Мартин Эрвиг

Занимательные Алгоритмы

Чему нас учат истории знаменитых героев

Занимательные алгоритмы

Чему нас учат истории знаменитых героев

Once Upon an Algorithm

How Stories Explain Computing

Martin Erwig

The MIT Press
Cambridge, Massachusetts
London, England

Занимательные алгоритмы

Чему нас учат истории знаменитых героев

Мартин Эрвиг

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Э74
УДК 681.3.07

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *И.В. Берштейна*

Под редакцией канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Эрвиг, Мартин.

Э74 Занимательные алгоритмы: чему нас учат истории знаменитых героев. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 352 с. : ил. — Парал. тит. англ.

ISBN 978-5-907144-08-8 (рус.)

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 MIT Press.

Copyright © 2019 by Dialektika Computer Publishing Ltd.

Authorized Russian translation of the English edition of *Once Upon an Algorithm: How Stories Explain Computing* (ISBN 978-0-2620-3663-4), published by MIT Press © 2017 Massachusetts Institute of Technology.

This translation is published and sold by permission of MIT Press, which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

Научно-популярное издание

Мартин Эрвиг

Занимательные алгоритмы:
чему нас учат истории знаменитых героев

Подписано в печать 20.03.2019
Формат 70x100/16. Гарнитура Times
Усл. печ. л. 22,38. Уч.-изд. л. 28,5
Тираж 500 экз. Заказ № 2436

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-08-8 (рус.)
ISBN 978-0-2620-3663-4 (англ.)

© ООО "Диалектика", 2019
© Massachusetts Institute of Technology, 2017

Оглавление

Предисловие 11

Часть I. АЛГОРИТМЫ 27

Глава 1. Путь к пониманию вычислений 33

Глава 2. От слов к делу: когда действительно происходит вычисление 47

Глава 3. Тайна знаков 65

Глава 4. Записная книжка сыщика 81

Глава 5. Поиск идеальной структуры данных 103

Глава 6. Сортировка алгоритмов сортировки 127

Глава 7. Трудноразрешимые задачи 147

Часть II. ЯЗЫКИ 163

Глава 8. Сквозь призму языка 169

Глава 9. Поиск нужного тона: смысл звука 187

Глава 10. Намылить, смыть, повторить 203

Глава 11. Счастливый конец не гарантируется 219

Глава 12. Своевременный стежок вычисляется впрок 237

Глава 13. Все дело в интерпретации 259

Глава 14. Волшебный тип 279

Глава 15. С высоты птичьего полета: от абстракции к деталям 299

Приложение А. Словарь специальных терминов 323

Приложение Б. Примечания 339

Предметный указатель 349

Содержание

Предисловие **11**
Благодарности 14
Введение 15

Часть I. АЛГОРИТМЫ **27**
Вычисления и алгоритмы: ***Гензель и Гретель***
Глава 1. Путь к пониманию вычислений **33**
Разделение задач на более простые 34
Вычисление невозможно без представления 35
По ту сторону решения задачи 37
Когда задачи возникают снова 40
Говорите ли вы “по-алгоритмийски”? 41
Перечень требований 42

Глава 2. От слов к делу: когда действительно происходит вычисление **47**
Создание разнообразия 48
Кто выполняет алгоритм 50
Стоимость жизни 52
Общее представление о затратах 53
Рост затрат 55

Представление и структуры данных: ***Шерлок Холмс***
Глава 3. Тайна знаков **65**
Знаки представления 65
О знаках до самого основания 68
Осмысление означающих 70
Три способа обозначения 72
Систематическое применение представлений 73

Глава 4. Записная книжка сыщика **81**
Главные подозреваемые 82
Сбор сведений 89

Когда порядок имеет значение 91
Передается по наследству 93

Решения задач и их ограничения: ***Индиана Джонс***

Глава 5. Поиск идеальной структуры данных **103**
Ключ к быстрому поиску 104
Выживание в боггле 107
Словарный подсчет 110
Скудость — не всегда добродетель 112
Балансирующая эффективность 115
Префиксное дерево 118

Глава 6. Сортировка алгоритмов сортировки **127**
Обо всем по порядку 128
Разбивайте, не стесняясь 132
Лучшее еще впереди 136
Поиск завершен: лучших алгоритмов сортировки нет 138
Лучше не бывает 140
Сохранение вычислений 141

Глава 7. Трудноразрешимые задачи **147**
Перевешивание чаши весов 149
Когда время выполнения взрывается 151
Общая судьба 153
Обманчивость нирваны 156
Отходы — в доходы! 158

Часть II. ЯЗЫКИ **163**

Языки и смысловое значение: ***Над радугой***

Глава 8. Сквозь призму языка **169**
Замечания по поводу мелодии 171
Грамматические правила 174
Структура растет на деревьях 179

Глава 9. Поиск нужного тона: смысл звука **187**

Звучит неубедительно 188

Обретение смыслового значения 192

Управляющие структуры и циклы: *День Сурка*

Глава 10. Намылить, смыть, повторить **203**

Вечность и один день 204

Все под контролем 207

Цикл за циклом 212

Глава 11. Счастливый конец не гарантируется **219**

Неуправляемый полет 220

Мы все еще здесь? 222

Конца и краю не видно 224

Рекурсия: *Назад в будущее*

Глава 12. Своевременный стежок вычисляется впрок **237**

Все о времени 238

Не важно, когда 242

Точно в срок 243

Борьба с парадоксами с помощью фиксированных точек 248

Зацикливать или не зацикливать 252

Многоликая рекурсия 253

Глава 13. Все дело в интерпретации **259**

Переписывание истории 260

Уменьшение следов 263

Двойные двойники удваиваются 270

Типы и абстракция: *Гарри Поттер*

Глава 14. Волшебный тип **279**

Типы волшебства и волшебство типов 280

Правило для правил 285

Когда правила неприменимы 288

Обеспечение соблюдения законов 291

Построение кода 293

Глава 15. С высоты птичьего полета: от абстракции к деталям **299**
Как сделать длинный рассказ короче 300
Скажите, когда 304
Последовательный ряд абстракций 307
Один тип для всего 310
Время абстракции 312
Машинный язык 315
Дальнейшее исследование 321

Приложение А. Словарь специальных терминов **323**
А. Вычисления и алгоритмы 323
Б. Представления и структуры данных 325
В. Решения задач и их ограничения 327
Г. Язык и смысловое значение 331
Д. Управляющие структуры и циклы 333
Е. Рекурсия 335
Ж. Типы и абстракция 336

Приложение Б. Примечания **339**
Введение 339
Глава 1, “Путь к пониманию вычислений” 339
Глава 2, “От слов к делу: когда действительно происходит вычисление” 339
По дороге 340
Глава 3, “Тайна знаков” 340
Глава 4, “Записная книжка сыщика” 341
Потери и находки 341
Глава 5, “Поиск идеальной структуры данных” 341
Приведение дел в порядок 342
Глава 6, “Сортировка алгоритмов сортировки” 342
Глава 7, “Трудноразрешимые задачи” 343
Предписания врача 344
Глава 8, “Сквозь призму языка” 344
Глава 9, “Поиск нужного тона: смысл звука” 345
Глава 10, “Намылить, смыть, повторить” 345

И не останавливаться 346
Глава 11, “Счастливый конец не гарантируется” 346
Глава 12, “Своевременный стежок вычисляется впрок” 347
Состояние дел 347
Глава 13, “Все дело в интерпретации” 347
Глава 14, “Волшебный тип” 348
Глава 15, “С высоты птичьего полета: от абстракции к деталям” 348

Предметный указатель 349

Предисловие

Когда меня спрашивают о моей работе, разговор быстро переходит к вопросу о том, что такое *информатика* или *компьютерные науки*. Назвать информатику наукой о вычислительных машинах означало бы ввести собеседника в заблуждение, хотя, строго говоря, такое определение нельзя назвать неверным. Но ведь большинство людей понимают под *вычислительными машинами* или *компьютерами* персональные или переносные компьютеры, из чего делают вывод, что специалисты по информатике занимаются конструированием вычислительной аппаратуры. А с другой стороны, назвать компьютерные науки изучающими вычисления означало бы дать им поверхностное определение, поскольку тут же возник бы вопрос о том, что же такое вычисление.

С годами я пришел к выводу, что обучать, вводя одно понятие за другим, — не лучший способ, поскольку это слишком абстрактно. Ныне я, как правило, начинаю описание информатики как науки, изучающей систематическое решение задач. Всем известно, что такое задача, и все понимают, что такое решение. Пояснив представление об информатике на конкретном примере, я нередко пользуюсь удобным случаем ввести понятие алгоритма, что дает мне возможность указать на важные отличия информатики от математики. И зачастую мне не нужно даже вести речь о языках программирования, компьютерах и связанных с ними технических вопросах. Но если дело доходит до этого, то пояснить все подобные понятия совсем нетрудно на примере конкретной задачи. Именно такой подход и развит в этой книге.

Информатика является относительно новым членом клуба наук, и иногда даже кажется, что она все еще не заслужила уважительного отношения со стороны таких серьезных научных дисциплин, как физика, химия или биология. Если взять, к примеру, сцену из фильма о физиках, то можно представить себе ученого-теоретика, обсуждающего сложные формулы, написанные мелом на доске, или практика в лабораторном халате, наблюдающего за ходом эксперимента. Показанный в фильме физик — уважаемый ученый, обладающий драгоценными знаниями. А теперь представьте себе сцену со специалистом по информатике или программированию. В ней вы, вероятнее всего, обнаружите какого-нибудь занудного типа, сидящего в темной неубранной комнате и уставившегося в экран

компьютерного монитора. Он лихорадочно набирает что-то на клавиатуре, вероятно, пытаясь взломать какой-нибудь код или пароль. В обеих сценах решается важная задача, но если физик может дать сколько-нибудь понятное объяснение, что это за задача, зачем она нужна и как ее можно решить, то решение вычислительной задачи зачастую остается тайным, загадочным и обычно слишком сложным, чтобы объяснить его неспециалисту. Но если информатика необъяснима простым смертным, то зачем вообще пытаться узнать о ней больше или понять ее?

Предметом информатики является *вычисление*[1] — явление, оказывающее влияние на каждого. Речь не идет только о мобильных телефонах, переносных компьютерах или Интернете. Складывание бумажного самолетика, поездка на машине на работу, приготовление пищи и даже транскрипция ДНК (процесс, происходящий в клетках вашего организма миллионы раз за то время, пока вы изучаете данную научную дисциплину) — все это примеры вычислений в широком смысле слова, т.е. систематического решения конкретных задач, даже если бы большинство людей не восприняли их именно так.

Наука дает общее представление о мире природы и предоставляет научный метод для надежного упрочения этого представления в виде знания. И то, что относится ко всей науке в целом, касается и информатики в частности, особенно если учесть, что вычисления встречаются в самых разных формах и ситуациях. Следовательно, общее представление о вычислениях дает такие же преимущества для общего представления об окружающем мире и более эффективного решения многих практических задач, как и знание основ физики, химии и биологии. Эта сторона вычислений обычно называется *вычислительным мышлением*.

Главная цель этой книги — подчеркнуть общий характер вычислений, а следовательно, широкую применимость компьютерных наук. Автор надеется, что эта книга возбудит в вас, читатель, более широкий интерес к информатике и желание узнать о ней побольше.

Сначала в этой книге выявляется наличие вычислений в повседневной деятельности, а затем соответствующие понятия компьютерных наук поясняются на примерах известных сказок. Решения повседневных задач берутся из распорядка

[1] В контексте книги перевод английского термина *computation* следует рассматривать не только как привычные математические *вычисления*. Словарь дает следующее определение этого термина: "1. Применение последовательности операций к значению для получения другого значения. 2. Процесс обработки данных с преобразованием входных значений (сигналов) в выходные, причем эта концепция не ограничивается данными, представленными в цифровой или иной электронной форме: биологические, химические, оптические, гидравлические и даже часовые механизмы — все они выполняют вычисления определенного вида". Именно в этой расширенной трактовке и следует воспринимать термин *вычисления* в данной книге. — *Примеч. ред.*

типичного рабочего дня: пробуждения утром, завтрака, поездки на работу, событий на рабочем месте, приема у врача, занятий любимым хобби после полудня, ужина и размышлений над событиями прошедшего дня вечером. Небольшие истории обо всем этом представлены в отдельных пятнадцати главах книги, где понятия вычислений поясняются также на примерах семи известных историй. Каждая история охватывает две или три главы и относится к конкретному вопросу информатики.

Эта книга делится на две части, посвященные *алгоритмам* и *языкам*. Это главные опоры, на которых покоится понятие вычисления. В табл. 1 перечислены все истории и понятия информатики, которые они иллюстрируют.

Все мы отдаем должное интересной истории. Такие истории нас утешают, дают нам надежду и вдохновляют нас. Они рассказывают нам о мире, позволяют осознать задачи, с которыми нам приходится иметь дело, а иногда и предлагают их решения. Истории могут также служить руководством в нашей жизни. Если поразмыслить над тем, чему истории должны нас научить, то в нашем уме, вероятно, возникнет мысль о любви, конфликте, судьбе человека. Но в моем уме возникает также мысль о вычислениях. Когда Джульетта спрашивает "Что значит имя?[2]", она задает очень важный вопрос о представлении информации. В своем философском эссе "*Миф о Сизифе*" Альбер Камю поднимает вопрос не только о том, как бороться с абсурдностью жизни, но и как выявить бесконечное вычисление.

Таблица 1. Истории и темы книги

История	*Главы*	*Тема*
Часть I		
Гензель и Гретель	1, 2	Вычисления и алгоритмы
Шерлок Холмс	3, 4	Представления и структуры данных
Индиана Джонс	5, 6, 7	Решение задачи и его ограничения
Часть II		
Над радугой	8, 9	Язык и смысловое значение
День сурка	10, 11	Управляющие структуры и циклы
Назад в будущее	12, 13	Рекурсия
Гарри Поттер	14, 15	Типы и абстракция

У историй многоуровневое смысловое значение, и зачастую они содержат (иногда скрытый) вычислительный уровень. В этой книге предпринята попытка раскрыть данный уровень и предложить читателям взглянуть на известные истории по-новому с вычислительной точки зрения. Надеюсь, что читатели по достоинству оценят вычислительное содержимое представленных в этой книге историй, и эта новая точка зрения возбудит в них интерес к компьютерным наукам.

---

[2] Шекспир, "Ромео и Джульетта", пер. Т. Щепкина-Куперник.

Благодарности

Мысль написать эту книгу возникла в ходе многих бесед с друзьями, студентами, коллегами и даже с людьми в автобусе по дороге на работу. Благодарю всех моих собеседников за то терпение, с которым они слушали мои пояснения сущности компьютерных наук, а также за их дружеское нетерпение, когда мои пояснения становились слишком пространными и сложными. Цель написания этой общедоступной книги об информатике была в значительной степени подкреплена опытом этих бесед.

За последнее десятилетие у меня была возможность работать со студентами многих вузов во время их летней практики, что послужило дополнительным стимулом для написания этой книги. Эти летние практики были поддержаны несколькими грантами от Национального научного фонда (National Science Foundation), и я благодарен за такую поддержку научных исследований и обучения наукам в Соединенных Штатах.

В поисках материала для этой книги я полагался, в частности, на Интернет, Википедию (`wikipedia.org`) и веб-сайт TV Tropes (`tvtropes.org`). Благодарю авторов всех этих материалов за их преданность делу и неуемное желание делиться своими знаниями с остальным миром.

В процессе написания этой книги отдельные главы прочитали Эрик Уолкингшоу (Eric Walkingshaw), Пол Калл (Paul Cull) и Карл Шмельцер (Karl Smeltzer), которые дали профессиональные отзывы на прочитанное содержимое и стиль его изложения. Благодарю их за полезные советы, а также Дженнифер Пархем-Мочелло (Jennifer Parham-Mocello) не только за чтение отдельных глав, но и за проверку нескольких примеров вместе с ее учащимися первого курса колледжа. Благодарю также моего сына Александра за корректуру рукописи этой книги и дельные советы по вопросам, связанным с историей Гарри Поттера. Бóльшая часть этой книги была написана во время, свободное от лекций в Университете штата Орегон. Благодарю руководство моей кафедры и университета за поддержку данного проекта.

Воплотить идею этой книги в жизнь оказалось намного труднее, чем я предполагал. Выражаю искреннюю признательность Мари Лафкин-Ли (Marie Lufkin-Lee), Катрин Альмейде (Katherine Almeida), Кэтлин Хенсли (Kathleen Hensley) и Кристин Саваж (Christine Savage) из издательства MIT Press за поддержку данного проекта и помощь, оказанную мне в ходе работы над ним.

И наконец, мне выпало счастье быть супругом самого терпеливого и откровенного читателя. Поэтому благодарю мою жену, Аню, за моральную поддержку на протяжении всего времени написания этой книги. Она всегда живо откликалась на мои вопросы, которые чаще всего были довольно нудными и абстрактными. Она прочитала немало черновиков и терпеливо старалась направлять мой стиль письма в нужное русло, когда он становился слишком академическим и чрезмерно испещренным техническим жаргоном. Завершением работы над этой

книгой я обязан в большей степени своей жене, чем кому-нибудь еще, поэтому я и посвящаю данную книгу ей.

Введение

Сегодня вычисления играют заметную роль в обществе. Но если только вы не стремитесь стать специалистом по информатике, то зачем вам эта книга и зачем что-то знать об этой науке? Ведь вы можете просто пользоваться с выгодой для себя технологиями, приводимыми в действие вычислениями. Вряд ли вам нужно разбираться в бортовом электронном оборудовании, чтобы воспользоваться самолетом, или стать дипломированным врачом, чтобы извлекать пользу из достижений современного здравоохранения.

Мир, в котором мы живем, состоит не только из изобретенных человеком технологий. Нам по-прежнему приходится взаимодействовать с неавтономными объектами, действующими по физическим законам. Поэтому понимать основы и механизмы действия этих объектов полезно просто для того, чтобы предвидеть их поведение и безопасно перемещаться в своей среде. То же самое можно сказать и о пользе от изучения как самих вычислений, так и связанных с ними понятий. Вычисления обнаруживаются не только в компьютерах и электронных устройствах, но и за пределами вычислительных машин. Ниже приводится краткое описание основных принципов информатики и поясняется их назначение.

Вычисление и алгоритмы

Предлагаю вам выполнить следующее упражнение. Для этого вам понадобятся линейка, карандаш и лист бумаги в клеточку. Проведите сначала горизонтальную линию длиной в 1 дм, а затем — вертикальную линию той же длины перпендикулярно первой от одного из ее концов. И наконец, соедините оба свободных конца проведенных линий диагональю, чтобы сформировать прямоугольный треугольник. А теперь измерьте длину проведенной диагонали. Примите мои поздравления: вы только что вычислили квадратный корень из 2, как показано на рис. 1.

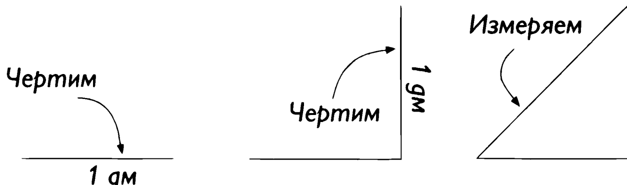


Рис. 1. Вычисление квадратного корня из 2 карандашом и линейкой

Но какое отношение это упражнение в геометрии имеет к вычислениям? Как поясняется в главах 1, “Путь к пониманию вычислений”, и 2, “От слов к делу: когда действительно происходит вычисление”, выполнение алгоритма на компьютере

приводит к вычислениям. В данном упражнении вы выступили в роли компьютера, выполнившего алгоритм черчения линий и измерения их длин, что привело к вычислению квадратного корня из 2. Наличие алгоритма имеет решающее значение, поскольку лишь в этом случае на разных компьютерах можно повторно выполнять вычисления в разное время. Важная особенность вычисления состоит в том, что для его проведения требуются ресурсы (например, карандаш, бумага и линейка) и время. И здесь очень важно описание алгоритма, поскольку оно помогает проанализировать требования к ресурсам для вычислений.

В главах 1 и 2 поясняется следующее:

- что такое алгоритмы;
- что алгоритмы применяются для систематического решения задач;
- что для проведения вычислений алгоритмы должны выполняться компьютером (человеком, машиной и т.д.);
- что для выполнения алгоритмов требуются определенные ресурсы.

Почему это так важно?

Примерами алгоритмов служат рецепты. Всякий раз, когда вы делаете сэндвич, печете шоколадный пирог или готовите свое любимое блюдо, следуя инструкциям рецепта, вы, по существу, выполняете алгоритм превращения исходных ингредиентов в конечный продукт. К числу ресурсов, которые для этого требуются, относятся ингредиенты, орудия труда, энергия и время приготовления.

Зная алгоритмы, мы можем точнее ответить на вопросы, касающиеся правильности выбранного метода и требований к его ресурсам. Такое знание помогает нам выявить возможности для улучшения процессов во всех областях жизни через (ре)организацию отдельных стадий и материалов. Например, при вычислении квадратного корня геометрическим способом можно было бы опустить рисование диагональной линии, просто измерив расстояние между двумя несоединенными конечными точками.

В приготовлении пищи улучшение может состоять в простой и очевидной экономии времени на посещение холодильника, если заранее предусмотреть и подготовить необходимые ингредиенты. Кроме того, вы можете заранее продумать, как эффективнее пользоваться своей печкой или плитой и сэкономить время на распараллеливании стадий процесса приготовления пищи, предварительно разогревая печку или плиту или промывая салат, пока готовится картофель. Аналогичные методики применяются и во многих других областях: от простых инструкций по сборке мебели до процессов организации работы конторы или производственного цеха.

В области технологий алгоритмы, по существу, управляют всеми вычислениями на компьютерах. Характерным тому примером служит сжатие данных, без которого передавать музыку и фильмы через Интернет было бы практически

невозможно. Алгоритм сжатия выявляет в данных часто встречающиеся образцы и заменяет их небольшими кодами. Сжатие данных позволяет решить задачу ресурсов для вычислений, сокращая память, требующуюся для представления музыкальных композиций и фильмов, а следовательно, и время, затрачиваемое на их загрузку через Интернет. Еще одним характерным примером служит принятый в компании Google алгоритм составления рейтинга веб-страниц, определяющий порядок представления полученных результатов пользователю. По этому алгоритму важность веб-страницы оценивается путем подсчета количества ссылок на нее с учетом веса, обозначающего важность этих ссылок.

Представление и структуры данных

Следует ожидать, что числовые вычисления выполняются с помощью чисел, поскольку мы пользуемся индо-арабской системой счисления, а вычислительные машины оперируют нулями и единицами. Таким образом, геометрический способ вычисления квадратного корня из 2 может показаться необычным. Но данный пример вычисления наглядно демонстрирует, что одно и то же понятие (например, количество) можно представить разными способами (числовыми знаками или отрезками).

Сущность вычисления состоит в преобразовании представления. В главе 3, "Тайна знаков", поясняется, что такое представления и как они употребляются в вычислениях. Многие вычисления оперируют большими объемами информации, и поэтому в главе 4, "Записная книжка сыщика", поясняется, как эффективно организовать коллекции данных. Этот вопрос усложняется тем, что в любой конкретной организации данных одни формы эффективного доступа к данным могут поддерживаться, а другие формы — не поддерживаться.

