узел
 AND, 279; 280; 284
 OR, 279; 280; 284
 графа, 103
 корневой, 532
 начальный, 229
 поиска начальный, 450
 поиска целевой, 450
 целевой, 229; 278
 Узел-посредник, 233
 Уинстон, 15
 Унификация, 73
 Уоррен, 18
 Упорядочение
 алфавитное, 161
 лексикографическое, 192
 цифровое, 161
 Управление
 перебором с возвратами, 121
 Уравнение
 QDE, 487
 дифференциальное, 479
 Усвоение
 сообщенных знаний, 408
 Условие
 достаточное, 565

правила, 31
 согласования, 52
 Усовершенствование, 450
 алгоритмов, 171
 отношений, 171
 поэтапное, 170
 Утверждение, 525
 Уточнение, 451
 наименее конкретное, 462
 Учет
 неопределенности, 337

Ф

Файл, 137
 user, 137
 входной, 137
 выходной, 137
 последовательный, 138
 с произвольным доступом, 138
 текстовый, 138
 Факт, 26; 34
 производный, 186
 Форма
 конъюнктивная нормальная, 73
 обучения, 408
 предложений, 73
 представления в виде предложений, 584
 синтаксическая, 45
 Форматирование
 объектов данных, 136
 Формирование
 вариантов программы, 68
 дерева доказательства, 563
 имен системой, 55
 объяснения, 336
 расписания, 262
 Формула
 Байеса, 436
 исчисления вероятностей, 344
 энтропии, 429
 Формулировка
 предиката рекурсивная, 65
 Фраза, 511
 Фрейм, 350
 Функтор, 48
 а, 208
 digraph, 208
 е, 208
 graph, 208
 par, 51
 point; 49
 point3, 49
 rectangle, 56

seg, 49
 seq, 51
 state, 63
 triangle, 49
 списка, 77
 терма главный, 49; 87
Функция
 h, 248
 двухступенчатая, 121
 монотонно возрастающая, 490
 монотонно убывающая, 490
 оценки, 534
 оценки Cost, 463
 оценки статическая, 536
 эвристическая, 263; 535
 эвристическая h, 289

Х

Хвост
 списка, 76
Ход
 правильный, 535

Ц

Цель, 29
 assert, 161
 consult, 146; 147
 fail, 129; 165
 functor, 184
 get0, 143
 getsentence, 144
 in, 198
 put, 143
 read, 139
 repeat, 165
 retract, 161
 see, 137
 seen, 138
 setof, 166
 squeeze, 143
 sum, 152
 tab, 139
 tell, 137
 told, 138
 treat, 142
 true, 130; 165
 write, 139; 206
 игры, 63
 консервативная, 543
 лучшая, 543
 неконкретизированная, 403
 отрицаемая, 378

отрицаемая неконкретизированная,
 134
 родительская, 124
Цикл
 бесконечный, 67
 скрытый, 106
Цифра
 переноса, 152

Ч

Число, 47
 с плавающей точкой, 47
 Фибоначчи, 187; 308
 целое, 47
Чтение
 данных из файла, 136
 программ, 136; 146

Ш

Шаблон
 L-образный, 552
Шахматы, 532
Шашки, 532
Школа
 ортодоксальная, 19

Э

Эвристика, 245
Экземпляр
 отношения, 26
 предложения, 57
Эксперт
 специализированный, 544
Эмулятор
 конечного автомата, 103; 108
Эндшпиль
 шахматный, 543
Этап
 усовершенствования, 457
Эффект
 горизонта, 542
Эффективность, 169
 вычислительная, 178
 поиска в словаре, 200

Я

Язык, 511
 Advice Language 0, 545
 AL0, 545
 Fortran, 15

Lisp, 15
Microplanner, 15
Pascal, 61
Prolog, 18
Prolog чистый, 74
высокого уровня, 15
гипотез, 411; 412; 447
декларативного типа, 15

низкого уровня, 15
описания объектов, 412
описания понятий, 411; 412
процедурного типа, 15
советов Advice Language, 532; 543
удовлетворения ограничений, 305
Япония, 18

Научно-популярное издание

Иван Братко

Алгоритмы искусственного интеллекта на языке PROLOG, 3-е издание

Литературный редактор	<i>И.А. Попова</i>
Верстка	<i>А.Н. Полинчик</i>
Художественный редактор	<i>С.А. Чернокозинский</i>
Корректоры	<i>З.В. Александрова, Л.А. Гордиенко, О.В. Мишутина</i>

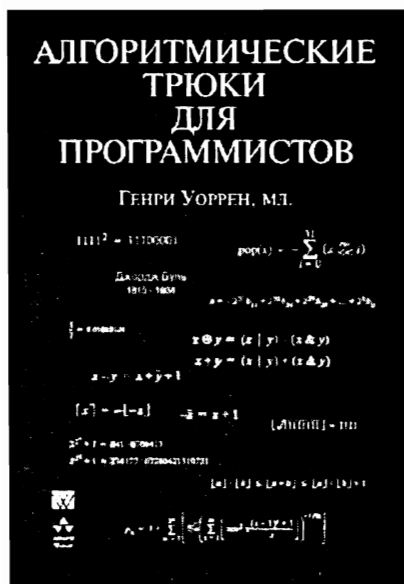
Издательский дом “Вильямс”.
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 12.07.2004. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 51,6. Уч.-изд. л. 41,6.
Тираж 3000 экз. Заказ № 180.

Отпечатано с диапозитивов в ФГУП “Печатный двор”
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

АЛГОРИТМИЧЕСКИЕ ТРЮКИ ДЛЯ ПРОГРАММИСТОВ

Генри С. Уоррен, мл.



www.williamspublishing.com

Эта книга, несмотря на ее название в оригинале *Hacker's Delight* не имеет никакого отношения к подбору паролей или взлому сетей. Здесь слову "хакер" возвращено его первоначальное значение человека увлеченного, талантливого программиста, способного к созданию чрезвычайно эффективного и элегантного кода. В любой области человеческой деятельности эффективность эстетична, и программирование не является исключением.

В этой книге вы познакомитесь с приемами работы с целочисленной арифметикой, которые доставят вам массу чисто эстетического наслаждения при этом попутно повысив вашу квалификацию.

В книге воплощен сорокалетний стаж ее автора в области разработки компиляторов и архитектуры компьютеров. Здесь вы найдете множество приемов для работы с отдельными битами, байтами, вычислений различных целочисленных функций – причем большей части материала сопутствует строгое математическое обоснование. Каким бы высоким не был ваш профессионализм – вы все равно найдете в этой книге что-то новое для себя или по крайней мере книга заставит вас посмотреть на уже знакомые вещи с другой стороны. Не в меньшей степени эта книга пригодна и для начинающего программиста, который может просто воспользоваться готовыми советами из книги, применяя их в своей повседневной практике.

в продаже