В главах 3 и 4 обсуждаются следующие вопросы:

- формы представления;
- способы организации коллекций данных и доступа к ним;
- преимущества и недостатки видов организации данных.

Почему это так важно

Ингредиенты в рецептах могут измеряться по весу или объему. Это разные формы представления, требующие разной кухонной утвари (весов или мерных чашек) для правильного исполнения рецепта или алгоритма. Что же касается организации данных, то способ хранения продуктов в холодильнике или кладовой оказывает влияние на быстроту извлечения ингредиентов, требующихся по рецепту. С другой стороны, можно рассмотреть вопрос представления применительно к самому рецепту. В частности, рецепт может быть представлен в виде последовательного ряда рисунков или даже в форме видеозаписи, выложенной в YouTube. Эффективность алгоритмов нередко зависит от выбора способа представления.

С одной стороны, вопрос организации коллекции элементов или людей находит немало примеров применения, включая наведение порядка на рабочем столе или в гараже, помогающее быстрее находить нужные вещи, или организация книжных полок в библиотеке. С другой стороны, можно рассмотреть разные случаи ожидания в очереди: в продовольственном магазине (стоя в очереди), приемной врача (сидя в ожидании вызова вашего номера) или же при посадке на самолет (несколько очередей).

В области информатики к самым удачным инструментальным средствам программирования относятся электронные таблицы. Этой удачей они во многом обязаны организации данных в табличной форме, поскольку она поддерживает быстрое и простое формулирование сумм по строкам и столбцам, а также представляет данные и результаты вычислений в одном месте. С другой стороны, Интернет является одним из самых преобразовывающих изобретений конца XX века, позволивших организовать веб-страницы, компьютеры и соединения между ними в сеть. Такое представление поддерживает гибкий доступ к информации и эффективную передачу данных.

Решения задач и их ограничения

Алгоритм — это средство решения задач, будь то извлечение квадратного корня из числа или выпекание пирога. Информатика — это научная дисциплина, занимающаяся систематическим решением задач.

Среди многих задач, которые могут быть решены с помощью алгоритмов, подробного рассмотрения заслуживают две задачи. В главе 5, "Поиск идеальной структуры данных", поясняется задача поиска — один из наиболее часто выполняемых видов вычислений данных, а в главе 6, "Сортировка алгоритмов сортировки" — задача сортировки (также в ней демонстрируется эффективный способ ее решения и рассматриваются трудности, присущие этой задаче). В главе 7, "Трудноразрешимые задачи", описывается класс так называемых трудноразрешимых задач. Алгоритмы существуют и для этих задач, но для их выполнения требуется слишком много времени, и поэтому на практике их можно считать неразрешимыми.

В главах 5–7 проясняются следующие вопросы:

- причины, по которым поиск может быть трудным и долгим;
- методы усовершенствования поиска;
- алгоритмы сортировки;
- что в вычислениях одного вида могут поддерживаться другие виды, например сортировка может поддерживать поиск;
- что алгоритмы со временем выполнения, изменяющимся по экспоненциальному закону, на самом деле считаться решениями задач не могут.

Почему это так важно

Мы тратим бесчисленные часы своей жизни на поиски, будь то ключи от автомашины или сведения в Интернете. Поэтому полезно разбираться в поиске и знать методики, способные помочь сделать его более эффективным. Кроме того, задача поиска демонстрирует, каким образом выбор способа представления оказывает влияние на эффективность алгоритмов, отражая следующее наблюдение Джона Дьюи (John Dewey): "Правильно поставленная задача означает половину ее решения" [1][3].

Знать, когда задачу *нельзя* решить эффективно, так же важно, как и знать алгоритмы для решаемых задач, поскольку это помогает нам не заниматься поиском эффективных решений там, где их не существует. Это предполагает, что в некоторых случаях мы должны довольствоваться приблизительными ответами.

В области информатики наиболее очевидным примером поиска служат механизмы поиска в Интернете, например в Google. Результаты поиска по запросу представлены не в произвольном порядке, а. как правило, отсортированными по их предполагаемой важности или релевантности. Знание трудности задач используется при разработке алгоритмов, вычисляющих приблизительные решения там, где вычисление точных решений отнимает слишком много времени. Одним из известных примеров служит задача коммивояжера, которая состоит в определении кругового маршрута для посещения определенного количества городов, который минимизирует общее проиденное расстояние.

Знанием об отсутствии эффективного алгоритма решения задачи можно эффективно воспользоваться и положительным образом. Одним из известных тому примеров служит шифрование с открытым ключом, обеспечивающее закрытые транзакции в Интернете, включая ведение банковских счетов и совершение закупок в оперативном режиме. Но такое шифрование действует лишь потому, что в настоящее время не существует эффективного алгоритма, обеспечивающего разложение большого числа на простые сомножители (т.е. дающего представление числа в виде произведения простых чисел). Если это положение изменится, то шифрование с открытым ключом станет ненадежным.

Языки: смысловое значение

Любой алгоритм может быть выражен на каком-нибудь языке. Современные компьютеры нельзя запрограммировать на человеческом языке, поскольку естественные языки содержат слишком много неоднозначностей, с которыми могут легко справиться люди, но не компьютеры. Следовательно, алгоритмы, которые предполагается выполнять на вычислительных машинах, должны быть написаны на языках, имеющих вполне определенную структуру и смысловое значение.

---

[3] Здесь следует упомянуть еще одну мысль, на этот раз — фантаста Роберта Шекли из рассказа *"Верный вопрос"*: "Чтобы правильно задать вопрос, нужно знать бо́льшую часть ответа". — *Примеч. ред.*

В главе 8, “Сквозь призму языка”, поясняется, что такое язык и как мы можем определить его синтаксис. Благодаря определению синтаксиса языка обеспечивается вполне определенная структура каждого из его предложений, и на этом основании удается понять и определить смысловое значение предложений и языков. А в главе 9, “Поиск нужного тона: смысл звука”, обсуждаются смысловое значение языков и проблема неоднозначности.

В главах 8 и 9 описывается:

- каким образом грамматика определяет язык;
- как с помощью грамматики построить все предложения, принадлежащие языку;
- что такое синтаксические деревья;
- каким образом синтаксические деревья представляют структуру предложений и разрешают неоднозначность в смысловом значении предложений.

Почему это так важно

Мы употребляем языки для передачи смыслового значения. А для того чтобы такая передача работала, общающиеся стороны должны прийти к согласию относительно того, что следует считать истинным предложением и что означает каждое предложение. Например, инструкции в рецептах должны быть точны в отношении мер измерений продуктов, температуры в печке, времени приготовления и так далее, чтобы получить желаемые результаты, а не нечто несъедобное и обгорелое.

В большинстве областей нашей жизни мы выработали специальную терминологию и языки, облегчающие общение и делающие его более эффективным. Это особенно справедливо в отношении информатики, где наиболее важная часть общения происходит через вычислительные машины. А поскольку вычислительные машины обладают более скромными способностями в смысле обработки естественных языков, чем люди, то очень важно дать точное определение языка, чтобы обеспечить желаемое поведение запрограммированных вычислительных машин.

В области информатики широко распространен такой язык программирования, как язык формул электронных таблиц. Всякий, кому доводилось когда-нибудь вводить формулу в электронную таблицу, уже написал программу для электронных таблиц. Но электронные таблицы печально известны тем, что они порой действуют ошибочно и это приводит к убыткам в миллиарды долларов из-за неточных формул. Еще одним распространенным языком является HTML (язык разметки гипертекста). Всякий раз, когда вы загружаете веб-страницу на свой персональный или переносной компьютер или же сотовый телефон, то, вероятнее всего, содержимое этой страницы представлено в вашем браузере на языке HTML, ясно выражающем структуру веб-страницы и однозначно ее

представляющем. И если язык HTML служит лишь для представления информации и сам не описывает вычисления, то динамическое поведение веб-страниц определяется на JavaScript — другом языке, понятном любому современному браузеру и созданном специально для возможности динамического поведения веб-страниц.

Управляющие структуры и циклы

Инструкции выполняют в алгоритме две разные функции: непосредственно манипулируют данными и решают, какие именно инструкции должны быть выполнены следующими и как часто это должно происходить. Последняя разновидность инструкций называется управляющими структурами. Подобно тому, как сюжет фильма или рассказа связывает отдельные действия и сцены в слаженное повествование, управляющие структуры составляют алгоритмы из отдельных инструкций.

В главе 10, "Намылить, смыть, повторить", поясняются различные управляющие структуры с особым акцентом на циклы, с помощью которых выражается повторение действий. А в главе 11, "Счастливый конец не гарантируется", обсуждается важный вопрос: завершается ли цикл или он выполняется бесконечно и можно ли выяснить это с помощью алгоритма.

В главах 10 и 11 обсуждаются следующие вопросы:

- что такое управляющие структуры;
- почему управляющие структуры играют решающую роль в любом языке для выражения алгоритмов;
- каким образом повторяющиеся задачи могут быть выражены с помощью циклов;
- что такое задача останова и каким образом она иллюстрирует основополагающее свойство вычислений.

Почему это так важно

Прежде чем печь блины, нужно смазать маслом сковородку. Порядок следования стадий приготовления пищи в рецептах имеет особое значение. Более того, рецепты иногда содержат решения, которые принимаются, исходя из свойств имеющихся ингредиентов или употребляемой кухонной утвари. Так, если вы пользуетесь конвекционной печью, то должны сократить время или понизить температуру выпекания или же сделать и то, и другое. Рецепт содержит цикл, если он предписывает выполнять повторяющиеся действия, например добавлять яйцо и взбивать смесь ингредиентов для приготовления кекса.

Отличие управляющих структур от других операций, по существу, означает различие между каким-то действием и организацией порядка и частоты его выполнения. Для выполнения любого процесса или алгоритма требуется знать,

делает ли он именно то, что должен делать, а еще проще — завершится ли он вообще. Этот довольно простой вопрос, который ставит проблема останова, служит лишь одним из примеров многих свойств алгоритмов, о которых хотелось бы знать. Зная, какие именно свойства одних алгоритмов могут быть автоматически определены с помощью других алгоритмов, можно выяснить пределы их досягаемости и ограничения вычислений.

В технологических областях управляющие структуры применяются там же, где и алгоритмы, т.е. везде. Любая информация, посылаемая через Интернет, передается циклически, повторяясь до тех пор, пока не будет правильно принята. Светофор работает под управлением бесконечно повторяющихся циклов, а многие производственные процессы состоят из заданий, которые повторяются до тех пор, пока не будет достигнута требуемая мера качества. Прогнозирование поведения алгоритмов в отношении неизвестных данных, которые могут быть введены впоследствии, имеет немало приложений в области безопасности. Например, хотелось бы заранее знать, уязвима ли система к атакам злоумышленников. То же самое применимо и к роботам-спасателям, которые должны использоваться в ситуациях, отличающихся о тех, в которых они обучены. Точность прогнозирования поведения робота в неизвестных ситуациях может, по существу, означать жизнь или смерть.

Рекурсия

Принцип сведения, т.е. процесс объяснения или построения системы из более простых частей, играет важную роль в большей части науки и техники. Рекурсия — это особая форма сведения, обращающегося к самому себе. Многие алгоритмы рекурсивны. Примером тому служат следующие инструкции для поиска слова в словаре, где на каждую статью приходится одна страница: "Открыть словарь. Если слово найдено, прекратить его поиск. В противном случае искать слово в части словаря, предшествующей или следующей после текущей страницы". Обратите внимание на то, что инструкция для поиска в последнем предложении означает рекурсивную ссылку на весь процесс, который возвращает к началу инструкций. Это избавляет от необходимости вводить в описание нечто вроде "повторять до тех пор, пока слово не будет найдено".

В главе 12, "Своевременный стежок вычисляется впрок", поясняется рекурсия, которая представляет собой управляющую структуру, которая применяется также для определения организации данных. А в главе 13, "Все дело в интерпретации", представлены разные подходы к пониманию рекурсии.

В главах 12 и 13 исследуется следующее:

- понятие рекурсии;
- различие между разными формами рекурсии;

- два метода раскрытия и осмысления рекурсивных определений;
- каким образом эти методы помогают понять рекурсию и взаимосвязь между разными ее формами.

Почему это так важно

Инструкции "приправить по вкусу" можно дать следующее рекурсивное определение: "Попробовать блюдо. Если оно приятно на вкус, на этом остановиться. В противном случае добавить щепотку специй, а затем приправить по вкусу". Любое повторяющееся действие может быть описано рекурсивно с помощью повторяемого действия (в данном случае — "приправить по вкусу") в его определении и условия, когда следует остановиться.

Рекурсия является важным принципом для получения конечных описаний потенциально бесконечных данных и вычислений. В грамматике языка рекурсия упрощает бесконечное количество предложений, а рекурсивный алгоритм позволяет обрабатывать входные данные произвольного объема.

Рекурсия представляет собой общую управляющую структуру и механизм организации данных и поэтому является неотъемлемой частью многих программных систем. Впрочем, рекурсия находит и ряд других непосредственных применений. Например, эффект Дросте (Droste), означающий наличие в изображении его более мелкого варианта, может быть получен в результате образования петли обратной связи между сигналом (изображением) и получателем (фотокамерой). Петля обратной связи служит рекурсивным описанием повторяющегося эффекта. Фракталы являются самоподобными геометрическими рисунками, которые могут быть описаны с помощью рекурсивных уравнений. В природе примерами фракталов могут служить снежинки и кристаллы. Фракталы применяются также при анализе белков и структур ДНК. Кроме того, фракталы применяются в нанотехнологиях для проектирования самособирающихся наносхем. В настоящее время проводятся исследования самовоспроизводящихся машин с точки зрения их применимости для освоения космоса.

Типы и абстракция

Вычисления состоят в преобразовании представлений. Но не всякое преобразование применимо к каждому представлению. Так, хотя можно умножать числа, умножать отрезки нельзя. И аналогично можно вычислить длину линии или площадь прямоугольника, но для числа это не имеет смысла.

Представления и преобразования могут быть разделены на группы с целью упростить отличие целесообразных преобразований от нецелесообразных. Эти группы называются типами, а правила, определяющие допустимые сочетания преобразований и представлений, — правилами типизации. Типы и правила типизации поддерживают разработку алгоритмов. Так, если требуется вычислить

число, то следует выполнить операцию, производящую числа. А если требуется обработать список чисел, то придется выполнить операцию, принимающую в качестве входных данных список чисел.

В главе 14, "Волшебный тип", поясняется, что такое типы и как можно пользоваться ими для формулирования правил, описывающих закономерности вычислений. Такие правила могут быть использованы для обнаружения ошибок в алгоритмах. Истинный потенциал типов лежит в их способности пренебрегать подробными сведениями об отдельных объектах, а следовательно, правила могут быть сформулированы на более общем уровне. Процесс пренебрежения подробностями называется абстракцией, которой посвящена глава 15, "С высоты птичьего полета: от абстракции к деталям", в ней поясняется, почему абстракции принадлежит центральное место в информатике и как она применяется не только в типах, но и в алгоритмах и даже в языках и компьютерах.

В главах 14 и 15 обсуждаются следующие вопросы:

- что такое типы и правила типизации;
- каким образом они могут быть использованы для описания законов вычислений таким образом, чтобы это помогало выявлять ошибки в алгоритмах и составлять надежные алгоритмы;
- что типы и правила типизации являются лишь особым случаем более общей концепции понятия абстракции;
- что алгоритмы служат абстракцией вычисления;
- что типы служат абстракцией представлений;
- что трудности, возникающие во время выполнения, служат абстракцией времени исполнения.

Почему это так важно

Если по рецепту требуется открыть консервную банку с бобами, то вы удивитесь, если кто-нибудь попытается решить эту задачу ложкой. Ведь это нарушило бы правило типизации, согласно которому ложка вряд ли пригодна для решения данной задачи.

Типы и прочие абстракции для описания правил и процессов применяются повсеместно. Любая процедура, которая должна быть повторена, предполагает алгоритмическую абстракцию, т.е. описание, пренебрегающее излишними подробностями и заменяющее переменные части параметров. Рецепты также содержат алгоритмические абстракции. Так, во многих кулинарных книгах содержится раздел, описывающий основные приемы приготовления пищи, например снятие кожуры и очистка помидоров от семян, а в самих рецептах делается ссылка на этот раздел, когда в них указывается, сколько требуется помидоров,

очищенных от кожуры и семян. Кроме того, роли различных объектов, принимающих участие в такой абстракции, обобщаются типами, характеризующими требования к ним.

В сфере технологий существует множество примеров типов и абстракций. Примерами физических типов являются все разновидности электровилок и розеток; винтов, отверток и бит шуруповертов; замков и ключей. Разные формы предназначены для предотвращения неправильного сочетания инструментов. В программном обеспечении примеры типов можно найти в веб-формах, которые требуют ввода номера телефона или адреса электронной почты в определенном формате. Существует множество примеров дорогостоящих ошибок, которые были вызваны игнорированием типов. Например, в 1998 году NASA потеряло спутник Mars Climate Orbiter стоимостью 655 миллионов долларов из-за несовместимых представлений чисел. Это была ошибка типа, которую можно было бы предотвратить при применении системы типов. Наконец, понятие компьютера само является абстракцией людей, машин или других субъектов, которые способны выполнять алгоритмы.

Как читать эту книгу

На рис. 2 даны общее схематическое представление понятий, обсуждаемых в этой книге, а также взаимосвязи между ними. В главах 7, 11 и 13, показанных в темно-серых прямоугольниках на рис. 2, представлено более формальное описание этих понятий. Эти главы можно пропустить, поскольку их чтение необязательно для понимания материала остальной книги.

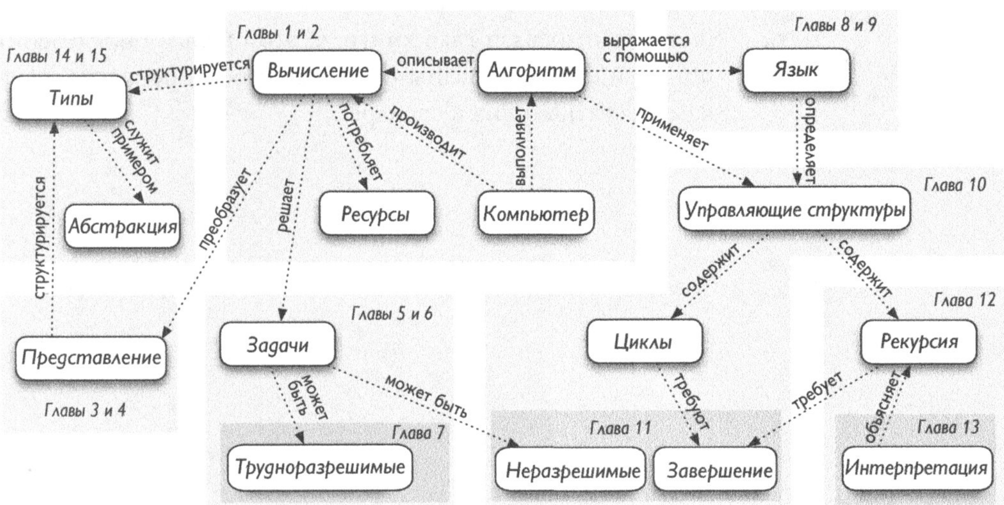


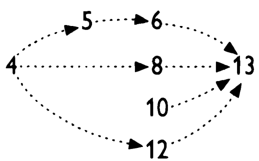
Рис. 2. Понятия вычисления и взаимосвязи между ними

Несмотря на то что материал этой книги упорядочен определенным образом, ее совсем необязательно читать именно в таком порядке. Многие ее главы можно читать независимо, даже если в последующих главах иногда делаются ссылки на понятия и примеры, представленные в предыдущих главах.

Ниже даются рекомендации по выбору глав и порядка их изучения, а также точки выхода и сокращения, дающие читателям возможность продвигаться вперед, читая избранные главы. Когда понятия вычисления обсуждаются в этой книге с использованием событий, людей и объектов, возникающих в историях, то иногда вводится новое обозначение и на конкретных примерах более подробно демонстрируются важные особенности. Поэтому одни части этой книги читать легче, чем другие. Как читатель многих научно-популярных книг я хорошо осознаю, что у читателей этой книги может быть разный интерес к таким подробностям. Именно поэтому я и надеюсь, что приведенные здесь рекомендации помогут читателям лучше ориентироваться в содержимом книги.

Сначала рекомендуется прочитать главы 1 и 2, поскольку в них представлены такие основополагающие понятия вычисления, упоминаемые далее в этой книге, как алгоритм, параметр, компьютер и временн*ая* сложность. Усвоить материал этих двух глав будет нетрудно.

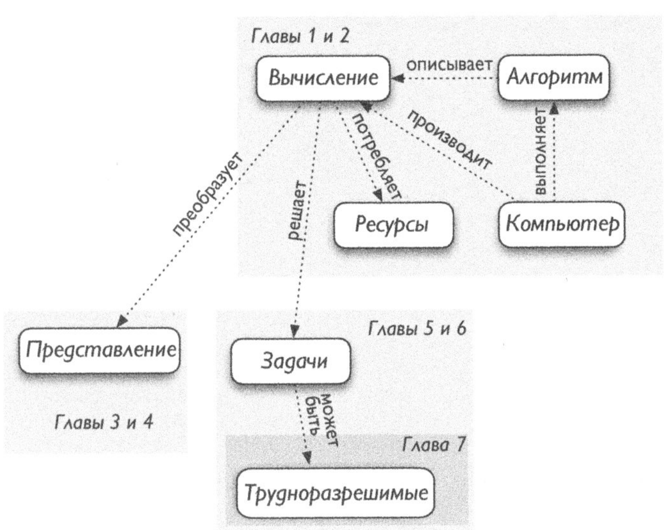
Остальные темы в светло-серых прямоугольниках на рис. 2 в основном не зависят одна от другой, но главы в пределах каждой темы следует читать по порядку. Так, в главе 4 представлен ряд структур данных, и поэтому ее следует прочитать прежде глав 5, 6, 8, 12 и 13, как показано на рисунке.



И наконец, в словаре специальных терминов, приведенном в конце книги, представлены дополнительные сведения о взаимосвязи глав книги. С этой целью определения понятий сгруппированы по темам и перекрестным ссылкам на статьи о них в словаре.

Часть I

АЛГОРИТМЫ



Вычисления и алгоритмы

Гензель и Гретель

Подъем

Раннее утро. Звенит будильник. Вы неохотно поднимаетесь с постели, окончательно проснувшись, и одеваетесь. Эта простая ежедневная процедура подъема по утрам решает повторяющуюся задачу через последовательный ряд вполне определенных шагов. В информатике такая процедура называется *алгоритмом*. Принятие душа, чистка зубов, завтрак и так далее служат примерами алгоритмов, решающих конкретные задачи или проблемы.

Но постойте. В чем здесь проблема, если не учитывать то, что вы, вероятно, не выспались? Мы обычно не воспринимаем обыденные повседневные виды деятельности как производящие решения задач. Возможно, это объясняется тем, что подобные задачи имеют очевидные или легко достигаемые решения. Но ведь термином *задача* обычно обозначаются те ситуации или вопросы, для которых имеются хорошо известные решения. Примером тому служат экзамены, состоящие из задач с вполне определенными решениями. Таким образом, *задача* — это любой вопрос или ситуация, требующая решения, даже если совершенно ясно, как его достичь. В этом смысле необходимость подниматься с постели по утрам является задачей, для решения которой имеется хорошо известный метод.

Зная, как решить задачу, мы редко задумываемся, каким образом был изобретен тот или иной метод ее решения. В частности, если такой метод очевиден и прост в применении, то, по-видимому, нет никакой нужды размышлять об этом. Но размышление о том, *как* можно решать задачи, способно помочь нам в будущем решать пока неизвестные задачи. Решения задач не всегда очевидны. Задним числом большинство решений кажется очевидным, но что если заранее неизвестно, как решить задачу подъема с постели по утрам? Как в таком случае подойти к ее решению?

Одно очень важное наблюдение состоит в том, что непростые задачи могут быть разложены на более простые подзадачи, а их решения объединены в единое решение первоначальной задачи. В частности, задача подъема по утрам состоит из двух подзадач: подъема с постели и одевания. У нас имеются следующие алгоритмы для решения обеих этих задач: перемещение нашего тела из постели и надевание одежды соответственно. И мы можем объединить эти алгоритмы в единый алгоритм подъема, но сделать это аккуратно и в правильном порядке. В самом деле одеваться в постели неудобно, поэтому мы должны сначала предпринять первый

шаг с целью выбраться из постели. Если данный пример покажется вам неубедительным, подумайте о том порядке, в каком следует принять душ и одеться. В данном простом примере имеется лишь одна последовательность шагов, приводящая ко вполне жизнеспособному решению, хотя так бывает не всегда.

Разложение задачи не ограничивается только одним уровнем. Например, задача одевания также может быть разложена на несколько подзадач, включая такие, как надевание штанов, рубашки, носков, обуви и т.д. Положительная сторона разложения задачи на подзадачи состоит в том, что оно помогает модуляризировать процесс поиска решений, а это означает, что решения разных подзадач могут быть выработаны независимо одно от другого. Модульность важна потому, что она позволяет распараллеливать выработку решений задач по отдельным командам.

Нахождение алгоритма для решения задачи еще не означает конец дела. Ведь алгоритм еще предстоит привести в действие, чтобы решить задачу фактически. Знать, как сделать что-нибудь, и сделать это — означает две разные вещи. Некоторые из нас болезненно осознают эту разницу каждое утро, когда звенит будильник. Следовательно, между алгоритмом и его применением имеется существенное различие.

В информатике каждое применение алгоритма называется *вычислением*. Но означает ли это, что, поднимаясь по утрам, мы вычисляем свой переход от постели к одежде? Это кажется нелепым, но что бы мы сказали о роботе, который сделал бы то же самое? Ведь робота нужно запрограммировать на выполнение такого задания. Иными словами, роботу необходимо сообщить алгоритм на понятном ему языке. Когда робот выполняет свою программу подъема с постели, то разве нельзя сказать, что он производит вычисление? Людей, конечно, нельзя называть роботами, но данный пример наглядно показывает, что люди вычисляют, выполняя алгоритмы.

Своей мощью алгоритмы обязаны тому обстоятельству, что они могут выполняться многократно. Как и пресловутое колесо, которое не ст*о*ит изобретать, однажды разработанный удачный алгоритм остается служить нам навсегда. Им можно многократно пользоваться в самых разных ситуациях, чтобы вычислять решения повторяющихся задач. Именно поэтому алгоритмы играют центральную роль в информатике, а их разработка относится к числу самых важных и увлекательных задач для специалистов в информатике.

Информатику можно было бы назвать наукой о решении задач. И хотя это определение вряд ли можно найти в многочисленной литературе, такое представление об информатике служит полезным напоминанием о причинах, по которым она оказывает влияние на все больше и больше областей нашей жизни. Более того, многие полезные вычисления происходят вне вычислительных машин и выполняются людьми, не подготовленными в информатике, для решения насущных задач. В главе 1, “Путь к пониманию вычислений”, представлено понятие вычисления и на примере истории Гензеля и Гретель освещаются вопросы, связанные с решением задач и человеческим фактором.

1 Путь к пониманию вычислений

Что же такое вычисление? Этот вопрос лежит в основе информатики. В этой главе дается (по крайней мере, предварительный) ответ, связывающий представление о вычислениях с некоторыми близко примыкающими к ним понятиями. В ней, в частности, поясняется взаимосвязь вычислений с понятиями задачи и алгоритма. С этой целью в ней описываются две взаимодополняющие особенности вычислений: что они делают и чем они являются.

На первый взгляд, *вычисления решают задачи*, и этим подчеркивается, что задачу можно решить посредством вычислений, если она надлежащим образом представлена и разложена на подзадачи. Такое представление о вычислениях не только отражает огромное влияние, которое информатика оказала на самые разные отрасли науки, но и объясняет, почему вычисления являются существенной частью всех видов деятельности человека независимо от применения вычислительных машин.

Тем не менее при взгляде на вычисление с точки зрения решения задач упускаются из виду некоторые их важные особенности вычисления. Более внимательный анализ отличий вычисления от решения задач приводит к другому представлению: *вычисление — это выполнение алгоритма*. Алгоритм дает точное определение вычислений и делает возможными их автоматизацию и анализ. Такое представление описывает вычисление как процесс, состоящий из нескольких стадий, что помогает объяснить, как и почему оно оказывается столь эффективным в решении задач.

Ключ к обузданию вычисления лежит в группировании сходных задач в один класс и разработке алгоритма, решающего каждую задачу из этого класса. Благодаря этому алгоритм становится похожим на умение. Так, умение печь пироги или чинить автомашину может потребоваться в разное время и может быть использовано неоднократно для решения разных примеров конкретного класса задач. Можно научиться умению и делиться им с другими, что только расширит сферу его влияния. Аналогично алгоритм можно выполнять повторно для решения разных примеров задач, формируя при каждом выполнении вычисление, решающее конкретную задачу.

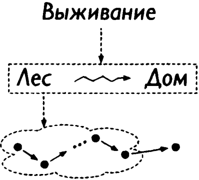
Разделение задач на более простые

Итак, начнем с первой точки зрения, рассматривая вычисление как процесс, в ходе которого решается конкретная задача. В качестве примера здесь выбрана известная сказка *Гензель и Гретель* братьев Гримм о детях, оставленных их родителями умирать в лесу. Исследуем находчивость Гензеля, благодаря которой ему и его сестре Гретель удалось найти обратный путь домой после того, как они были оставлены одни в лесу. Эта история раскрывается в контексте голода, когда мачеха Гензеля и Гретель побуждает их отца отвести детей в лес и бросить их там на произвол судьбы, чтобы дать возможность выжить самим родителям. Подслушав этот разговор родителей, Гензель выходит заполночь из дому и собирает несколько пригоршней камешков, набив ими свои карманы. На следующий день, когда они шли в лес, он бросал камешки вдоль дороги, отмечая тем самым путь из дома домой. После того как родители оставили детей одних в лесу, они подождали до темноты, когда камешки начнут сверкать в лунном свете, и, следуя по камешкам, вернулись домой.

И хотя на этом история не заканчивается, именно эта ее часть дает нам конкретный пример того, как решается задача с помощью вычисления. В данном случае решается задача выживания, т.е. намного более серьезная задача, чем утренний подъем. Задача выживания представляется как перемещение из одного места в лесу в другое место, где находится дом Гензеля и Гретель. Это непростая задача — особенно потому, что ее нельзя решить в один прием. Если задача оказывается слишком сложной, чтобы решить ее в один прием, она должна быть разбита на подзадачи, решить которые по отдельности легче, а затем объединить их решения в единое решение общей задачи.

Задача поиска пути из леса может быть разложена на подзадачи благодаря определению последовательности промежуточных мест, которые расположены настолько близко одно к другому, что между ними можно легко перемещаться. Эти местоположения образуют путь из леса обратно к дому Гензеля и Гретель, а отдельные перемещения из одного местоположения в другое достигаются без

труда. Если объединить все эти перемещения, то в конечном счете получится передвижение из исходного места в лесу домой. Именно это передвижение систематически решает задачу выживания Гензеля и Гретель. А ведь систематическое решение задачи является одной из главных характеристик вычисления.



Как демонстрируется в данном примере, вычисление обычно состоит не из одного, а из многих шагов. На каждом из этих шагов решается подзадача и немного изменяется состояние всей задачи в целом. Например, каждое передвижение Гензеля и Гретель к следующему камешку является этапом вычисления, на котором изменяется их местоположение в лесу, что соответствует решению подзадачи для достижения следующей цели на пути домой. И хотя в большинстве случаев каждый отдельный шаг приближает вычисление к окончательному решению, это совсем необязательно происходит на каждом шаге вычисления. Решение дают лишь все взятые вместе шаги вычисления. И хотя в данной истории каждое местоположение Гензеля и Гретель в общем приближает их к дому, вполне вероятно и то, что их путь домой не лежит на прямой линии. Некоторые камешки могут даже заставить их отклоняться в сторону, чтобы, например, обойти препятствия или пересечь реку по мосту, но это все равно не меняет конечный результат объединенного передвижения.

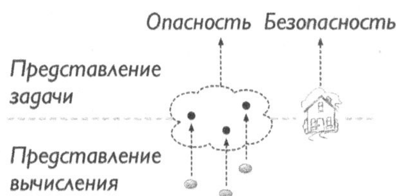
Из данной истории можно извлечь следующий важный урок: решение достигается посредством систематического разложения задачи на составляющие. Но, несмотря на то, что разложение на составляющие является главной стратегией для достижения общего решения задачи, одного только разложения не всегда оказывается достаточно, поскольку решения могут зависеть от дополнительных элементов (в истории Гензеля и Гретель — камешков).

Вычисление невозможно без представления

Если вычисление состоит из целого ряда этапов, то что же фактически делается на каждом из этих этапов и как на всех этих этапах вместе достигается решение данной задачи? Чтобы добиться совокупного результата, на каждом шаге должен быть достигнут такой результат, чтобы на его основании могли быть выполнены последующие этапы, а в конечном счете — достигнуто решение задачи как совокупный результат выполнения всех этапов. В рассматриваемой здесь истории на каждом этапе местоположение Гензеля и Гретель изменяется, а задача оказывается

решенной, когда их текущее местоположение оказывается их домом. В общем, отдельный шаг вычисления может оказывать влияние практически на все что угодно, будь то конкретные физические объекты или абстрактные математические сущности.

Чтобы решить задачу, необходимо, чтобы вычисление манипулировало *представлением* чего-то значимого в реальном мире. Местоположение Гензеля и Гретель представляет одно из двух возможных состояний: все места в лесу представляют состояние задачи, означающее опасность, а возможно, и смерть, тогда как дом — состояние решения, означающее безопасность и спасение. Именно поэтому вычисление, приводящее Гензеля и Гретель к их дому, решает задачу, поскольку оно продвигает от опасности к безопасности. А вычисление, направляющее их от одного места в лесу к другому, не достигнет этой цели.



В данном примере имеется еще один уровень представления. Поскольку вычисление, определяемое перемещениями между местами в лесу, выполняется Гензелем и Гретель, они должны распознавать эти места. Именно поэтому Гензель бросает камешки по пути в лес. Камешки представляют места в виде, позволяющем компьютеру (в данном случае его роль играют Гензель и Гретель) выполнять шаги вычисления. Как правило, у представления имеется несколько уровней. В данном случае имеется один уровень, определяющий задачу (местоположения в лесу), и еще один уровень, позволяющий вычислить решение (камешки). Кроме того, все камешки, взятые вместе, составляют еще один уровень представления, поскольку они представляют путь из леса обратно домой. Все эти уровни представления сведены в табл. 1.1.

Таблица 1.1. Уровни представления

Представление вычисления		*Представление задачи*	
Объект	**Представляет**	**Понятие**	**Представляет**
Один камешек	Место в лесу	Место в лесу	Опасность
	Дом	Дом	Безопасность
Все камешки	Путь из леса	Путь из леса	Решение задачи

На рис. 1.1 представлена общая картина вычисления для решения рассматриваемой здесь задачи: поиск Гензелем и Гретель пути домой показан как наглядный пример того, что вычисление манипулирует представлением с помощью последовательности шагов. В задаче подъема утром мы также находим представления, например, в виде места (в постели, вне постели) и в виде будильника, представляющего время. Представления могут принимать самые разные формы. Более подробно они рассматриваются в главе 3, "Тайна знаков".

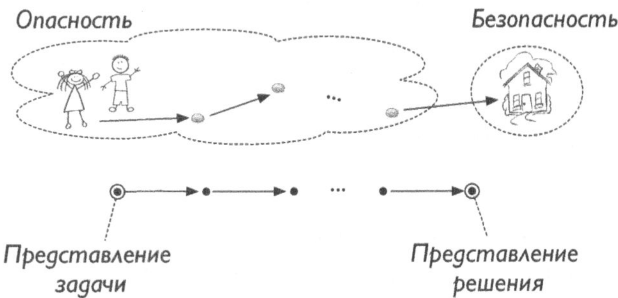


Рис. 1.1. Вычисление — это процесс решения конкретной задачи. Как правило, вычисление состоит из нескольких шагов. Начиная с представления задачи, каждый шаг преобразует ее представление до тех пор, пока не будет достигнуто решение. Гензель и Гретель решают задачу выживания с помощью процесса пошагового изменения их местоположения камешек за камешком по пути из леса домой

По ту сторону решения задачи

Рассмотрение вычисления как процесса решения задачи позволяет понять предназначение вычисления, но никак не объясняет, что же на самом деле представляет собой вычисление. Более того, у подобного представления о решении задачи имеются некоторые ограничения, поскольку не всякое действие по решению задачи является вычислением.

Как показано на рис. 1.2, имеются вычисления и имеются решения задач. И хотя эти множества зачастую пересекаются, некоторые вычисления не решают задачи, а некоторые задачи не решаются посредством вычислений. В данной книге основное внимание уделяется пересечению множеств вычислений и решений задач, но для того чтобы это стало понятным, рассмотрим несколько примеров двух других случаев.

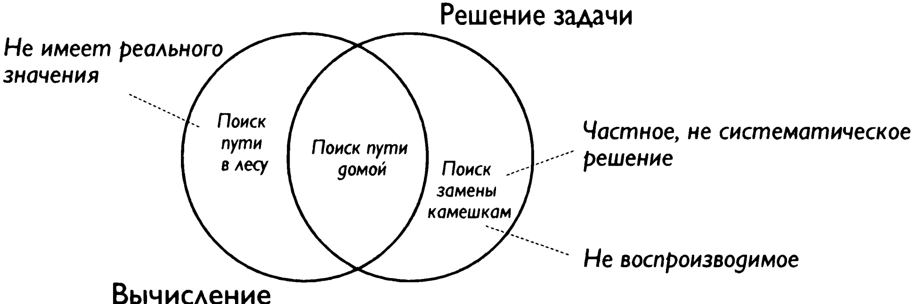


Рис. 1.2. Различение между решением задачи и вычислением. Вычисление, результат которого не имеет никакого смыслового значения в реальном мире, не решает ни одну из задач. А частное, не пригодное для повторения решение задачи не является вычислением

В первом случае допустим, что вычисление состоит в том, чтобы следовать по камешкам от одного места в лесу к другому. Этапы этого процесса в принципе такие же, как в и первоначальной истории, но соответствующая смена места не решит задачу выживания Гензеля и Гретель. В качестве еще более драматичного примера представьте ситуацию, когда камешки образуют замкнутую цепь. Это означает, что соответствующее вычисление, по-видимому, ничего не достигнет, поскольку исходное и конечное положения оказываются одинаковыми. Иными словами, вычисление не дает требуемого совокупного результата. Разница между этими случаями и тем, который имел место в рассматриваемой здесь истории, заключается в смысловом значении процессов.

Процессы без такого очевидного смыслового значения все же остаются вычислениями, но могут не являться решением задачи. Этот случай не так важен, поскольку можно всегда назначить какое-нибудь произвольное смысловое значение представлению, которым оперирует конкретное вычисление. Таким образом, любое вычисление можно было бы, несомненно, считать решением задачи, хотя сама задача зависит от того, какое именно смысловое значение связано с представлением. Например, следование по замкнутому кругу вряд ли помогло бы Гензелю и Гретель, но могло бы решить задачу разминки бегуна. Следовательно, способно ли вычисление решить задачу, зависит от точки зрения наблюдателя, т.е. от полезности самого вычисления. Но независимо от того, присвоит ли некто вычислению статус решения задачи, это не оказывает никакого влияния на саму суть вычисления.

Рассмотренный случай разительно отличается от решения невычислительной задачи, поскольку она предоставляет в наше распоряжение дополнительные критерии для определения самого вычисления. Два таких критерия приведены на рис. 1.2, причем оба они тесно связаны. Так, если задача решается некоторым особым способом, не следующим определенному методу, то это не вычисление. Иными словами, вычисление непременно должно быть систематическим.

Некоторые примеры решения такого рода невычислительной задачи можно найти и в рассматриваемой здесь истории. Одним из таких примеров служит ситуация, когда Гензель и Гретель становятся пленниками ведьмы, которая пытается откормить Гензеля, чтобы его съесть. Но поскольку она плохо видит, то оценивает вес Гензеля по толщине его пальца. Гензель вводит ведьму в заблуждение относительно своего веса, показывая ей мелкую косточку вместо своего пальца. Эта мысль не является результатом систематического вычисления, хотя и решает задачу, отсрочивая момент, когда ведьма решит, что пора съесть Гензеля.

Еще одним примером решения невычислительной задачи служит ситуация, которая возникает после первого возвращения Гензеля и Гретель домой. Родители решают снова отвести детей в лес на следующий день, но на этот раз мачеха закрывает двери на ночь, чтобы Гензель не смог набрать камешков. Проблема состоит в том, что Гензелю стали недоступны камешки, которые сослужили ему добрую службу в первый раз и на помощь которых он надеялся, чтобы найти обратный путь домой. Поэтому его решение состояло в том, чтобы найти замену камешкам в виде хлебных крошек. И здесь очень важно, каким образом он приходит к такому решению: у него возникает идея, творческая мысль. Решение, которое включает в себя момент озарения, как правило, очень трудно, если вообще возможно, получить через вычисление систематически, поскольку для этого требуется тонкое знание об объектах и их свойствах.

К несчастью для Гензеля и Гретель, решение употребить хлебные крошки не дало ожидаемого результата:

> *Когда взошла луна, они отправились в путь, но не нашли ни одной хлебной крошки, поскольку многие тысячи птиц, летающих по лесам и полям, подобрали все крошки до единой. [1]*

Хлебные крошки исчезли, и поэтому Гензель и Гретель не могут найти дорогу домой, и далее рассказывается остальная часть истории.

Но допустим на мгновение, что Гензелю и Гретель все же удалось каким-то образом найти снова обратный путь домой и что родители попытались в третий раз оставить их одних в лесу. В таком случае Гензелю и Гретель пришлось бы придумать еще оно средство, чтобы пометить обратный путь домой. Для этого им пришлось бы найти нечто другое, что можно было бы бросать по пути, а возможно, пытаться как-то пометить деревья и кусты. Каким бы ни было их решение, оно было бы получено в результате обдумывания задачи и выработки еще одной творческой мысли, хотя и не систематически применяемым методом. И здесь выявляется другой критерий для вычисления: его можно повторять для решения многих сходных задач. В этом отношении метод решения задачи поиска обратного пути домой по камешкам отличается, поскольку может выполняться повторно для самых разных мест нахождения камешков.

Таким образом, несмотря на то что взгляд на вычисление с точки зрения решения задач показывает, что вычисление является систематическим и разлагаемым на составляющие процессом, этого оказывается недостаточно, чтобы дать полное и точное представление о вычислении. Взгляд на вычисление как на решение задачи демонстрирует, каким образом можно применять вычисление в самых разных ситуациях, а следовательно, демонстрирует его особое значение, хотя и игнорирует некоторые важные свойства, поясняющие, каким образом вычисление работает и почему его можно успешно применять такими разными способами.

Когда задачи возникают снова

Перед Гензелем и Гретель задача поиска обратного пути домой вставала *дважды*. Если не считать практических трудностей, связанных с отсутствием камешков, во второй раз эта задача была решена таким же образом, как и в первый: следованием по последовательному ряду отметок. И в этом нет ничего удивительно, поскольку Гензель и Гретель просто применили общий метод поиска обратного пути. Такой метод называется *алгоритмом*.

Рассмотрим алгоритм, примененный Гензелем и Гретель для поиска обратного пути домой. Конкретный метод в исходном тексте сказки подробно не поясняется; там об этом говорится следующее:

> *И когда взошла полная луна, Гензель взял свою младшую сестру Гретель за руку и последовал по камешкам, которые сияли, как недавно отчеканенные серебряные монеты, показывая им путь.*

Простой алгоритм, соответствующий приведенному выше словесному образу, можно описать, например, так:

> Найти сверкающий камешек, не посещавшийся ранее, и направиться к нему. Продолжать этот процесс до тех пор, пока не будет достигнут родительский дом.

Важное свойство алгоритма состоит в том, что его может повторно использовать тот же самый или другой человек для решения той же самой или тесно связанной с ней задачи. Алгоритм, формирующий вычисление с физическим результатом, приносит пользу даже в том случае, если решает только одну конкретную задачу. Например, по рецепту для пирога можно неоднократно приготовить один и тот же пирог. Алгоритм дает преходящий результат (пирог съедается), и поэтому очень полезно воспроизводить один и тот же результат. То же самое относится и к задаче подъема с постели и одевания. Действие алгоритма, который ее решает, должно воспроизводиться каждый день, хотя и, вероятно, с разной одеждой и в другое время по выходным дням. То же самое относится и к Гензелю с Гретель.

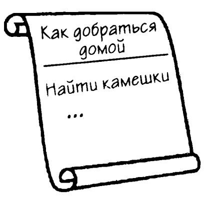
Даже если их приведут в то же самое место в лесу, что и в первый раз, обратный их путь домой повторно вычисляется по повторяющемуся алгоритму для решения совершенно той же задачи.

Совсем иначе обстоит дело с алгоритмами, дающими нефизические, абстрактные результаты, например числа. В таком случае результат можно просто записать, чтобы просмотреть его в следующий раз, когда в этом возникнет потребность, вместо того чтобы выполнять алгоритм снова. Чтобы алгоритм приносил пользу в подобных случаях, он должен быть в состоянии решать целый класс задач, а это означает, что соответствующий метод можно применять для решения самых разных, но взаимосвязанных задач. [2]

В рассматриваемой здесь истории применяется достаточно общий метод для решения самых разных задач поиска пути, поскольку конкретное местоположение камешков не имеет особого значения. Куда бы родители ни завели детей в лесу, алгоритм будет действовать в любом случае [3], а следовательно, приведет к вычислению, решающему задачу выживания Гензеля и Гретель. Сила и влияние алгоритмов объясняются в основном тем обстоятельством, что *один* алгоритм пригоден для *множества* вычислений.

Понятие алгоритма является одним из самых важных в информатике, поскольку служит основанием для систематического изучения особенностей вычисления. Именно поэтому в данной книге обсуждаются самые разные особенности алгоритмов.

Говорите ли вы “по-алгоритмийски”?



Алгоритм служит описанием порядка вычисления, а значит, должен быть сформулирован на каком-то языке. В рассматриваемой здесь истории алгоритм упоминается лишь вскользь. В уме Гензеля, конечно, имелся определенный алгоритм, и он мог бы сообщить его Гретель, но этот алгоритм не записан как часть повествования. Однако тот факт, что алгоритм может быть записан, является важным его свойством, поскольку обеспечивает надежный способ поделиться алгоритмом, а следовательно, дает возможность многим людям воспользоваться им для решения задач. Возможность выразить алгоритм на каком-либо языке способствует распространению вычислений, поскольку вместо одного человека, производящего многие вычисления, такая возможность помогает многим людям производить еще больше вычислений. Если язык, на котором выражен алгоритм, может быть понятен вычислительной машине, то распространение вычисления кажется практически безграничным, ограниченным лишь ресурсами, требующимися для построения компьютеров и работы с ними.

А требует ли вообще алгоритм подъема по утрам описания на каком-то языке? Вероятно, не требует. Благодаря неоднократному выполнению все мы усвоили

его шаги до такой степени, что выполняем их неосознанно, не требуя отдельного описания. Тем не менее описание отдельных частей данного алгоритма все же существует, зачастую в виде последовательности рисунков. Характерным тому примером служит процедура завязывания галстука или плетения волос в сложную косичку. Если выполнить одну из этих процедур в первый раз без посторонней помощи, то научиться ей можно по такому описанию.

Возможность выразить алгоритм на каком-нибудь языке имеет еще одно важное следствие. Она позволяет выполнять систематический анализ и формальное манипулирование алгоритмами, что является предметом теории вычислительных машин и систем и языков программирования.

Алгоритм для его выполнения должен быть выражен на языке, понятном компьютеру. Более того, описание алгоритма должно быть *конечным*, т.е. ограниченным, не длящимся бесконечно. И наконец, каждый отдельный шаг алгоритма должен быть *эффективным*. Это означает, что всякий, кто выполняет алгоритм, должен быть в состоянии понимать и выполнять все его шаги. Очевидно, что алгоритм Гензеля и Гретель конечен, поскольку содержит лишь несколько инструкций, а отдельные его шаги эффективны, по крайней мере в том случае, если допустить, что камешки расположены на видимом расстоянии один от другого. Требование всегда находить камешек, не посещавшийся прежде, можно поставить под сомнение, поскольку не так-то просто запомнить все обнаруженные прежде камешки. Впрочем, реализовать это требование совсем нетрудно, просто подбирая каждый камешек, как только он будет найден. Но это был бы уже совсем другой алгоритм. Кстати, такой алгоритм упростил бы Гензелю и Гретель поиск обратного пути домой на следующий день, поскольку Гензель сохранил бы у себя все камешки. Но стоит хотя бы немного изменить алгоритм, и это, увы, была бы уже совсем иная история, а не классическая сказка, поведанная нам братьями Гримм.

Перечень требований

Помимо определения характеристик, имеется ряд свойств, желательных для алгоритмов. Например, алгоритм должен всегда производить вычисление, которое *завершается* и дает *правильные результаты*. Гензель положил конечное число камешков, отмечающих обратный путь к родительскому дому, и поэтому выполнение описанного выше алгоритма завершится, ведь Гензель посетит каждый камешек не больше одного раза. Но, как ни странно, этот алгоритм может не дать правильного результата во всех возможных случаях, поскольку процесс его выполнения может застопориться.

Как указывалось ранее, в рассматриваемом здесь алгоритме не указывается, к какому именно камешку следует направиться. Если бы Гензеля и Гретель вели

в лес не по прямой линии, а, скажем, зигзагом, то вполне возможно, что с места нахождения одного камешка было бы видно несколько других камешков. К какому камешку следовало бы тогда подойти Гензелю и Гретель? Об этом алгоритме ничего не говорится. Так, если предположить, что все камешки расположены на видимом расстоянии один от другого, то может возникнуть ситуация, наглядно показанная на рис. 1.3.

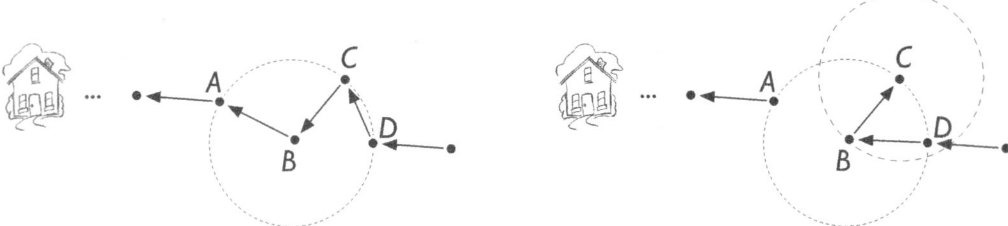


Рис. 1.3. Путь, демонстрирующий возможный тупик в алгоритме. Слева: обход камешков в обратном порядке приводит Гензеля и Гретель домой. Справа: все камешки B, C и D находятся на видимом расстоянии один от другого, и поэтому Гензель и Гретель могли бы выбрать путь сначала от камешка D к камешку B, а затем — к камешку C. Но тогда они застряли бы на месте, поскольку из камешка C не видно ни одного камешка, не посещавшегося ими прежде. В частности, они не смогли бы добраться до камешка A, расположенного следующим на их пути домой

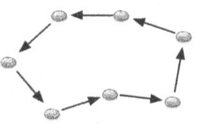
Допустим, что последовательный ряд камешков *A*, *B*, *C* и *D* размещается Гензелем на пути в лес. Допустим также, что камешек *A* можно видеть от камешка *B*, а от камешка *C* — камешек *B*, но камешек *A* находится слишком далеко, чтобы его было видно от камешка *C*, что показано кругами видимости, нарисованными вокруг камешков *B* и *C*. Кроме того, допустим, что камешек *D* находится на видимом расстоянии от камешков *B* и *C*. Это означает, что когда Гензель и Гретель дойдут до камешка *D*, то они смогут увидеть оба камешка *B* и *C*, а следовательно, им придется сделать выбор. Если они выберут камешек *C*, то обнаружат далее камешек *B* и, наконец, камешек *A*, и все будет замечательно (см. левую часть рис. 1.3). Но если Гензель и Гретель выберут камешек *B* вместо камешка *C* (что вполне возможно по рассматриваемому здесь алгоритму, поскольку *B* — это видимый, но еще не посещавшийся камешек), то они могут попасть в беду. Ведь если они выберут камешек *C*, который также виден и еще не посещался ими, то застрянут на этом камешке. Дело в том, что с места нахождения камешка *C* они могут видеть только два камешка, *B* и *D*, которые они уже посещали, а следовательно, эти камешки по данному алгоритму выбирать нельзя (см. правую часть рис. 1.3).

Безусловно, мы могли бы попытаться исправить данный алгоритм, введя инструкции для возврата в предыдущее положение в случаях, подобных описанному

выше, чтобы выбрать другой вариант, но цель рассматриваемого здесь примера — продемонстрировать случай, когда алгоритм не приводит к правильному результату. В этом примере также продемонстрировано, что поведение алгоритма не всегда удается легко предугадать, что делает разработку алгоритмов трудным, но увлекательным занятием.

Завершаемость алгоритмов — не так легко распознаваемое свойство. Если удалить из алгоритма условие поиска только не посещавшихся ранее камешков, то вычисление может легко скатиться к бесконечному перемещению между двумя камешками. Можно, конечно, возразить, что Гензель и Гретель никогда бы не совершили подобную глупость, распознав такой повторяющийся образец перемещения между камешками. Может, и так, но тогда они фактически не придерживались бы алгоритма в точности, намеренно избегая посещенного ранее камешка.

Несмотря на то что случай бесконечного перемещения между двумя камешками обнаружить нетрудно, рассматриваемая здесь задача в целом может намного усложниться. Допустим, что путь, выбранный родителями в лесу, несколько раз пересекает сам себя. В итоге расположение камешков будет содержать несколько замкнутых кругов, в каждый из которых могли бы попасть Гензель и Гретель. Они могли бы избежать замкнутых кругов, если бы запоминали посещенные ими камешки. Более подробно проблема завершаемости рассматривается в главе 11, “Счастливый конец не гарантируется”.



Вопросы правильности и завершаемости кажутся не столь важными для алгоритма подъема по утрам, но люди известны тем, что порой надевают разные носки или нарушают правила застегивания рубашки на пуговицы — вдевая пуговицы не в свои петли. А если разрешить многократное нажатие кнопки отбоя будильника, то алгоритм подъема по утрам никогда не завершится.

Начало дня

Для большинства людей день на самом деле не начинается до тех пор, пока они не позавтракают. Овсянка, фрукты, яичница с ветчиной, сок, кофе, т.е. все, что входит в меню на завтрак, вероятнее всего, потребует приготовления в той или иной форме. Некоторые из этих форм приготовления могут быть описаны алгоритмами.

Если вы предпочитаете варьировать меню завтрака, добавляя разные ингредиенты в овсянку или заваривая разные объемы кофе, то алгоритм, описывающий приготовление завтрака, должен отражать подобную гибкость меню. Ключ к обеспечению контролируемой изменчивости состоит в применении одного или более заполнителей, называемых *параметрами* и заменяемых конкретными значениями всякий раз, когда выполняется алгоритм. Благодаря применению разных значений заполнителей алгоритм производит разные вычисления. Например, вместо параметра "фрукты" можно подставить разные фрукты в разные дни недели, и в итоге алгоритм будет производить овсянку как с голубикой, так и с бананами. Алгоритм подъема по утрам также содержит параметры, чтобы не вынуждать нас вставать в одно и то же время и носить одну и ту же рубашку каждый день.

Если вы предпочитаете пить кофе в кофейне по дороге на работу или заказываете завтрак в ресторане, то и в этих случаях применяются алгоритмы по приготовлению вашего завтрака, только вместо вас это делают другие. Человек или автомат, выполняющий алгоритм, называется *вычислительной машиной* или просто *компьютером* и оказывает заметное влияние на результат вычисления. Вполне возможно, что компьютер не в состоянии выполнить алгоритм, если он не понимает язык, на котором тот задан, или же если он не способен выполнить один из шагов алгоритма. Представьте, что вы гостите на ферме и алгоритм получения молока по утрам включает дойку вами коровы. Такой шаг может оказаться невыполнимым.

Но даже если компьютер способен выполнить все шаги алгоритма, имеет значение затрачиваемое на них время. В частности, время выполнения алгоритма может заметно различаться на разных компьютерах. Например, опытная доярка может надоить ведро молока намного быстрее, чем начинающая. Но в информатике такие различия в основном игнорируются, поскольку они временные

и не очень важные. Ведь быстродействие электронных вычислительных машин постоянно растет, а начинающие доярки могут стать более опытными и в конечном счете научиться доить коров быстрее. Важнее другое: различия во времени выполнения разных алгоритмов для решения одной и той же задачи. Так, если вам требуется получить стакан молока для каждого члена вашей семьи, вы могли бы принести этот стакан каждому из них в отдельности или же принести кувшин с молоком и разлить молоко по стаканам уже за накрытым к завтраку столом. В последнем случае вам придется преодолеть расстояние от стола до коровника дважды, тогда как в первом случае — дважды для каждого члена семьи (для семьи из пяти членов — десять раз). Такое различие двух алгоритмов существует независимо о того, как быстро вы можете доить корову и ходить, а следовательно, оно служит явным признаком сложности обоих алгоритмов и может стать основанием для выбора одного из них.

Помимо времени выполнения, алгоритмы могут различаться ресурсами, которые требуются для их выполнения. Допустим, что по утрам вы любите пить кофе, а не молоко и можете заварить кофе в кофеварке или джезве. Оба способа требуют наличия воды и молотого кофе, но первый из них дополнительно требует фильтров для кофе. Требования к ресурсам для разных алгоритмов получения молока еще более разительны. Чтобы получить парное молоко, требуется корова, тогда как для хранения молока, приобретенного в продуктовом магазине, нужен холодильник. Данный пример также показывает, что результаты вычислений могут быть сохранены для последующего применения и что вычисления иногда могут быть заменены хранением. Так, можно сэкономить на дойке коровы, храня надоенное ранее молоко в холодильнике.

За получение результатов выполнения алгоритма приходится расплачиваться используемыми ресурсами. Следовательно, чтобы сравнить разные алгоритмы для решения одной и той же задачи, очень важно иметь возможность измерить объем потребляемых ими ресурсов. Возможно, иногда придется жертвовать точностью ради эффективности. Допустим, что по дороге на работу вам нужно купить ряд продуктов в магазине. А поскольку вы спешите, то оставляете мелочь, полученную на сдачу, а не кладете ее в кошелек. Точный алгоритм должен предполагать обмен конкретной суммы денег на продукты, тогда как приближенный алгоритм — округление суммы, чтобы ускорить выполняемую вами торговую операцию.

Исследование свойств алгоритмов и их вычислений, включая требования к ресурсам, является очень важной задачей информатики, поскольку оно позволяет легче судить, является ли отдельный алгоритм пригодным для решения конкретной задачи. В продолжение истории Гензеля и Гретель в главе 2, "От слов к делу: когда действительно происходит вычисление", будет пояснено, каким образом разные вычисления могут быть выполнены с использованием одного и того же алгоритма и как измерить объем требующихся для этого ресурсов.

2

От слов к делу: когда действительно происходит вычисление

В предыдущей главе было показано, каким образом Гензель и Гретель решили задачу выживания, вычислив свой обратный путь домой. Это вычисление систематически преобразовало их местоположение в течение отдельных шагов и решило задачу перемещением из исходного положения в лесу, представляющего опасность, в конечное положение дома, представляющее безопасность. Вычисление обратного пути домой стало результатом выполнения алгоритма для следования по пути, проложенному по камешкам. Вычисление происходит в том случае, когда вступают в действие алгоритмы.

Несмотря на то что у нас теперь имеется ясное представление о том, чем вычисление *является*, мы рассмотрели лишь одну сторону того, что на самом деле вычисление *делает*, а именно: преобразует представление. Но имеется еще немало дополнительных подробностей, заслуживающих внимания. Следовательно, чтобы расширить понимание за пределы *статического* представления через алгоритмы, в этой главе обсуждается *динамическое* поведение вычисления.

Алгоритм примечателен тем, что им можно пользоваться неоднократно для решения разных задач. Как же это происходит и как вообще возможно, что на основании одного конкретного описания алгоритма можно выполнять разные вычисления? Более того, как пояснялось в предыдущей главе, вычисление является результатом выполнения алгоритма, но кто или что выполняет алгоритм? И наконец, во что обходится роскошь иметь в своем распоряжении алгоритм для решения задачи? Алгоритм оказывается приемлемым вариантом выбора лишь в том случае, если он может быстро дать решение поставленной задачи с использованием имеющихся ресурсов.

Создание разнообразия

Если вернуться к истории Гензеля и Гретель, то, как было показано ранее, их алгоритм отслеживания обратного пути домой по камешкам может быть повторно использован в разных ситуациях. А теперь рассмотрим более подробно, как это на самом деле осуществляется. Описание данного алгоритма зафиксировано, и поэтому в некоторой части этого описания необходимо принимать во внимание изменяемость вычислений. И эта часть называется *параметром*. В алгоритме параметр обозначает конкретное значение, а когда выполняется алгоритм, то вместо параметра в данном алгоритме должно быть подставлено конкретное значение. Такое значение называется *входным значением* или просто *входом* для *алгоритма*.

Например, в алгоритме для приготовления кофе может быть использован параметр *количество*, обозначающий количество заваренных чашек кофе, чтобы на него можно было ссылаться в инструкциях к данному алгоритму. Ниже приведена выдержка из описания данного алгоритма [1]:

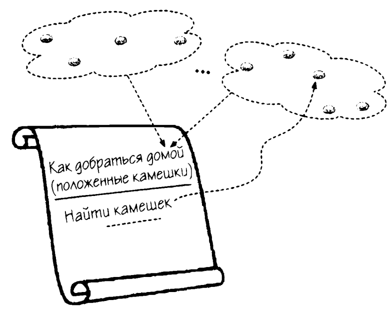
> Залить воду на заданное *количество* чашек.
>
> Добавить количество столовых ложек молотого кофе, в 1,5 раза большее заданного *количества*.

Чтобы выполнить данный алгоритм для приготовления трех чашек кофе, необходимо подставить входное значение **3** вместо параметра *количество* в инструкциях к данному алгоритму. В итоге получается следующая специализированная версия данного алгоритма:

> Залить воду на 3 чашки.
>
> Добавить количество столовых ложек молотого кофе, в 1,5 раза большее 3.

Благодаря использованию параметра алгоритм можно применять в самых разных ситуациях. Каждая ситуация представлена отдельным входным значением (в данном примере — количеством заваренных чашек кофе), подставляемым вместо параметра, причем после подстановки алгоритм приспосабливается к ситуации, представленной входным значением.

В алгоритме Гензеля и Гретель применяется параметр, определяющий количество камешков, брошенных в лесу. Раньше этот параметр не указывался явно, поскольку в инструкции фраза "Найти не посещавшийся прежде камешек" ясно указывает на камешки, брошенные Гензелем. Но этот параметр можно указать явно, заменив упомянутую выше инструкцию следующей: "Найти из числа *брошенных Гензелем камешков* не посещавшийся прежде камешек". Всякий раз, когда выполняется данный алгоритм, параметр *брошенные Гензелем камешки* заменяется количеством брошенных Гензелем камешков, по крайней мере мы можем так думать. Очевидно, что



мы не можем физически ввести камешки в описание данного алгоритма и поэтому трактуем параметр как ссылку или указатель на входное значение. *Указатель* — это механизм доступа к входному значению. Он сообщает нам, где искать входное значение, когда оно потребуется в алгоритме. В алгоритме поиска пути входное значение должно быть обнаружено на поверхности почвы в лесу. А в алгоритме приготовления кофе входное значение мы держим в уме, извлекая его из памяти всякий раз, когда его требуется подставить вместо параметра. Тем не менее понятие подстановки служит удобной аналогией, помогающей установить конкретную взаимосвязь между алгоритмом и вычислением.

Внедряя параметр и подставляя вместо него конкретные значения, мы можем обобщить алгоритм, чтобы сделать его пригодным в самых разных ситуациях. Так, если в конкретном алгоритме подъема по утрам содержится инструкция "Подъем в *6:30* утра", мы можем подставить вместо конкретного времени параметр *время подъема*, что приведет к обобщенной инструкции "Подъем в *время подъема*". Аналогично алгоритм приготовления овсянки может быть расширен после введения параметра *фрукты* в соответствующую его часть.

Но с другой стороны, теперь, чтобы выполнить алгоритм подъема по утрам, нам придется снабдить его входным значением, чтобы сделать инструкции более конкретными, заменив параметр значением конкретного времени подъема. И хотя это, как правило, не вызывает особых затруднений, требуется принятие определенного решения, и возникает потенциальный источник ошибок. Все зависит от того, насколько ценной оказывается изменяемость. Будильник, способный пробудить лишь в одно заранее установленное, но никогда не изменяемое время, вряд ли можно считать приемлемым, хотя многие люди могут обходиться и без выбора другого времени для пробуждения.

Наконец, алгоритм, не имеющий параметров, а следовательно, лишенный возможности принимать разные входные значения, будет всегда производить одно и то же вычисление. Как упоминалось ранее, это не вызывает особых затруднений в алгоритмах со вр*е*менным физическим результатом, как в случае рецептов для приготовления пирогов (поскольку приготовленные пироги съедаются), или подачи молока к завтраку (за которым молоко выпивается). В подобных случаях повторное вычисление одного и того же результата имеет смысл. Но в алгоритмах для вычисления значений, которые могут быть сохранены для последующего применения, требуется один или несколько параметров для повторного использования.

Параметры являются очень важной составляющей алгоритмов, но на вопрос, насколько общим или конкретным должен быть алгоритм, трудно дать простой

ответ. Более подробно этот вопрос обсуждается в главе 15, "С высоты птичьего полета: от абстракции к деталям".

Кто выполняет алгоритм

Как пояснялось ранее, вычисление является результатом выполнения алгоритма. В этой связи возникает следующий вопрос: кто или что и как именно может выполнять алгоритм? Как демонстрировалось в приведенных выше примерах, люди, безусловно, способны выполнять алгоритмы, но на это способны и электронные вычислительные машины (т.е. компьютеры). А какие еще имеются для этого возможности и какие требования предъявляются к выполнению алгоритма?

Того, кто или что может выполнять вычисление, можно, конечно, обозначить термином *компьютер* (или, более обобщенно, *вычислитель*). На самом деле этим термином первоначально обозначались люди, выполнявшие вычисления [2]. Этот термин будет употребляться далее в общем смысле, обозначая любого естественного или искусственного посредника, способного выполнять вычисления.

Можно выделить два принципиально разных случая выполнения алгоритма, исходя из возможностей выполняющего его вычислителя. С одной стороны, имеются универсальные вычислители (например, люди, переносные компьютеры или смартфоны). В принципе универсальный вычислитель может выполнить любой алгоритм, при условии, что он выражен на понятном вычислителю языке. Универсальные вычислители устанавливают взаимосвязь между алгоритмами и вычислением. Всякий раз, когда такой вычислитель выполняет алгоритм для решения конкретной задачи, он предпринимает шаги, изменяющие некоторое представление (рис. 2.1).

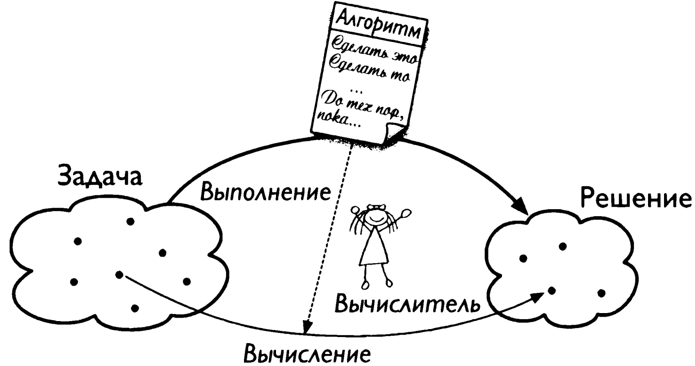


Рис. 2.1. При выполнении алгоритма формируется вычисление. Алгоритм описывает метод, пригодный для решения целого класса задач, а его выполнение оказывает воздействие на представление конкретного примера задачи. Выполнять алгоритм должен вычислитель (например, человек или машина), способный понимать язык, на котором задан алгоритм

С другой стороны, имеются вычислители, выполняющие лишь один алгоритм (или, возможно, ряд предопределенных алгоритмов). Например, карманный калькулятор состоит из электронных схем, жестко запрограммированных на алгоритмы для выполнения арифметических вычислений, а будильник — на производство звуковых сигналов для пробуждения в установленное время. Еще один занимательный пример можно найти в клеточной биологии.

Подумайте о том, что миллионы раз происходит в ваших клетках, пока вы читаете этот текст. Рибосомы производят белки для поддержания функций ваших клеток и представляют собой мелкие вычислительные машины, собирающие белки так, как описано в молекулах РНК. Это последовательности аминокислот, которые предписывают рибосоме производить конкретные белки. Вы живете благодаря вычислению, производимому рибосомными компьютерами в ваших клетках, надежно выполняющими алгоритм преобразования молекул РНК в белки. Но даже если алгоритм, применяемый рибосомой, может произвести огромное разнообразие белков, это всего лишь алгоритм, который способны выполнять только рибосомы. Несмотря на всю полезность рибосом, их возможности все же ограничены. В частности, они не в состоянии помочь вам одеться или найти дорогу из леса.

В отличие от вычислителей с жестко запрограммированными алгоритмами, к универсальным вычислителям предъявляется следующее важное требование: они должны понимать язык, на котором заданы их алгоритмы. Если такой вычислитель представляет собой вычислительную машину, то его алгоритм называется *программой*, а язык, на котором он задан, — *языком программирования.*

Если бы Гензель и Гретель написали свои воспоминания, включив в них описание алгоритма, спасшего им жизнь, то другие дети, прочитавшие книгу их воспоминаний, смогли бы выполнить описанный в ней алгоритм только в том случае, если бы понимали язык, на котором написана книга. Но подобное требование не распространяется на неуниверсальные компьютеры, выполняющие лишь фиксированные, жестко прошитые алгоритмы.

В качестве требования к каждому виду компьютера предъявляется способность доступа к представлению, применяемому в алгоритме. В частности, вычислитель должен быть в состоянии оказывать воздействие на требующиеся изменения в представлении. Если бы Гензель и Гретель были привязаны цепью к дереву, то изобретенный ими алгоритм ничем бы им не помог, поскольку они не смогли бы изменить свое положение, что требуется для выполнения алгоритма поиска пути домой.

Таким образом, любой компьютер должен быть в состоянии читать и управлять представлениями, на которые воздействуют алгоритмы. Кроме того,

универсальный компьютер должен понимать язык, на котором выражены алгоритмы. С этого места и далее термин *компьютер* употребляется для обозначения *универсального компьютера*.

Стоимость жизни

Компьютер должен выполнять какую-то реальную работу. Об этом всякий раз напоминает факт нагревания ноутбука, когда он воспроизводит высококачественную графику в видеоигре, или быстрой разрядки батареи питания смартфона, когда в фоновом режиме выполняется слишком много приложений. А причина, по которой приходится устанавливать будильник на более ранее время, чем время выхода на работу, объясняется тем, что для выполнения алгоритма подъема по утрам требуется некоторое время.

Одно дело — придумать алгоритм для решения задачи и совсем другое — обеспечить достаточно быстрое получение ее решения в ходе конкретного вычисления, формируемого этим алгоритмом. С этим связан следующий вопрос: имеется ли у компьютера, ответственного за выполнение алгоритма, достаточно ресурсов, прежде всего, для выполнения вычисления.

Например, когда Гензель и Гретель возвращаются домой по обратному пути, прослеживаемому по камешкам, все вычисление выполняется в течение стольких шагов, сколько камешков бросил Гензель [3]. Следует, однако, иметь в виду, что в данном случае под шагом подразумевается "шаг алгоритма", а не "шаг как часть ходьбы". В частности, один шаг алгоритма обычно соответствует нескольким шагам, предпринимаемым Гензелем и Гретель по пути из леса. Таким образом, количество камешков служит мерой времени выполнения алгоритма, поскольку для достижения каждого камешка требуется один шаг алгоритма. Определение количества шагов алгоритма, требующихся для выполнения задачи, означает оценку *временной сложности* алгоритма.

Кроме того, данный алгоритм будет действовать лишь в том случае, если у Гензеля и Гретель окажется достаточно камешков, чтобы покрыть путь от их дома к тому месту в лесу, где их оставили. Это характерный пример ограниченности ресурсов. Недостаток камешков мог быть вызван их ограниченным количеством возле дома (ограниченность внешних ресурсов) или ограниченным объемом карманов Гензеля (ограниченность возможностей компьютера). Оценка *пространственной сложности* алгоритма означает выяснение величины того пространства, которое требуется компьютеру для выполнения алгоритма. В данном

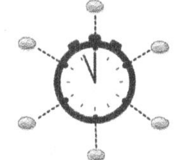
примере это означает выяснение количества камешков, требующихся для поиска пути конкретной длины и объема карманов Гензеля, способных вместить все эти камешки.

Таким образом, несмотря на то что данный алгоритм может теоретически действовать в любом месте в лесу, заранее неизвестно, окажется ли удачным исход вычисления на практике, поскольку это может отнять слишком много времени или потребовать объема ресурсов, превышающего доступные. Прежде чем исследовать ресурсы вычисления более подробно, следует пояснить два важных допущения относительно измерения затрат на вычисления, которые определяют практическую целесообразность подобного рода анализа. Основное внимание далее будет уделено времени выполнения алгоритма, хотя его обсуждение касается и пространственных ресурсов.

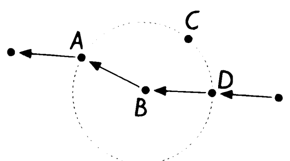
Общее представление о затратах

Алгоритм можно рассматривать как обобщение многих вычислений. Как пояснялось ранее, различия в вычислениях фиксируются в описании алгоритма с помощью параметров, а любое отдельное вычисление может быть получено путем выполнения алгоритма с конкретными входными значениями, подставляемыми вместо параметров. Аналогично было бы желательно получить обобщенное описание требований к ресурсам алгоритма, которое применимо не только к отдельному вычислению, но и ко всем вычислениям в целом. Иными словами, нам требуется найти обобщенное описание затрат на вычисление алгоритма. И такого обобщения можно достичь, используя параметры, позволяющие сделать количество шагов, требующихся для выполнения алгоритма, зависимым от объема входных данных. Таким образом, време*нн*ая сложность является функцией, получающей количество шагов в вычислении для входных данных заданного объема.

Например, количество шагов вычисления, а следовательно, и время выполнения алгоритма прослеживания пути по камешкам приблизительно равны количеству брошенных камешков. А поскольку на пути к разным местам в лесу обычно содержится разное количество камешков, для вычисления этих путей также потребуется разное количество шагов. Это обстоятельство отражается в том, что време*нн*ая сложность выражается в виде функции от объема входных данных. Так, в алгоритме прослеживания пути по камешкам нетрудно получить точную меру для каждого вычисления, поскольку количество шагов вычисления, по-видимому, взаимно-однозначно соответствует количеству камешков. Например, для вычисления пути из 87 камешков потребуется 87 шагов.



Однако так бывает *не* всегда. Обратимся снова к пути, отображенному на рис. 1.3. Данный пример пути был использован для того, чтобы наглядно показать, что алгоритм может застрять. Но им можно воспользоваться и для того, чтобы показать, как тот же алгоритм может привести к вычислениям, требующим меньше шагов, чем имеется камешков. В частности, камешки *B* и *C* видны из места нахождения камешка *D*, и поэтому можно выбрать камешек *B*. А поскольку из места нахождения камешка *B* видны камешки *A* и *C*, следующим можно выбрать камешек *A*, т.е. правильный путь *D*→*B*→*A*. Этот путь проложен в обход камешка *C*, и поэтому для его вычисления потребуется на один шаг меньше, чем количество находящихся на нем камешков.



Обратите внимание на то, что в данном случае количество шагов в вычислении оказалось *меньше*, чем предполагалось благодаря измерению затрат на вычисление алгоритма, которые были переоценены. Таким образом, временн*ая* сложность сообщает о сложности вычисления, которая может возникнуть в *наихудшем случае*. Это помогает принять решение, следует ли вообще выполнять алгоритм для конкретных входных данных. Если оцененное время выполнения алгоритма оказывается приемлемым, то алгоритм может быть выполнен. А если вычисление фактически выполняется быстрее и требует меньше шагов, то еще лучше, хотя сложность в худшем случае гарантирует, что выполнение алгоритма не отнимет времени, большего указанного. Наихудшая оценка времени выполнения для алгоритма подъема по утрам предполагает, что на прием душа может потребоваться 5 минут с учетом времени на нагрев воды, хотя это время может оказаться более коротким, если кто-то уже принимал душ до вас.

Поскольку анализ времени выполнения производится на уровне алгоритмов, он пригоден только для алгоритмов, но не для отдельных вычислений. Это также означает, что время выполнения вычисления можно оценить *до* того, как оно произойдет, поскольку анализ алгоритма основывается на его описании.

Еще одно допущение относительно временн*ой* сложности состоит в том, что шаг алгоритма обычно соответствует нескольким шагам, совершаемым исполнителем данного алгоритма. Это становится очевидно из конкретного примера. Так, камешки, вероятно, бросаются не на расстоянии одного шага, и поэтому Гензелю и Гретель придется сделать несколько шагов, переходя от одного камешка к другому. Но каждый шаг алгоритма совсем необязательно должен самовольно приводить к большему количеству шагов вычисления. Это количество должно быть постоянным и относительно малым по сравнению с количеством шагов, совершаемых по алгоритму. В противном случае сведения о времени выполнения алгоритма окажутся бессодержательными, так как количество шагов алгоритма не будет точно соответствовать фактической мере времени выполнения. Здесь

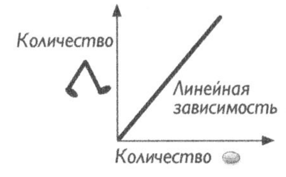
следует вспомнить еще и то обстоятельство, что компьютеры различаются своими характеристиками производительности. В рассматриваемом примере ноги Гензеля могут оказаться длиннее ног Гретель, а следовательно, ему потребуется сделать меньше шагов, чтобы добраться от одного камешка до другого. Но в то же время Гретель может идти быстрее, чем Гензель, т.е. совершить заданное количество шагов за меньшее время. Впрочем, всеми этими факторами можно пренебречь, уделив основное внимание времени выполнения алгоритма.

Рост затрат

Сведения о времен*ó*й сложности алгоритма задаются в виде функции, которая фиксирует различия во времени выполнения разных вычислений. Этот подход отражает то обстоятельство, что чем больше объем входных данных, тем больше времени требуется для выполнения алгоритмов.

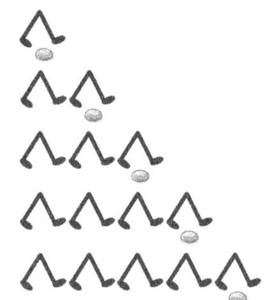
Сложность алгоритма Гензеля и Гретель можно охарактеризовать следующим правилом: "Время выполнения пропорционально количеству камешков", а это означает, что отношение количества совершенных на пути шагов и камешков остается постоянным. Иными словами, если удвоится длина пути, а следовательно, и количество камешков, то и время выполнения возрастет вдвое. Следует, однако, иметь в виду, что это совсем не означает, что количество совершенных на пути шагов равно *количеству* камешков, а лишь увеличивается или уменьшается таким же образом, как входные данные.

Такая зависимость называется *линейной* и на графике зависимости количества шагов, необходимых для совершения на пути, от количества имеющихся камешков выглядит, как прямая линия. В подобных случаях говорят, что алгоритм обладает *линейной* времен*ó*й зависимостью, а иногда он называется еще короче — *линейным.*



Линейные алгоритмы очень удобны и зачастую оказываются лучшим из того, на что только можно надеяться. В качестве примера разной времен*ó*й сложности рассмотрим алгоритм, выполняемый Гензелем, когда он бросает камешки по пути из дому в лес. В первоначальной версии рассматриваемой здесь истории все камешки находились в его кармане, а следовательно, он мог бросать их по дороге в лес. Гензель совершает постоянное количество шагов, чтобы достичь следующего места для бросания очередного камешка.

А теперь представим, что у Гензеля нет возможности хранить и скрывать от родителей камешки, но есть возможность возвращаться за каждым новым камешком домой. В таком случае он возвращается и приносит новый



камешек туда, где его следует бросить, т.е. совершает в два раза больше шагов, чтобы доставить каждый камешек на свое место. Таким образом, общее количество совершенных шагов состоит из суммы шагов, требующихся для доставки каждого камешка на пути из дому в лес. А поскольку расстояние от дома постоянно увеличивается с каждым брошенным камешком, общее количество совершенных шагов пропорционально сумме 1 + 2 + 3 + 4 + 5 + ..., т.е. квадрату числа брошенных камешков.

Эту зависимость можно продемонстрировать наглядно, представив расстояние, которое должен преодолеть Гензель, измеренным количеством камешков. Чтобы бросить два камешка, Гензелю придется добраться до того места, где он сможет бросить первый камешек, вернуться назад, чтобы принести еще один камешек, а затем, пройдя мимо первого камешка, добраться до того места, где он сможет бросить второй камешек. В итоге ему придется преодолеть общее расстояние, равное расстоянию между четырьмя камешками. Чтобы бросить три камешка, Гензелю придется сначала преодолеть уже известное учетверенное расстояние для доставки на место двух камешков, а затем вернуться назад и принести третий камешек, т.е. преодолеть удвоенное расстояние между камешками. Для того же, чтобы положить третий камешек на место, ему придется преодолеть утроенное расстояние между камешками, что в конечном итоге составит общее расстояние, равное 4 + 2 + 3 = 9 расстояний между камешками.

Рассмотрим еще один камешек. Прежде чем бросить четвертый камешек, Гензель уже преодолел расстояние для доставки на место трех камешков, поэтому он должен сначала вернуться назад (три расстояния между камешками), а затем преодолеть еще четыре расстояния между камешками, чтобы добраться до места бросания четвертого камешка, так что в итоге он проходит расстояние, равное 9 + 3 + 4 = 16 расстояниям между камешками. Аналогично можно вычислить расстояние для разбрасывания пяти камешков (16 + 4 + 5 = 25), шести камешков (25 + 5 + 6 = 36) и т.д.

Из сказанного выше ясно прослеживается следующая закономерность: количество шагов, которое требуется совершить Гензелю, пропорционально квадрату числа брошенных камешков. Об алгоритмах с такой закономерностью говорят, что они обладают *квадратичной* временн*о*й сложностью, а еще короче они называются *квадратичными*. Время выполнения квадратичного алгоритма растет намного быстрее, чем время выполнения линейного алгоритма. Так, если для разбрасывания десяти камешков потребуется выполнить 10 шагов линейного алгоритма, то для такого же количества камешков требуется 100 шагов квадратичного алгоритма, а для ста камешков — 100 и 10 000 шагов соответственно.

Следует, однако, иметь в виду, что фактическое количество шагов Гензеля может быть б*о*льшим. Как упоминалось ранее, линейный алгоритм может потребовать любое постоянное количество шагов между камешками, скажем, 2, 3 или 14. Следовательно, он может потребовать 30, 20 или 140 шагов соответственно для преодоления пути, например, для 10 камешков. То же самое справедливо и для квадратичного алгоритма, количество шагов которого может быть также умножено на коэффициент. Это наблюдение указывает на то, что в любом случае линейный алгоритм совсем необязательно должен выполняться быстрее, чем квадратичный. При большом постоянном коэффициенте он может потребовать больше шагов, чем квадратичный алгоритм с небольшим постоянным коэффициентом — конечно, для небольших входных данных. Например, линейный алгоритм, требующий 14 шагов на каждый камешек, потребует 140 шагов для входных данных из 10 камешков, что на 40 шагов больше, чем требуется квадратичному алгоритму с 1 шагом на каждый камешек. Но в то же время можно заметить, что по мере увеличения объема входных данных влияние постоянного коэффициента постепенно снижается, а на его место приходит рост шагов квадратичного алгоритма. Например, для разбрасывания 100 камешков линейный алгоритм потребует 1400 шагов, тогда как квадратичный алгоритм — уже 10 000 шагов.

Для рассматриваемой здесь истории квадратичный алгоритм не годится и не будет действовать. Подумайте о времени, которое потребовалось бы на то, чтобы бросить последний камешек. Для этого Гензелю пришлось бы сначала пройти весь путь домой, а затем вернуться в лес, т.е., по сути, пройти удвоенное расстояние до леса. Отец потерял бы терпение, пока он бросал бы камешки по такому квадратичному алгоритму.

> *Отец спросил его: "Гензель, что это ты все оглядываешься да отстаешь? Смотри, не зевай и прибавь шагу".*

Отец явно не ожидал, что Гензель будет возвращаться домой всякий раз, когда ему потребуется очередной камешек. Следовательно, время выполнения алгоритма действительно имеет большое значение. Так, если алгоритм слишком медленный, он может оказаться бесполезным с практической точки зрения (см. главу 7, "Трудноразрешимые задачи").

Данный пример демонстрирует также, что временн*ая* и пространственная эффективность алгоритма нередко взаимозависимы. В данном случае можно повысить временн*ую* эффективность алгоритма от квадратичной до линейной за счет повышения используемого пространства для хранения камешков, т.е. воспользоваться линейной пространственной эффективностью, обеспечив возможность хранения всех необходимых камешков в карманах Гензеля.

Если два алгоритма решают одну и ту же задачу, но показатель времен*ной* сложности у одного из них ниже, чем у другого, то более быстродействующий алгоритм считается *более эффективным* по времени выполнения. Аналогично, если один алгоритм использует меньше оперативной памяти, чем другой, то он считается *более эффективным* с точки зрения использования пространства. В данном примере линейный алгоритм разбрасывания камешков оказывается более эффективным по времени выполнения, чем квадратичный алгоритм, но менее эффективным по используемому пространству, поскольку заполняет карманы Гензеля камешками.

Дальнейшее исследование

Камешки в истории Гензеля и Гретель служили представлениями для алгоритма поиска пути, но у отметки пути могут быть и другие применения. С одной стороны, они помогают найти обратный путь при исследовании неизвестной местности, что, собственно, и происходит в истории Гензеля и Гретель. С другой стороны, они помогают одним людям следовать за другими, как, например, в фантастическом романе Дж. Толкиена (J.J.R. Tolkien) *Властелин колец: Две крепости*, когда Пиппин, взятый вместе с Мэри в плен орками, бросает брошь как знак для Аргорна, Леголаса и Гимли. Аналогично в фильме *Индиана Джонс и королевство хрустального черепа* Мак тайно разбрасывает радиомаяки, чтобы за ним могли последовать другие.

Во всех трех приведенных выше примерах метки располагаются на более или менее открытой местности, где можно передвигаться практически в любом направлении. Но бывают и другие случаи, когда передвижение ограничивается определенным числом соединений между перекрестками, как в романе Марка Твена (Mark Twain) *Приключения Тома Сойера*, когда Том и Бекки исследуют пещеру и оставляют дымовые отметки на стенах, чтобы найти обратный путь (хотя они все равно заблудились в пещере). И когда через несколько дней Бекки настолько ослабела, что не могла идти дальше, Том продолжал исследовать пещеру, употребляя в качестве более надежного метода нить от воздушного змея, чтобы найти обратный путь к Бекки. И вероятно, самый известный (и древний) пример применения нити, чтобы не потеряться в лабиринте, можно найти в древнегреческом мифе о Минотавре, в котором Тезей воспользовался нитью, которую ему дала Ариадна, чтобы выбраться из лабиринта. Таким же методом воспользовался и Адсо, послушник монашеского ордена бенедиктинцев, в романе Умберто Эко (Umberto Eco) *Имя розы*, чтобы найти обратный путь из лабиринта в монастырской библиотеке.

Любопытно сравнить разные виды отметок, употреблявшихся в упомянутых выше историях, а также следствия из соответствующих алгоритмов поиска пути. Например, применение камешков, броши или дымовых отметок все равно требует некоторого поиска, чтобы передвигаться от одной отметки к другой, поскольку они оказываются лишь в немногих местах. А вот нить служит постоянным

путеводителем, которого достаточно придерживаться без всякого поиска. Более того, используя нить, можно избежать тупиковых ситуаций, подобных описанным в главе 1, "Путь к пониманию вычислений". Такие ситуации могут возникнуть, если употреблять камешки или другие виды дискретных отметок.

Метод Гензеля и Гретель применяется в современных пользовательских интерфейсах файловых систем или систем запросов, где он называется ***навигационной цепочкой*** (буквально — breadcrumb navigation, навигация по хлебным крошкам). Например, проводники файловых систем нередко отображают список родительских, или охватывающих, папок для текущей папки. Кроме того, в поисковых интерфейсах программ электронной почты или баз данных нередко отображается список критериев поиска, применяемых к показываемому в настоящий момент элементу выбора. Действие по возврату к родительской папке или удалению последнего критерия поиска с целью расширить переделы выбора соответствует движению к камешку и его подбиранию.

Представление и структуры данных

Шерлок Холмс

По дороге

Представьте себя на пути на свою работу. Добираясь до работы на автомашине, на велосипеде или пешком, вы встречаете дорожные знаки и светофоры, регулирующие ваше движение и движение ваших попутчиков. Определенные правила, связанные с дорожными знаками, являются алгоритмами. Например, знак остановки на перекрестке предписывает вам остановиться, подождать, пока подъехавший прежде вас транспорт пересечет перекресток, после чего вы можете пересечь его сами [1]. Соблюдение правил дорожного движения, указанных на дорожных знаках, по существу, означает выполнение соответствующих алгоритмов, а возникающее в итоге движение служит примером вычисления. В этот вид деятельности вовлечено немало водителей и транспортных средств, пользующихся дорогой как общим ресурсом, и поэтому дорожное движение фактически служит примером распределенного вычисления, хотя здесь это и не самое главное.

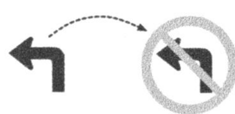
Примечательно, что ежедневно миллионам людей с совершенно разными целями удается эффективно согласовывать свои действия и успешно прокладывать свой путь среди других. Безусловно, дорожные заторы и аварии происходят регулярно, но в общем дорожное движение действует исправно. Еще более примечательно, что все это становится возможным благодаря небольшому ряду дорожных знаков. В частности, с помощью дорожного знака остановки в виде красного восьмиугольника со словом "STOP" во всех странах координируется пересечение дорог бесконечным потоком транспорта.

Каким же образом подобные знаки достигают столь сильного эффекта? Главное наблюдение состоит в том, что они несут *смысловое значение*. Например, указательный знак предоставляет сведения о направлении к интересующему месту. По такому знаку путник может принять решение, куда ему следует повернуть или где остановиться. Другие знаки предупреждают (например, о препятствиях или крутых поворотах), запрещают определенные действия (например, ограничивают максимальную скорость) и регулируют доступ к общим дорожным пространствам (например, перекресткам). В главе 1, "Путь к пониманию вычислений", термином *представление* определялись знаки, обозначавшие нечто другое (например, камешки, представлявшие места нахождения в лесу). С этой точки зрения знаки получают свою силу на том основании, что являются представлениями.

Действие знака не проявляется каким-то волшебным образом; оно должно быть извлечено некоторым агентом. Этот процесс называется *интерпретацией*, причем разные агенты могут интерпретировать знак по-разному. Например, типичный участник дорожного движения интерпретирует дорожные знаки как информацию или инструкцию; однако в то же время они могут быть предметом коллекционирования. Интерпретация требуется для понимания не только дорожных знаков, но и всех представлений в информатике.

Знаки связаны с вычислениями разными способами, что подчеркивает их важность. Во-первых, знак может непосредственно представлять вычисление, как это происходит со знаком остановки, обозначающим конкретный алгоритм для выполнения участниками дорожного движения. И даже если вычисление, представленное отдельным знаком, оказывается тривиальным, определенное сочетание подобных знаков способно произвести значительное вычисление как совокупное. Характерным тому примером служат камешки Гензеля и Гретель. Отдельный камешек инициирует простое действие “Подойди ко мне, если ты не посещал меня ранее”, тогда как все камешки вместе обеспечивают спасительное движение из леса.

Во-вторых, вычислением является систематическое преобразование знака [2]. Например, перечеркивание знака меняет его смысловое значение на обратное.



Это нередко делается для того, чтобы запретить определенные действия (например, изогнутая стрелка, перечеркнутая по диагонали в красном кругу, запрещает поворот в ту в сторону, куда она указывает). Еще одним примером служит светофор: когда красный свет сменяется зеленым, это означает соответствующую смену действия с остановки на движение.

И наконец, в-третьих, вычислением является процесс интерпретации знака. Это не вполне очевидно для таких простых знаков, как камешки или знаки остановки, но становится более очевидным при рассмотрении составных знаков. Характерным тому примером служит перечеркнутый знак, смысловое значение которого получается из исходного знака, на которое накладывается смысловое значение перечеркивания. К другим примерам относятся знаки, направляющие к пунктам питания. Их смысловое значение получается из сочетания указываемого направления и логотипов заведений общественного питания, на которых обозначены разные виды пищи. Более подробно интерпретация обсуждается в главах 9, “Поиск нужного тона: смысл звука”, и 13, “Все дело в интерпретации”. Здесь же главное — что знаки вовлекаются в вычисление самыми разными способами. Поэтому было бы неплохо понимать, что они собой представляют, как они действуют и какие роли они играют в вычислении. И это является предметом главы 3, “Тайна знаков”.

3 Тайна знаков

Как было показано в главах 1 и 2, вычисления служат для манипулирования представлениями, которые являются символами или знаками, обозначающими содержательные сущности. Ранее было показано, каким образом Гензель и Гретель употребили камешки как представления для отметки мест нахождения в поддержку изобретенного ими алгоритма поиска обратного пути домой. В этом отношении камешки должны были отвечать целому ряду требований, которые мы приняли как данность. Если же проанализировать эти требования более внимательно, то удастся лучше понять, что такое представление и как оно поддерживает вычисление.

Представление состоит по крайней мере из двух частей: той, которая представляет, и той, которая представляется. Это обстоятельство фиксируется в понятии *знака*, или *символа* (sign). При этом во внимание необходимо принимать следующие три особенности знаков: они способны оперировать на многих уровнях; они могут быть неоднозначными (например, один знак может представлять разные предметы); кроме того, один предмет может быть представлен разными знаками. В этой главе обсуждаются также различные механизмы, позволяющие употреблять знаки как представления.

Знаки представления

Есть ли у вас какие-нибудь сомнения в том, 1 + 1 равно 2? Вероятно, нет, если только вы не перенеслись во времени из Древнего Рима. Ведь в таком случае приведенные выше числовые символы показались бы вам загадочными, но вместо

этого вы охотно согласились бы с тем, что I + I равно II, если бы кто-нибудь объяснил вам назначение символа +, которое не было известно древним римлянам, поскольку впервые этот символ вошел в употребление в XV веке. А если бы можно было задать тот же самый вопрос электронной вычислительной машине, построенной на основании двоичной системы счисления, то она ответила бы, что 1 + 1 равно 10 [1]. Что же все это означает?

Данный пример демонстрирует, что для обсуждения даже самых простых арифметических действий требуется соглашение о символах, представляющих отдельные величины. То же самое справедливо и для вычислений величин. Так, удвоение числа 11 дает в итоге число 22 в десятичной системе, основанной на индо-арабских цифрах. В древнеримской системе счисления удвоение числа II дает число IV, но не IIII [2], а на электронной вычислительной машине будет получен результат 110, поскольку 11 — это двоичное представление десятичного числа 3, а 110 — двоичное представление десятичного числа 6 [3].

Все это наглядно показывает, что смысловое значение вычисления зависит от смыслового значения того представления, которое оно преобразует. Например, вычисление, преобразующее двоичное число 11 в число 110, означает удвоение первого из них, тогда как в десятичной системе счисления оно означает умножение числа 11 на 10. Но это же преобразование в древнеримской системе счисления не имеет никакого смысла, поскольку в ней отсутствует представление нуля.

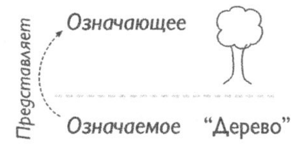
В связи с тем что представление играет решающую роль в вычислении, очень важно понять, чем же оно в действительности является. А поскольку представление слов употребляется по-разному, то очень важно выяснить его смысловое значение в информатике. С этой целью прибегнем к помощи Шерлока Холмса, известного сыщика, методы расследования уголовных дел которого позволяют узнать немало о том, каким образом представления оказывают поддержку вычислению. Для Шерлока Холмса были характерны особое внимание к мельчайшим деталям и необычная их интерпретация. И зачастую подобные заключения помогали ему раскрыть преступления, но иногда они служили лишь увлекательным способом раскрытия сведений для развития истории. Но в любом случае заключения Шерлока Холмса нередко основывались на интерпретации представлений.

Представления играют важную роль в *Собаке Баскервилей* — одном из самых известных и популярных повествований о приключениях Шерлока Холмса. В привычном для Шерлока Холмса стиле эта повесть начинается с ряда наблюдений о тросточке, оставленной посетителем, доктором Мортимером. Холмс и его помощник доктор Ватсон интерпретируют следующую надпись, выгравированную на тросточке: "Джеймсу Мортимеру, MRCS, от коллег по CCH". Как англичанам Холмсу и Ватсону было известно, что сокращение "MRCS" обозначает или *представляет* название "Member of the Royal College of Surgeons" (Член Королевского хирургического колледжа). Основываясь на этом знании и обратившись к

помощи медицинского словаря, Холмс приходит к заключению, что сокращение "ССН" обозначает "Charing Cross Hospital" (Госпиталь на Чаринг-Кросс), поскольку доктор Мортимер некогда в нем работал. Кроме того, Холмс приходит к выводу, что тросточка была подарена доктору Мортимеру в знак признания его заслуг, когда он покинул госпиталь и стал сельским врачом, хотя впоследствии выяснится, что этот вывод Холмса оказался неверным и что тросточка была подарена доктору Мортимеру по случаю его женитьбы[1].

Выгравированная надпись содержит три немедленно распознаваемые представления: два сокращения и краткое посвящение, представляющее в целом событие, связанное со свадебной годовщиной доктора Мортимера. Каждое из этих представлений зафиксировано в форме *знака* — понятия, введенного швейцарским языковедом Фердинандом де Соссюром (Ferdinand de Saussure). Знак состоит из следующих двух частей: *означающего* и *означаемого.* Означающее — это то, что воспринимается или представляется, тогда как означаемое — это понятие, обозначаемое означающим. Чтобы связать такое понятие знака с понятием представления, можно сказать, что означающее *представляет* означаемое. В этой книге термин *представлять* всегда употребляется в смысле "означать", и поэтому можно также сказать, что означающее обозначает означаемое.

Понятие знака имеет большое значение, поскольку оно лаконично фиксирует понятие представления. В частности, взаимосвязь означающего с означаемым и то, что оно представляет, производит смысловое значение (в данном случае — часть профессионального прошлого доктора Мортимера, выясненную из надписи на его тросточке). Означаемое нередко ошибочно принимается за какой-нибудь физический объект из окружающего мира, но Соссюр имел в виду совсем другое. Например, слово "дерево" означает не конкретное дерево, а понятие дерева, которое складывается в нашем уме.



В силу этой особенности описать знаки очень трудно, поскольку, с одной стороны, текст и диаграммы, используемые для описания знаков, сами являются знаками, а с другой стороны, абстрактные понятия, возникающие в уме, нельзя отобразить непосредственно и в конечном счете их приходится представлять все теми же знаками. В литературе о семиотике, т.е. теории знаков и их смысловом значении, понятие знака нередко иллюстрируется диаграммой, состоящей из слова "дерево" в качестве примера означающего, а также из рисунка дерева в качестве предмета, означаемого "деревом". Но ведь сам рисунок является знаком для понятия дерева, а следовательно, подобная диаграмма может ввести в заблуждение, поскольку "дерево" является означающим не рисунка, а того, что сам рисунок означает, т.е. понятия дерева.

[1] После (и вследствие) которой он покинул госпиталь и стал сельским врачом. — *Примеч. ред.*

Мы привязаны к языку, чтобы рассуждать о самом языке и представлении, и поэтому мы не в состоянии разрешить эту дилемму и вообще представить понятия иначе, чем языковыми средствами. Требуется ли рассуждать об означающем или означаемом, нам всегда приходится пользоваться для этой цели означающим. Правда, мы можем зачастую обойти это препятствие, заключив слово или фразу в кавычки, употребляемые в качестве означающего, или же набрав слово или фразу в специальном формате (например, выделив курсивом). Кавычки действуют как ссылка на слово или фразу, а следовательно, не требуют интерпретации. А если слово или фраза употребляется без кавычек, то она интерпретируется как то, что обозначает, т.е. как понятие означающего.

Таким образом, "дерево" означает слово из шести букв, тогда как само это слово без кавычек — понятие дерева. Отличие слова в кавычках, обозначающего само слово, от этого же слова без кавычек, обозначающего конкретное понятие, называется в аналитической философии *отличием употребления от упоминания*. Слово без кавычек фактически *употребляется* и обозначает именно то, что оно представляет, тогда как слово в кавычках только *упоминается* и не называет то, что оно обозначает. Заключение в кавычки останавливает воздействие интерпретации на том, что находится в кавычках, а следовательно, дает возможность ясно отличить рассуждение о слове от его смыслового значения. Например, мы можем сказать, что слово "дерево" состоит из шести букв, тогда как само дерево состоит не из букв, а из ствола, ветвей и листьев.

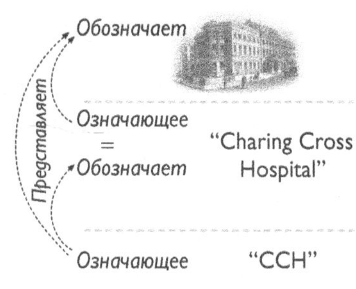
Простое, на первый взгляд, понятие знака содержит немалую гибкость. Например, знаки могут действовать на многих уровнях, иметь разное смысловое значение и по-разному устанавливать связь межу означающим и означаемым. Все эти особенности знаков описываются в последующих разделах.

О знаках до самого основания

Помимо трех знаков, обнаруженных на тросточке доктора Мортимера (сокращения "MRCS", обозначающего члена Королевского хирургического колледжа, сокращения "CCH", обозначающего госпиталь на Чаринг-Кросс, а также всего посвящения в целом, обозначающего женитьбу доктора Мортимера), для данного примера подходит и ряд других знаков. Прежде всего, название "член Королевского хирургического колледжа" обозначает членство в профессиональном обществе, а название "госпиталь на Чаринг-Кросс" — конкретную больницу в Лондоне (т.е. понятие, а не здание). Но это еще не все. Сокращение "MRCS" обозначает членство в хирургическом обществе, а сокращение "CCH" — лондонскую больницу.

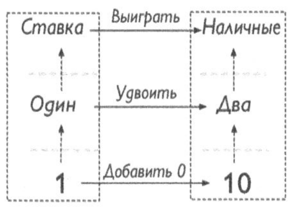
Таким образом, сокращение имеет два возможных смысловых значения и может иметь два разных означаемых, поскольку простирается до означаемого своего

означаемого. Что это означает? Означаемым сокращения "CCH" служит фраза "Charing Cross Hospital", которая, в свою очередь, служит означающим лондонского госпиталя, и поэтому сокращение "CCH" может представлять лондонский госпиталь путем объединения двух представлений, в которых означаемое первого из них служит означающим второго. Аналогично сокращение "MRCS" объединяет два уровня представления в один, поскольку представляет членство в хирургическом обществе, ссылаясь на означаемое названия "Member of the Royal College of Surgeons".



Почему это так важно и какое это имеет отношение к информатике? Напомним, что в главе 1, "Путь к пониманию вычислений", были выявлены две разные формы представления: представление задачи и представление вычисления. Благодаря тому что знаки позволяют объединять два уровня представления в один, появляется возможность придавать смысловое значение вычислениям, которые в противном случае оказались бы чисто символическими. Поясним эту мысль на примере, в котором используется обсуждавшееся ранее представление чисел.

В бинарной форме означающее "1" представляет число "один" на уровне представления вычислений. Это число может представлять разные факты в различных контекстах, а следовательно, имеет разные представления задачи. Если вы играете в рулетку, это могла бы, например, быть денежная сумма, которую вы ставите на черное. А преобразование, присоединяющее 0 к 1, означает на уровне представления вычисления удвоение числа "один" до числа "два". В контексте представления задачи преобразование может также означать, что выпало черное, ваша ставка выиграла и теперь принадлежащая вам денежная сумма удвоилась.

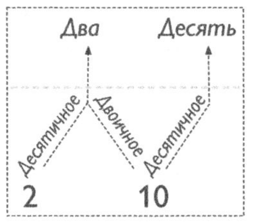


Аналогично камешки в лесу служат означающими, которые представляют местоположение Гензеля и Гретель и относятся к представлению вычисления. Кроме того, достигнутое местоположение представляет опасное положение в представлении задачи. Чтобы различать местоположения, можно было бы дополнительно измерять количественно степень опасности по расстоянию от достигнутого Гензелем и Гретель местоположения до их дома. Перемещение одного камешка к другому означает не только простую смену местоположения в представлении вычисления, но и уменьшение опасности в представлении задачи, если достигнутое местоположение приблизилось к дому. Переходный характер знаков объясняет, почему фраза "он работал в CCH" означает, что он работал в госпитале на Чаринг-Кросс, а *не* он работал в "Госпиталь на Чаринг-Кросс", что не имело бы смысла, поскольку нельзя работать в названии госпиталя.

Осмысление означающих

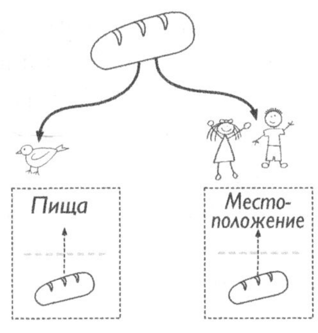
Знак, действующий на разных уровнях представления, служит примером означающего, связанного со многими означаемыми. Так, знак "1" в примере с рулеткой обозначает число "один" и ставку; камешки, используемые Гензелем и Гретель, представляют достигнутое местоположение и опасность; а совокупность камешков — путь и движение от опасности к безопасности. Таким образом, любое сокращение представляет название, а также означает понятие, обозначаемое этим названием.

Тем не менее одно означающее может также представлять разные, не связанные вместе понятия, и вполне возможно, что одно понятие может быть представлено разными, не связанными вместе означающими. Например, "10" означает число "десять" в десятичном представлении и число "два" — в двоичном представлении. А число "два" представлено означающим "2" в десятичном представлении и означающим "10" в двоичном представлении. Безусловно, несколько представлений существует и на уровне представления задачи. Очевидно, что с помощью числа "один" можно представить и другие предметы, а не только ставку на черное на столе для игры в рулетку.



Оба эти явления хорошо известны языковедам. С одной стороны, термином *омоним* обозначаются два или больше разных обозначаемых понятий. Например, слово "ствол" может обозначать прямую часть дерева, вертикальную выработку на шахте или дуло огнестрельного оружия. А с другой стороны, два слова называются *синонимами*, если оно представляют одно и то же понятие. Примерами синонимов служат слова "друг" и "товарищ" или "пес" и "собака". В контексте вычисления омонимы поднимают ряд важных вопросов.

Так, если одно означающее может представлять разные означаемые, то какое из их представлений фактически действует, когда применяется означающее? Нет ничего удивительного в том, что представление, вызываемое означающим, зависит от контекста, в котором они применяется. Так, если задать вопрос "Что означает сокращение «CCH»", то в нем спрашивается смысловое значение данного сокращения, т.е. название "Charing Cross Hospital". А если задать вопрос "Приходилось ли вам бывать в «CCH»", то в нем употребляется ссылка на госпиталь, а следовательно, выбирается второе представление. Кроме того, означающее "10" представляет число "десять" или "два" в зависимости от того, применяется ли десятичное или двоичное представление этого числа. История Гензеля и Гретель также демонстрирует особое значение контекста применения для того, чтобы знак мог играть особую представительскую роль. Так, если бы камешки лежали перед домом Гензеля и Гретель, они не представляли бы ничего особенного. Но



как только они оказались намеренно разбросанными в лесу, они стали представлять места для поиска обратного пути домой.

Одно и то же означающее может иметь разное смысловое значение для разных агентов, их интерпретирующих. Например, хлебные крошки, использовавшиеся Гензелем и Гретель во второй раз, означали для них места для поиска обратного пути домой. Но птицы в лесу интерпретировали хлебные крошки как пищу. Обе интерпретации хлебных крошек имеют смысл и пригодны как с точки зрения Гензеля и Гретель, так и с точки зрения птиц. Не требуется особого воображения, чтобы заметить, что омонимы способны вызывать трудности в алгоритмах, поскольку они, по существу, представляют неоднозначность, которую приходится каким-то образом разрешать. Почему же иногда требуется (или даже необходимо), чтобы одно имя представляло разные значения в алгоритме? О том, как разрешаются очевидно возникающие неоднозначности, рассказывается в главе 13, "Все дело в интерпретации".

И наконец, вполне возможно и ошибиться в представлении, связав с означающим неверное означаемое. Характерным тому примером служит вывод Шерлока Холмса о том, что посвящение на тросточке доктора Мортимера представляло его отставку, тогда как на самом деле оно представляло его женитьбу (рис. 3.1). Эта неверная интерпретация обсуждается и разрешается по ходу повести *Собака Баскервилей.*

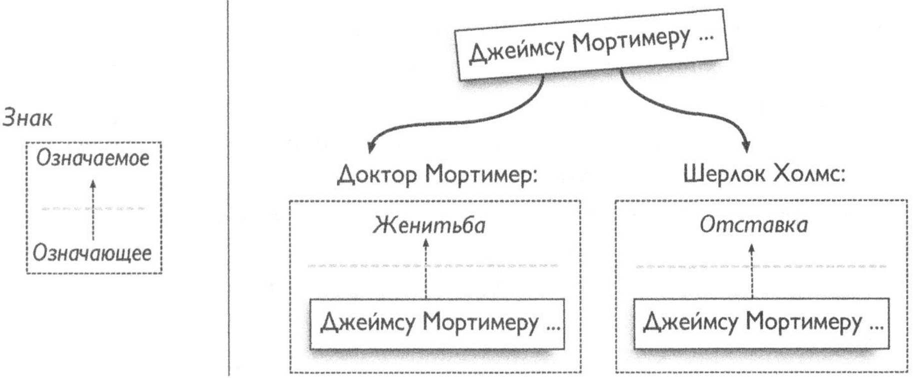


Рис. 3.1. Знаки служат основанием для представления. Знак состоит из означающего, представляющего некоторое понятие, которое называют означаемым. Одно означающее может обозначать разные понятия

Правильность интерпретации имеет решающее значение для вычисления. Ведь если вычисление примет неверное представление в качестве входных данных, то

оно даст неверные результаты. Это обстоятельство иногда еще называется следующим образом: "какой вход, такой и выход". И неудивительно, что вычисления, основанные на неверных входных данных, приводящих к неверным результатам, могут иметь опустошительные последствия. Так, если бы камешки представляли путь в самую глубину леса, то никакой правильный алгоритм поиска пути не помог бы Гензелю и Гретель найти обратную дорогу домой, и они погибли бы в лесу.

Суровым напоминанием о важности тщательного выбора представлений служит потеря космического летательного аппарата Mars Climate Orbiter, запущенного НАСА в 1998 году для исследования климата и атмосферы на Марсе. Выполняя маневр коррекции своей орбиты, этот космический летательный аппарат слишком сблизился с поверхностью планеты и разбился. Причиной неудачного маневра стало применение двух разных представлений чисел в управляющей программе и космическом летательном аппарате. В частности, управляющая программа вычислила силу тяги в британских (неметрических) единицах измерения, а регулятор тяги ожидал получения входных данных в метрических единицах измерения. Такая ошибка в представлении обошлась в 655 миллионов долларов США. Методы, позволяющие избежать подобного рода ошибок, обсуждаются в главе 14, "Волшебный тип".

Три способа обозначения

Принимая во внимание точность представлений, как же установить взаимосвязь между знаком и тем, что он обозначает? Это можно сделать самыми разными способами и соответственно описать знаки. Логик, ученый и философ Чарльз Сандерс Пирс (Charles Sanders Peirce) определил три разных вида знаков.

Во-первых, *графический знак*, или *пиктограмма*, представляет объект, исходя из его сходства или подобия с самим объектом. Например, рисунок может представлять человека, подчеркивая особые его черты. Очевидным примером графического представления в повести *Собака Баскервилей* служит портрет сэра Хьюго Баскервиля, представляющий его через его подобие. Человек на портрете очень похож на убийцу, а следовательно, служит еще одним примером означающего, которое может обозначать разные означаемые. Тот факт, что портрет, по существу, содержит два знака, помогает Шерлоку Холмсу раскрыть преступление. К другим примерам относятся сокращения "CCH" и "MRCS", которые служат для представления тех фраз, которые они обозначают. Здесь подобие устанавливается символами, общими для фразы и сокращения. И наконец, Шерлок Холмс обращается к карте девонширского болота, чтобы выяснить место, где произошло убийство. Эта карта знаковая, поскольку ее характерные черты (дороги, реки, лес и пр.) напоминают (главным образом своей формой и расположением) те объекты, которые они представляют.

Во-вторых, *указатель*, или *индекс*, представляет объект через некоторую подобную закону взаимосвязь, позволяющую наблюдателю указателя вывести суждение об объекте через эту взаимосвязь. Очевидным тому примером служит флюгер, по положению которого можно выяснить направление ветра. К другим примерам относятся всевозможные приборы, сконструированные в качестве указателей физических характеристик (температуры, давления, скорости и прочего). Поговорка "Нет дыма без огня" основана на том, что дым указывает на огонь. Указательный знак определяется тем объектом, который он обозначает через подобную закону взаимосвязь между ними. Другими важными в *Собаке Баскервилей* (и в других повестях о Шерлоке Холмсе) указательными знаками являются следы. Например, следы собаки, найденные рядом с покойным сэром Чарльзом Баскервилем, указывают на громадную гончую собаку. А тот факт, что ее следы обрываются на некотором расстоянии от сэра Чарльза, был интерпретирован Шерлоком Холмсом как указание на то, что собака не вступала в физический контакт с ним. Особая форма следов сэра Чарльза указывает на то, что он убегал от собаки. Еще одним указателем служит количество пепла от сигары сэра Чарльза, найденное на месте преступления и указывавшее на время, которое он ждал на месте своей гибели. Между прочим, Пирс сам пользовался примером убийцы и его жертвы в качестве примера указателя. Применительно к данной истории это означает, что покойный сэр Чарльз служит указателем его убийцы.

В-третьих, *символ* представляет объект лишь условно; никакого подобия или подобной закону связи в таком представлении не наблюдается. А поскольку связь между означающим и означаемым совершенно произвольна, то создатель и пользователь знака должны согласовать определение и интерпретацию знака, чтобы он мог действовать. Большинство современных языков являются символическими. Тот факт, что слово "дерево" обозначает дерево, нельзя вывести логически, но придется запомнить. Аналогично тот факт, что обозначение "11" служит символом десятичного числа "одиннадцать" и бинарного числа "три", а камешки — символами местоположения, является условием. Из тех знаков, которые упоминались в повести *Собака Баскервилей*, сокращения "MRCS" и "CCH" являются символами, если служат для представления членства в хирургическом обществе и госпиталя соответственно, поскольку они совершенно не похожи ни какую подобную закону взаимосвязь и не являются ее результатом. А символ "2704" представляет кеб, за которым следят Холмс и Ватсон, чтобы выявить лицо, подозреваемое в угрозах наследнику Баскервилей сэру Генри.

Систематическое применение представлений

Проведя различие между пиктограммой, указателем и символом, можно теперь рассмотреть, каким образом употребляются в вычислении эти разные

формы представления. А поскольку вычисление действует через преобразование представлений, то разные механизмы представления пиктограммы, указателя и символа приводят к разным формам вычисления.

Например, пиктограммы могут быть преобразованы таким образом, чтобы обнаруживать или скрывать конкретные свойства представляемого объекта, поскольку они представляют его через подобие. Так, в инструментальных средствах редактирования фотографий предоставляются многочисленные эффекты для систематического изменения фотографических изображений путем изменения цветов или искажения пропорций изображения. Такое вычисление, по существу, изменяет подобие графического знака. Другой метод вычисления с помощью графических представлений, имеющий непосредственное отношение к профессии Шерлока Холмса, состоит в создании реконструированных портретов или фотороботов подозреваемых по описаниям свидетелей. В этом случае свидетель описывает черты лица (например, форму и размер носа или цвет и длину волос), которые интерпретируются художником-криминалистом как инструкции для рисования портрета подозреваемого. Вычисление рисованного портрета подозреваемого является результатом алгоритма, который задается свидетелем и выполняется художником-криминалистом. Принимая во внимание алгоритмический характер такого метода, не стоит удивляться, что он был автоматизирован.

Данный метод обязан своим происхождением Альфонсу Бертильону, принципы антропометрии (т.е. измерения частей тела человека) которого были взяты парижской полицией в 1883 году на вооружение в качестве метода опознания преступников. Он создал систему классификации черт лица, которая первоначально использовалась для поиска отдельных подозреваемых в крупной коллекции совмещенных фотографий преступников в профиль и анфас. Данный метод служит примером вычисления, в котором рисованные портреты используются для важной алгоритмической задачи поиска (см. главу 5, “Поиск идеальной структуры данных”). Шерлок Холмс отдавал должное трудам Бертильона, хотя и не слишком высоко отозвался о нем в повести *Собака Баскервилей.* Еще одной разновидностью вычисления с помощью рисованных портретов является процесс логического заключения о личности подозреваемых на основании фотороботов. При таком вычислении, по существу, устанавливается знак, в котором означающим является фоторобот, а означаемым — подозреваемый. Знак устанавливается и в том случае, если подозреваемый опознается с помощью рисованного портрета или фотографии. Именно таким образом Шерлок Холмс опознает убийцу по портрету сэра Хьюго Баскервиля.

В качестве примера вычисления с помощью указательного знака можно напомнить о карте девонширского болота. Чтобы найти место, где конкретная дорога пересекает реку, Шерлок Холмс мог вычислить пересечение двух линий, представляющих реку и дорогу соответственно. На самом деле данная точка в

представлении карты, по существу, уже вычислена, и поэтому остается лишь считать ее с карты. Если допустить, что карта составлена точно, то найденная в итоге точка должна представлять искомое место [4]. Шерлок Холмс мог бы вычислить длину пути через болото по карте, используя ее масштаб, а также время, необходимое для преодоления этого пути. Но такой метод оказывается пригодным лишь в том случае, если карта составлена в масштабе. В вычислениях по указателям используется подобная закону взаимосвязь между знаком и означаемым для перехода от одного означаемого к другому через преобразование указателя.

Вычисления с помощью символов являются, вероятно, самыми распространенными в информатике, поскольку символы делают возможным представление произвольных задач. Наиболее очевидные вычисления, основанные на символах, подразумевают употребление чисел и арифметических операций, и пример тому можно найти первом приключении Шерлока Холмса из повести *Этюд в багровых тонах* [5], когда он вычисляет рост подозреваемого по длине его шага. Это очень простое вычисление, состоящее только из одной операции умножения, а соответствующий алгоритм — лишь из одного шага.

Более интересными с вычислительной точки зрения являются попытки Шерлока Холмса расшифровать зашифрованные сообщения. Так, в повести *Долина страха* он пытается разобрать присланное ему сообщение, начинающееся следующим шифром: 534 C2 13 127 36... Этот шифр обозначает сообщение. Первым делом Шерлоку Холмсу нужно было выяснить алгоритм, применявшийся для шифрования, поскольку это позволило бы ему расшифровать сообщение. И он приходит к выводу, что число 534 должно обозначать номер страницы какой-то книги, код C2 — второй столбец, а последующие числа — слова в данном столбце.

Но является ли шифр действительно символическим знаком? Шифр генерируется алгоритмом из заданного сообщения, и поэтому в алгоритме шифрования, по-видимому, устанавливается подобная закону взаимосвязь между сообщением и его шифром. Следовательно, шифр является не символом, а указателем. Данный пример демонстрирует еще один способ употребления знаков в вычислении. Если интерпретация дает означаемое для заданного означающего, то алгоритм для генерирования значений указателей действует в обратном направлении, получая означающее из заданного означаемого.

Из всего сказанного выше можно сделать следующий вывод: представление служит основанием для вычисления. Его характер и основные свойства можно понять через призму символов. И подобно тому, как основанием для произведений искусства могут служить самые разные материалы (глина, мрамор, краски и пр.), вычисление может основываться на разных представлениях. Особое значение представления было подчеркнуто в главе 1, “Путь к пониманию вычислений”, следующим образом: вычисление невозможно без представления.

В офисе

Представьте, что, прибыв на работу в свой офис, вы получили задание обработать пачку документов. Прежде чем приступить к работе, вы должны выбрать порядок, в котором следует обрабатывать документы, а также способ обращения с набором, чтобы обеспечить этот порядок. Эти вопросы уместны и в других контекстах. Представьте, например, автомеханика, которому приходится ремонтировать самые разные модели автомашин, или врача, принимающего разных пациентов в приемной.

Порядок, в котором обрабатываются элементы коллекции (набора документов, автомашин или людей), нередко определяется определенным правилом или принципом. Например, принцип "первым пришел, первым обслужен" означает, что элементы обрабатываются в порядке их поступления. Такой принцип требует, чтобы коллекция обслуживалась по определенной схеме, определяющей взаимозависимость между операциями доступа, ввода и удаления элементов. В информатике коллекция, доступная по особой схеме, называется *типом данных*, а тип данных, соблюдающий принцип "первым пришел, первым обслужен", — *очередью.*

Несмотря на то что очереди широко применяются для определения порядка, в котором обрабатываются элементы коллекции, им находится и другое применение. Так, если элементы обрабатываются с учетом определенного приоритета, а не по порядку их поступления, то такая коллекция называется типом данных *очереди с приоритетами.* Некоторые документы в вашем учреждении могут быть именно такого типа, например срочный запрос, требующий немедленного ответа; служебная записка, на которую необходимо ответить до обеда; или предложение, которое должно быть отправлено в тот же день. К другим примерам таких очередей относятся пациенты в пункте неотложной помощи, обслуживаемые в порядке серьезности симптомов их болезни, или пассажиры, которые могут совершать посадку в самолет в соответствии со своим положением часто летающего пассажира.

Еще одной схемой доступа служит обработка запросов в порядке, противоположном порядку их поступления. И хотя такая ситуация может поначалу показаться необычной, она возникает довольно часто. Допустим, что вы работаете над своей налоговой декларацией. Вы можете начать с основной налоговой формы, но когда в одном из полей формы вам потребуется указать свои налоговые вычеты, то для этого вам придется заполнить сначала другую форму, а для нее — извлечь

соответствующие счета и сложить оплаченные суммы. Таким образом, вам придется обработать три вида документов в порядке, обратном их поступлению: сначала ввести суммы из счетов и отложить их в сторону, затем — заполнить форму налоговых вычетов и наконец — вернуться к основной налоговой форме. Коллекция, элементы которой обрабатываются в подобном порядке, называется типом данных *стека*, поскольку она действует подобно стопке[2] блинов: первым съедается тот блин, который был положен в стопку последним, а последним — блин, который был положен в стопку первым. Схема доступа, описываемая типом данных стека, встречается в самых разных задачах: от выпечки десертов до сборки мебели. Например, яичные белки должны быть взбиты перед добавлением в смесь ингредиентов для приготовления кекса, а выдвижной ящик собран перед его вставкой в шкаф.

Знание схемы доступа для обработки элементов коллекции приводит к следующему вопросу: как лучше всего организовать элементы для поддержки данной схемы доступа. Такая организация элементов называется *структурой данных*. Рассмотрим в качестве примера тип данных очереди, чтобы выяснить способы его реализации. Если на вашем рабочем столе имеется достаточно свободного места (что может оказаться весьма оптимистичным предположением), вы можете упорядочить документы, сложив их в одну линию и добавляя их с одной стороны, и убирая с другой. Время от времени вы сдвигаете все документы так, чтобы свободное место переместилось на тот конец стола, с которого добавляются новые документы. Это можно сравнить с выстраиванием людей друг за другом в кофейне. Каждый человек присоединяется к очереди с одного конца и продвигается к ее началу вслед за всеми теми людьми, которые ее покидают. Во многих учреждениях применяется другой метод: все посетители получают свои номера и ожидают вызова по очереди. Вы можете воспользоваться такой системой для обработки документов в своем учреждении, наклеив на них бумажные наклейки с порядковыми номерами.

Последовательность документов на вашем рабочем столе и очередь в кофейне называются структурой данных *списка*. В данном случае физическое расположение людей в очереди обеспечивает требуемый порядок ее организации. А метод присваивания порядковых номеров не требует физического поддержания порядка среди людей или документов в очереди. Их можно расположить где угодно, поскольку порядковые номера задают правильный порядок. Такой способ присваивания элементам нумерованных мест называется структурой данных *массива*. Помимо нумерованных мест, распознаваемых людьми по присвоенным им номерам, приходится вести два подсчета: один — для следующего доступного места, а другой — для следующего обслуживаемого номера.

---

[2] Дословно *stack* переводится на русский язык как "стопка". — *Примеч. ред.*

Представление коллекций в виде структур данных делает их доступными для вычислений. Выбор подходящей структуры данных имеет значение для эффективности алгоритмов, и на него порой оказывают влияние другие соображения (например, доступное пространство). Когда Шерлок Холмс сохраняет сведения об уголовном деле (например, в виде набора подозреваемых), он, по существу, пользуется типами и структурами данных. Поэтому мы будем и далее обращаться к повести *Собака Баскервилей* для пояснения этих понятий.

4 Записная книжка сыщика

Вычисление оказывается особенно полезным, когда приходится иметь дело с крупными массивами данных, которые невозможно обработать за несколько отдельных шагов. В подобных случаях подходящий алгоритм может обеспечить систематическую обработку данных, зачастую делая это весьма эффективно.

Знаки, обсуждавшиеся в главе 3, "Тайна знаков", демонстрируют, каким образом представление служит для обозначения отдельных фрагментов информации и как это представление становится частью вычисления. Например, передвижение Гензеля и Гретель между камешками означает, что они находятся в опасности до тех пор, пока не перейдут от последнего камешка к своему дому. Но даже если коллекции знаков сами являются знаками, то еще не ясно, как проводить вычисление с такими коллекциями. Если речь идет о Гензеле и Гретель, то отдельный камешек служит означающим одного места, а коллекция всех камешков обозначает путь от опасности к безопасности. Но как такая коллекция построена и как пользоваться ею систематически? В связи с обслуживанием коллекции данных возникают два следующих вопроса.

Во-первых, в каком порядке данные должны вводиться, находиться и удаляться из коллекции? Ответ на этот вопрос зависит, конечно, от той вычислительной задачи, в которой задействована коллекция. Но при этом можно наблюдать повторение отдельных схем доступа к элементам коллекции. Такие схемы доступа к данным называются *типами данных*. Например, Гензель и Гретель обходят использованные ими камешки в обратном порядке по сравнению с тем, как они были положены. И такая схема доступа называется *стеком*.

И во-вторых, как следует хранить коллекцию, чтобы добиться наибольшей эффективности поддерживаемой в ней схемы доступа, или типа данных? Ответ на этот вопрос зависит от самых разных факторов. Например, сколько элементов должно храниться в коллекции? Известно ли это количество заранее?

Сколько места требуется для хранения каждого элемента в коллекции и одинаковы ли размеры всех элементов? Любой конкретный способ хранения коллекции называется *структурой данных*. Благодаря структуре данных коллекция оказывается пригодной для вычисления. Один тип данных может быть реализован разными структурами данных, а это означает, что конкретная схема доступа может быть реализована через разные способы хранения данных. Структуры данных различаются эффективностью поддержки отдельных операций над коллекцией. Кроме того, в одной структуре данных могут быть реализованы разные типы данных. В этой главе рассматриваются несколько типов данных, реализующие их структуры данных, а также порядок их применения как части вычислений.

Главные подозреваемые

Если виновник преступления известен (возможно, имеются свидетели и признание), то профессиональных способностей Шерлока Холмса для расследования уголовного дела не потребуется. Но если имеется несколько подозреваемых, то придется проследить их мотивы, алиби и прочие имеющие значение сведения для более подробного расследования уголовного дела.

В повести *Собака Баскервилей* к числу подозреваемых относятся доктор Мортимер, Джек Степлтон и его предполагаемая сестра Берил, которая на самом деле является его женой, мистер Френкленд, супруги Берримор и слуги покойного сэра Чарльза Баскервиля. Прежде чем отправить доктора Ватсона с визитом в усадьбу Баскервиль-Холл, Шерлок Холмс дает ему инструкцию: сообщить обо всех фактах, но исключить мистера Джеймса Десмонда из числа подозреваемых. Когда же доктор Ватсон предлагает Холмсу исключить из числа подозреваемых и супругов Берримор, на это предложение Шерлок Холмс отвечает так:

Нет, нет, мы оставим их пока что в своем списке подозреваемых [1].

Этот краткий обмен мнениями демонстрирует две вещи. Во-первых, несмотря на то, что Шерлоку Холмсу ничего не известно о структурах данных, он все-таки пользуется одной из них, поскольку, по-видимому, ведет список подозреваемых. А ведь *список* — это простая структура для хранения элементов данных путем их связывания вместе. Список предоставляет отдельную форму доступа к этим элементам данных и манипулирования ими. И во-вторых, список подозреваемых не является статической сущностью. Он разрастается и сокращается по мере ввода новых подозреваемых или удаления отброшенных подозреваемых. Для ввода,

удаления или иного изменения элементов в структуре данных нужны алгоритмы, для выполнения которых обычно требуется больше одного шага. Но именно время их выполнения определяет, насколько некоторая структура данных пригодна для решения конкретной задачи.

Благодаря своей простоте и универсальности списки оказываются едва ли не самыми широко распространенными структурами данных не только в информатике, но и в других областях человеческой деятельности. Все мы постоянно пользуемся списками в форме перечня неотложных дел, списка литературы для чтения, перечня пожеланий и всевозможных табелей о рангах и классификаций.

Порядок следования элементов в списке имеет значение, и, как правило, они доступны в порядке очереди, начиная с одного конца списка и продолжая до другого. Списки зачастую составляются по вертикали, по одному элементу на строку, причем первый элемент располагается в начале списка. Но специалисты по информатике составляют списки по горизонтали, представляя их элементы слева направо и соединяя их стрелками, чтобы указать порядок следования элементов в списке [2]. Используя такое обозначение, список подозреваемых Шерлоком Холмсом можно представить следующим образом:

Мортимер → Джек → Берил → Селден → ...

В таком обозначении списка стрелки называются *указателями* и устанавливают явную связь между элементами, что приобретает особое значение, когда требуется решить, как обновить список. Допустим, что список подозреваемых Шерлоком Холмсом состоит из элементов

Мортимер → Берил

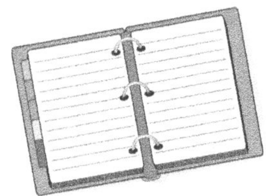
между которыми требуется внести Джека.

Если элементы расположены сверху вниз в виде вертикального списка и между ними отсутствует свободное место, то придется прибегнуть к какому-нибудь другому виду обозначений, чтобы уточнить положение нового элемента в списке. В качестве альтернативы можно просто составить новую копию всего списка. Но ведь это было бы излишней тратой времени и места. В худшем случае требующееся время и место составили бы квадрат от размера окончательного списка.

Указатели позволяют удобно записывать новые элементы везде, где найдется свободное место, и в то же время размещать их в нужной позиции в списке, связывая их с соседними элементами списка. Например, мы можем расположить Джека в средине списка, перенаправив исходящий от Мортимера указатель к Джеку, и добавить указатель от Джека к Берил.

В контексте рассматриваемой здесь истории порядок расположения подозреваемых в списке — произвольный и не имеет особого значения, но нетрудно заметить, что создание списка вынуждает нас выбирать определенный порядок расположения его элементов. Именно сохранение конкретного порядка расположения элементов является определяющей характеристикой списка.

Проверка элементов списка производится в определенном порядке, задаваемом самим списком. Следовательно, чтобы выяснить, является ли Селден подозреваемым, придется начать с начала списка и проверить все его элементы по очереди, следуя по указателям. И хотя элемент "Селден" можно непосредственно обнаружить в списке, эффективно это можно сделать лишь в относительно небольших списках. А поскольку наше поле зрения ограничено и мы не можем сразу распознать отдельный элемент в длинном списке, нам придется прибегнуть к поочередному обходу элементов списка.



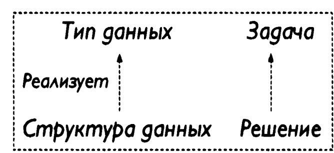
В качестве физической аналогии списка может служить кольцевой скоросшиватель, содержащий по одному листу бумаги на каждый элемент. Чтобы найти конкретный элемент в таком кольцевом скоросшивателе, придется просматривать по очереди отдельные страницы, а между страницами можно легко вставлять новые страницы.

Примечательное свойство списка состоит в том, что время, требующееся для поиска элемента, зависит от местоположения элемента в списке. В данном примере Селден будет обнаружен на четвертом шаге алгоритма. В общем, для поиска элемента, возможно, придется обойти весь список, поскольку этот элемент может оказаться последним в списке. При обсуждении времен*о*й сложности в главе 2, "От слов к делу: когда действительно происходит вычисление", такой алгоритм назывался *линейным*, поскольку времен*а*я сложность прямо пропорциональна количеству элементов в списке.

Как говорилось ранее, еще не ясно, составлен ли список подозреваемых Шерлоком Холмсом в показанном выше порядке, а то обстоятельство, что Селден следует в нем после Берил, ничего не означает, поскольку данный список служит лишь для запоминания только тех лиц, которые относятся к числу подозреваемых. Ведь самое главное, находится ли такое лицо в списке [3]. Но значит ли это, что список не является подходящим представлением для запоминания подозреваемых? Совсем нет. Это лишь означает, что список может содержать сведения (например, порядок следования элементов), которые не требуются для решения конкретной задачи. Такое наблюдение предполагает, что список является лишь одной из возможных структур данных для представления сведений о подозреваемых и что возможны другие представления, которыми можно было бы воспользоваться для той же самой цели, но при условии, что они поддерживают те же самые операции для ввода, удаления и поиска элементов. Эти операции выражают требования к тому, что следует сделать с данными.

Такие требования к данным, выражаемые через операции над коллекцией, называются в информатике *типом данных.* Требования к сведениям о подозреваемых состоят в способности вводить, удалять и находить элементы. И такой тип данных называется *множеством.*

Множества находят широкое применение, поскольку соответствуют утверждениям, связанным с задачей или алгоритмом. Например, множество подозреваемых соответствует утверждению "является подозреваемым", которое применимо к людям и может служить для подтверждения или опровержения таких заявлений, как "Селден подозревается", в зависимости от того, является ли лицо, к которому применяется утверждение, членом множества. В алгоритме прослеживания пути по камешкам, использованном Гензелем и Гретель, также употребляется утверждение, когда в нем предписывается "Найти сверкающий камешек, не посещавшийся прежде". В данном случае утверждением служит фраза "не посещавшийся прежде". Это утверждение применимо к камешкам и может быть представлено множеством, которое первоначально пусто и в которое вводятся камешки после их посещения.



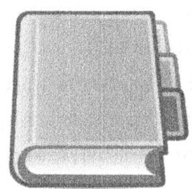
В то время как тип данных описывает требования к тому, что следует иметь возможность делать с данными, структура данных предоставляет конкретное представление для поддержки этих требований. Тип данных можно рассматривать как описание задачи управления данными, а структуру данных — как решение этой задачи. (Приведенные здесь мнемонические обозначения могут оказать помощь в запоминании смыслового значения этих двух понятий. В частности, тип данных описывает задачу, а структура данных — решение.) Тип данных служит более абстрактным описанием задачи управления данными, чем структура данных, и обладает тем преимуществом, что некоторые подробности могут быть оставлены без уточнения, что в конечном итоге может привести к более лаконичному и общему описанию. В рассматриваемой здесь истории из повести *Собака Баскервилей* тип данных множества отражает задачу ведения коллекции подозреваемых без раскрытия подробностей ее реализации. А в истории Гензеля и Гретель типа данных множества, предназначенного для запоминания посещенных камешков, оказывается достаточно для описания алгоритма. Но для того чтобы выполнить операции, предусматриваемые типом данных, компьютеру придется употребить конкретную структуру данных, определяющую порядок выполнения операций над представлением, предоставляемым этой структурой данных. И только тогда, когда для алгоритма выбрана конкретная структура данных, можно определить его временн*у*ю сложность.

В связи с тем что один и тот же тип данных может быть реализован разными структурами данных, возникает вопрос выбора подходящей структуры. Такая

структура данных должна реализовывать операции над типом данных с наилучшим показателем времени выполнения, чтобы как можно быстрее выполнить алгоритм, пользующийся этой структурой данных. Но сделать такой выбор не всегда легко, поскольку структура данных может поддерживать одни операции лучше, чем другие. Кроме того, у структур данных бывают разные требования к пространству. Это похоже на выбор транспортного средства для решения конкретной задачи перевозки. Так, велосипед является экологически чистым транспортным средством и не требует экономии расхода топлива в литрах на километры. Но это слишком медленное транспортное средство, к тому же не очень грузоподъемное, способное перевозить одного или двух человек на ограниченное расстояние. Для путешествия со многими спутниками на большие расстояния потребуется фургон или даже автобус, для перевозки крупных предметов — грузовик, а для удобной поездки — легковой автомобиль типа "седан" или даже спортивный автомобиль, если вы мужчина и вам под пятьдесят.

Если вернуться к вопросу о реализации типа данных множества, то двумя распространенными альтернативами спискам служат структуры данных *массива* и *бинарного дерева поиска.* Бинарные деревья поиска подробно обсуждаются в главе 5, "Поиск идеальной структуры данных", а здесь мы уделим основное внимание массивам.

Если список подобен кольцевому скоросшивателю, то массив подобен записной книжке с фиксированным количеством страниц, каждая из которых имеет особое обозначение. Отдельное поле структуры данных массива называется *элементом,* а идентификатор элемента — *индексом.* Обозначение (или индексация) элементов массива обычно осуществляется с помощью чисел, хотя для этой цели можно пользоваться буквами или именами, при условии, что с помощью такого идентификатора можно получать доступ к элементу массива, словно открывая записную книжку на нужной странице [4]. Примечательная особенность структуры данных массива состоит в том, что она обеспечивает очень быстрый доступ к каждому ее элементу в отдельности. Сколько бы ни было элементов в массиве, для доступа к любому из них требуется совершить лишь один шаг алгоритма. Если для выполнения некоторой операции требуется только один или несколько шагов алгоритма независимо от размера структуры данных, то говорят, что она выполняется в течение *постоянного,* или *константного, времени.*



Чтобы представить массив в виде записной книжки, допустим, что каждая ее страница имеет закладку, помеченную одним из возможных элементов массива. Следовательно, чтобы представить подозреваемых из повести *Собака Баскервилей,* необходимо пометить страницы записной книжки именами *всех* потенциальных подозреваемых, т.е. Мортимер, Джек и т.д. В такой записной книжке имеются также страницы для Десмонда и других лиц, которые могут быть в принципе отнесены к числу подозреваемых. Этим записная книжка

отличается от кольцевого скоросшивателя, содержащего только фактических подозреваемых. Чтобы ввести в число подозреваемых, скажем, Селдена, достаточно открыть страницу с пометкой “Селден” и поставить на ней какую-нибудь отметку (например, знак “плюс” или слово “да”). А для того чтобы удалить кого-нибудь из числа подозреваемых, также следует открыть страницу с именем этого лица, но снять отметку (написать знак “минус” или же слово “нет”). И наконец, чтобы выяснить, относится ли некто к числу подозреваемых, достаточно перейти на страницу с именем этого лица и проверить сделанную на ней отметку. По такому же принципу действует и массив. Мы обращаемся непосредственно к его элементам по индексу, чтобы прочитать или видоизменить хранящуюся в них информацию.

Главное отличие массивов от списков состоит в том, что обращаться к элементам массива можно непосредственно, тогда как список приходится просматривать от начала и до конца, чтобы найти нужный элемент (или достичь конца списка, если искомый элемент в нем отсутствует).

Конкретную страницу записной книжки можно открыть (или получить доступ к элементу массива) непосредственно, и поэтому все три операции (ввода, удаления и поиска подозреваемых) могут быть выполнены в течение постоянного времени, т.е. оптимально, поскольку сделать это быстрее, вероятно, нельзя. Для проверки и удаления подозреваемых из структуры данных списка требуется линейное время, и поэтому структура данных массива легко одерживает в этом отношении верх[1]. Так зачем мы вообще ведем речь о списках?

Недостаток массивов заключается в их фиксированном размере. Это означает, что у записной книжки имеется конкретное количество страниц, которое со временем не может увеличиваться. Из этого следуют два важных вывода. Во-первых, записная книжка должна быть выбрана достаточно большой, чтобы содержать записи обо всех возможных подозреваемых, сделанные с самого начала, даже если многие из них так и не станут подозреваемыми. Следовательно, нам придется напрасно тратить немало свободного места и носить с собой огромную записную книжку с сотнями или даже тысячами страниц потенциальных подозреваемых,

[1] В случае массива удаление элемента как таковое выполняется за линейное время, поскольку требует перенумерации всех последующих листов описываемой записной книжки и пометки их новыми индексами (быстро выполняется только операция, помечающая данный элемент как удаленный). Удаление же из списка уже найденного элемента выполняется за константное время; линейного времени требует удаление элемента по его значению (имени подозреваемого), так как включает поиск элемента. Удаление же сводится к простому перенаправлению указателя. — *Примеч. ред.*

хотя в действительности перечень подозреваемых может оказаться весьма скромным. В любой момент времени этот перечень вполне может содержать меньше десяти подозреваемых. Это также означает, что подготовка указателя на обрезе записной книжки с самого начала может отнять немало времени, поскольку для этого придется написать имя каждого потенциального подозреваемого на отдельной странице. И во-вторых, в начале расследования может быть совсем не ясен весь круг потенциальных подозреваемых, и это еще более серьезное затруднение. В частности, новые потенциальные подозреваемые могут становиться известными по мере развития истории, как это, собственно, и происходит в повести *Собака Баскервилей.* Именно этот недостаток информации препятствует использованию записной книжки, поскольку ее нельзя инициализировать.

Эта слабая сторона неуклюжего массива является в то же время сильной стороной проворного списка, который может со временем разрастаться или сокращаться по мере надобности, но никогда не хранить больше элементов, чем требуется. Выбирая структуру данных для реализации типа данных множества, необходимо принимать во внимание следующий компромисс. Массив обеспечивает очень быструю реализацию операций над множеством, но потенциально напрасно расходует свободное пространство, и поэтому может оказаться непригодным для всех возможных ситуаций. Список же расходует свободное пространство более эффективно, чем массив, и поэтому может оказаться пригодным при любых обстоятельствах, хотя и реализует некоторые операции менее эффективно. Это положение наглядно показано на рис. 4.1.

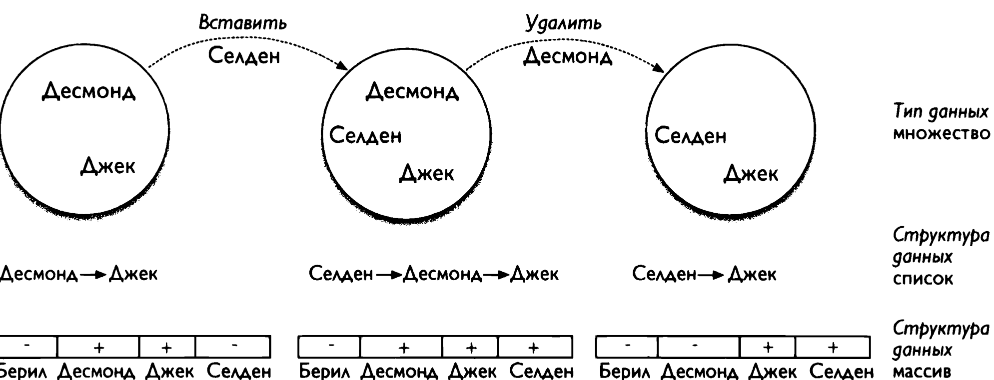


Рис. 4.1. Тип данных может быть реализован разными структурами данных. Чтобы ввести элемент в список, достаточно указать его в начале списка, но для его удаления придется обойти весь список, чтобы обнаружить его там. Операции ввода и удаления в массиве выполняются с непосредственным доступом к элементам массива по индексу, где соответственно изменяется отметка. Массивы допускают более быстродействующую реализацию типа данных, но списки более эффективно расходуют свободное пространство

Сбор сведений

Выявление подозреваемых — это лишь первый шаг в раскрытии тайны убийства. Чтобы сузить круг подозреваемых, Шерлоку Холмсу и доктору Ватсону необходимо собрать конкретные сведения о них, в том числе мотивы убийства и потенциальные алиби. Например, сведения о Селдене включают в себя тот факт, что он беглый каторжник. Все эти дополнительные сведения о каждом подозреваемом в отдельности должны быть сохранены вместе. Если бы Шерлок Холмс пользовался записной книжкой, ему пришлось бы вводить сведения о каждом подозреваемом лице на специально зарезервированной для него странице.

Операции над типом данных множества не позволяют этого сделать, хотя достаточно внести несколько мелких изменений в операции ввода и поиска элементов, чтобы это стало возможным. Во-первых, операция ввода элементов принимает следующие два фрагмента информации: *ключ* для обозначения информации, а также дополнительные сведения, связанные с этим ключом. Ключом к сведениям о подозреваемом служит его имя. И во-вторых, операции поиска и удаления подозреваемых будут принимать в качестве входных данных только ключ. При удалении имя подозреваемого лица и все дополнительные сведения о нем будут удалены. А при поиске подозреваемого лица по его имени в качестве результата будут возвращены хранящиеся о нем сведения.

Это незначительное, но очень важное расширение типа данных множества называется *словарем*, поскольку, как и настоящий словарь, он позволяет искать информацию по ключевому слову, как это сделал Шерлок Холмс в начале повести *Собака Баскервилей*, чтобы найти сведения о профессиональном прошлом доктора Мортимера в медицинском словаре. Словарь можно рассматривать как собрание знаков, а каждый ключ — как означающее для хранящейся в нем информации. Тип данных словаря отличается от традиционного печатного словаря в двух отношениях. Во-первых, содержимое печатного словаря фиксировано, тогда как тип данных словаря можно изменять, вводя новые, удаляя устаревшие и обновляя существующие определения. И во-вторых, статьи в печатном словаре отсортированы в алфавитном порядке по ключам, тогда как для типа данных словаря это формально не требуется. Статьи в печатном словаре требуют упорядочения, ведь иначе непосредственный доступ к конкретным страницам словаря будет затруднен из-за большого количества статей. Для доступа к каждой странице печатного словаря указатель на обрезе должен был бы содержать слишком много статей и оказался бы практически непригодным, поэтому для поиска статей в словаре по определенному алгоритму служат отсортированные ключи (см. главу 5, “Поиск идеальной структуры данных”).

Оба ограничения, присущие печатным словарям, т.е. потребность в отсортированных ключах и фиксированный объем и содержимое, не распространяются

на электронные словари. Широко известным примером динамического словаря служит Википедия [5], которой можно не только пользоваться; можно ее расширять и обновлять в ней информацию. В действительности содержимое Википедии составляется ее пользователями и служит замечательным примером удачного использования коллективных ресурсов и свидетельством силы сотрудничества. Если бы Шерлок Холмс и доктор Ватсон расследовали уголовное дело из повести *Собака Баскервилей* в настоящее время, они могли бы воспользоваться редактируемыми страницами формата *вики* [6] для хранения сведений о подозреваемых и ходе расследования вместо того, чтобы обмениваться письмами.

Динамический характер словаря не ограничивается только вводом и удалением сведений о подозреваемых, позволяя также обновлять эти сведения. Например, тот факт, что беглый каторжник Селден приходится братом Элизе Берримор, не был известен, когда он стал подозреваемым, и поэтому данный факт пришлось бы впоследствии добавить к статье о Селдене, уже существующей в словаре. Но как это сделать? Если в нашем распоряжении имеются лишь три операции — для ввода, удаления и поиска статей в словаре, то как нам обновить сведения после того, как они сохранены в словаре по заданному ключу? Мы можем добиться этого, объединив операции следующим образом: сначала найти нужную статью по заданному ключу, затем извлечь из нее сведения, видоизменить их должным образом, удалить статью из словаря и, наконец, ввести снова статью в словарь, но уже с обновленными сведениями.

Аналогичным образом новыми операциями может быть дополнен тип данных множества. Так, если бы Шерлок Холмс завел множество людей, которым была бы выгодна смерть сэра Чарльза, ему пришлось бы дополнять это множество мотивами каждого из подозреваемых. Для этого он мог бы вычислить пересечение множества заинтересованных лиц с множеством подозреваемых лиц. А возможно, ему потребовалось бы выявить новых подозреваемых, определив заинтересованных лиц, которые отсутствуют во множестве подозреваемых. И для этого ему пришлось бы вычислить разность двух множеств. Если допустить, что у типа данных множества имеется операция для сообщения обо всех элементах множества, то по алгоритму, вычисляющему пересечение или разность двух множеств, было бы достаточно обойти все элементы одного множества и проверить каждый из них, относится ли он и к другому множеству. Если это так, то данная операция сообщит о проверяемом элементе как о результате пересечения обоих множеств, а иначе — как о результате разности обоих множеств. Подобные вычисления весьма распространены, поскольку соответствуют сочетанию утверждений. Например, пересечение множеств подозреваемых и заинтересованных лиц соответствует утверждению "подозреваемый *и* заинтересованный", а разность множеств заинтересованных и подозреваемых лиц — утверждению "заинтересованный *и не* подозреваемый".

И наконец, для реализации словарей нам потребуется структура данных для их фактического вычисления. Поскольку тип данных словаря отличается от типа данных множества только связыванием с элементами дополнительных сведений, то большинство структур данных для множеств, включая массивы и списки, может быть расширено и до реализации словарей. Это справедливо для всех тех структур данных, которые явно представляют каждый элемент множества, поскольку в этом случае достаточно просто добавить дополнительную информацию в ключ. Это также означает, что любую структуру данных, предназначенную для реализации словаря, можно использовать для реализации множества — по заданному ключу можно просто хранить пустой или бессмысленный фрагмент информации.

Когда порядок имеет значение

Как упоминалось в главе 3, "Тайна знаков", вычисление может быть только таким же хорошим, как и представление, которым оно оперирует. Следовательно, Шерлоку Холмсу и доктору Ватсону хотелось бы, чтобы множество подозреваемых точно отражало состояние их расследования. В частности, им нужно, чтобы данное множество было как можно меньшим, чтобы не тратить понапрасну силы на ложные следы [7], но в то же время оно должно быть достаточно большим, чтобы убийца не остался ненайденным. А в остальном порядок, в котором подозреваемые вводятся или удаляются из множества, не имеет для них особого значения.

Для решения других задач порядок следования элементов в представлении данных все же имеет значение. Рассмотрим в качестве примера наследников покойного сэра Чарльза. Отличие первоочередного от второочередного права наследования оценивается в миллион фунтов стерлингов. И эти сведения не только важны для определения тех лиц, которые от этого разбогатеют или не разбогатеют, но и дают Шерлоку Холмсу и доктору Ватсону ключи к пониманию потенциальных мотивов подозреваемых ими лиц. В действительности убийца Степлтон стоит вторым в ряду наследников, и поэтому пытается убить сначала сэра Генри как первого наследника. Несмотря на то что правопреемство имеет значение, порядок наследования не определяется моментом, когда люди входят в число наследников. Например, когда у завещателя рождается ребенок, он не становится последним в ряду наследников, а получает преимущественное право на наследство по сравнению, скажем, с племянниками завещателя. Тип данных, в котором порядок следования элементов определяется не временем их поступления, а каким-то иным критерием, называется *очередью с приоритетами*. Название этого типа данных указывает на то, что положение элемента в очереди определяется некоторым приоритетом, например родственными отношениями завещателя с наследниками или серьезностью травм пациентов в пункте неотложной помощи.

Если же время поступления определяет положение элемента в коллекции, то такой тип данных называется *очередью*, если элементы удаляются в том порядке, в каком были введены, или *стеком*, если они удаляются в обратном порядке. Очередь — это то, через что вы проходите в супермаркете, кофейне или при досмотре багажа в аэропорту, становясь в очередь с одного конца и покидая ее с другого. Люди в очереди обслуживаются (и затем покидают ее) в том порядке, в каком они к ней присоединились. Следовательно, тип данных очереди устанавливает следующее правило: "первым пришел, первым обслужен". Такой порядок поступления и покидания очереди отдельными элементами иначе называется "FIFO" (first in, first out) — "первым пришел, первым ушел".

Стек элементы покидают в обратном порядке по сравнению с порядком их вхождения в стек. Характерным примером тому служит стопка книг на столе. Верхняя книга, положенная в стопку последней, должна быть взята из нее первой, прежде остальных книг, имеющихся в стопке. Если у вас место в самолете у окна, вы, по существу, находитесь на дне стека, когда приходится выбираться из своего ряда, чтобы сходить в туалет или отправиться на выход после посадки. Пассажиру, сидящему посредине ряда, придется покинуть свое место прежде вас, хотя ему и нужно будет ждать, пока встанет пассажир, который сидит на крайнем месте в ряду. Он сел на свое место последним в ряду и покинет ряд первым подобно элементу на вершине стека. Еще одним примером стека служит порядок, в котором Гензель и Гретель разбрасывают и обходят камешки. И если они пользуются усовершенствованным алгоритмом, чтобы не ходить кругами, то последний брошенный ими камешек будет поднят первым.

Применение типа данных стека кажется поначалу необычным способом обработки данных, но на самом деле стеки очень удобны для хранения данных в организованном порядке. Так, стековое расположение камешков позволяет Гензелю и Гретель систематически возвращаться на то место, где они были перед текущим и в конечном итоге найти обратный путь домой. Аналогичным образом Тезей вышел из лабиринта Минотавра с помощью нити Ариадны, сначала разматывая нить и как бы добавляя ее по сантиметрам в стек, а затем сматывая нить и словно извлекая ее в том же количестве из стека. Если наша повседневная деятельность прерывается, скажем, телефонным звонком, а тот, в свою очередь, стуком в дверь, то мы сохраняем все эти задачи в уме, как в стеке, и, завершив первой последнюю из них, возвращаемся к тому, на чем была прервана предыдущая задача. Поэтому порядок, в котором элементы поступают и покидают стек, называется "LIFO" (last in, first out) — "последним пришел, первым ушел". А порядок, в котором элементы поступают и покидают очередь с приоритетами, именуется "HIFO" (highest in, first out) — "с наивысшим (приоритетом) пришел, первым ушел".

Как и для типов данных множества и словаря, хотелось бы знать, какие именно структуры данных лучше всего подходят для реализации типов данных очереди

(обычной или с приоритетами) и стека. Нетрудно заметить, что стек можно легко реализовать с помощью списка, всегда добавляя и удаляя элементы в начале списка, т.е. фактически реализуя характерный для стека порядок LIFO, на что требуется лишь константное время. Аналогично, добавляя элементы в конец списка и удаляя их из начала списка, можно реализовать порядок FIFO, характерный для очереди. Кроме того, очереди и стеки можно реализовать и с помощью массивов.

Очереди и стеки сохраняют порядок следования элементов до тех пор, пока остаются в этих структурах данных, а следовательно, демонстрируют предсказуемый порядок, в котором их покидают элементы. Стоять в очереди на досмотр багажа в аэропорту обычно надоедает. Намного интереснее, когда люди проходят досмотр без очереди или через специальные коридоры для обслуживания приоритетных пассажиров. Такое поведение отражается в типе данных очереди с приоритетами.

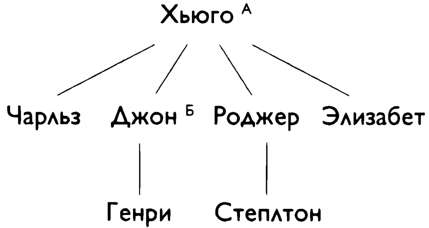
Передается по наследству

Наследники сэра Чарльза Баскервиля дают отличный пример очереди с приоритетами. Критерием приоритетности, определяющим положение каждого наследника в очереди, является дальность родства с покойным сэром Чарльзом. Но как определить эту дальность родства? Обычное простое правило наследования гласит: дети покойного наследуют от него в первую очередь, в порядке их возраста. Так, если наследуется само состояние покойного, то данное правило подразумевает следующее: если у покойного нет детей, то следующими наследниками становятся его старшие единокровные родственники (т.е. родные братья и сестры), а после них — их дети. А если у покойного нет родных братьев и сестер, то наследниками становятся его старшие дядья и тетки и их дети и т.д.

Данное правило можно продемонстрировать алгоритмически, представив всех членов семьи структурой данных *дерева*, отражающей их отношения типа "предшественник–наследник". В таком случае алгоритм обхода дерева позволяет составить список членов семьи, в котором положение каждого человека определяет приоритет его наследования. Этот приоритет может также служить критерием для очереди с приоритетами, состоящей из наследников. Но если имеется полный и правильно отсортированный список наследников, то зачем вообще нужна очередь с приоритетами? Она действительно не нужна, но может потребоваться в том случае, если полное генеалогическое дерево заранее неизвестно или же если оно изменяется, например, при рождении детей. В последнем случае генеалогическое дерево изменится, а список, вычисленный из прежнего дерева, не будет отражать правильный порядок наследования.

Как следует из повести *Собака Баскервилей*, у старшего Хьюго Баскервиля было четверо детей, и старшим из них был Чарльз, который и унаследовал от Хьюго. Следующим по старшинству был брат Джон, у которого был сын Генри,

а у младшего брата Роджера — сын, которым был Степлтон. В данной истории упоминается, что у Хьюго Баскервиля была еще дочь Элизабет, которая, как следует предположить, была младшим ребенком в семье. Имена в дереве называются *узлами*, и если узел Б (с именем Джон) соединяется с расположенным выше него узлом А (с именем Хьюго), то узел Б называется *потомком* узла А, тогда как узел А — *родителем* узла Б, как в генеалогическом дереве. Первый сверху узел дерева, у которого отсутствуют родители, называется *корневым*, а узлы, не имеющие потомков, — *листьями*.



Применительно к семейству Баскервилей правило наследования гласит, что от Хьюго должны по порядку наследовать Чарльз, Джон, Роджер и Элизабет. Но поскольку это правило также гласит, что дети должны наследовать прежде единокровных родственников, то Генри наследует прежде Роджера, а Степлтон — прежде Элизабет. Иными словами упорядоченный список наследников должен выглядеть следующим образом:

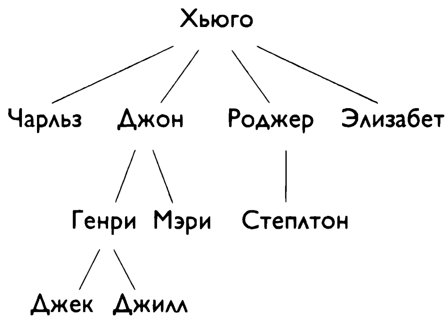
Хьюго → Чарльз → Джон → Генри → Роджер → Степлтон → Элизабет

Такой список наследования можно вычислить, обойдя дерево в определенном порядке, т.е. посетив каждый узел дерева прежде его потомков. Кроме того, при таком обходе потомки старших потомков посещаются прежде младших потомков и их потомков. Алгоритм вычисления списка наследования для узла дерева может быть описан следующим образом:

> Чтобы вычислить список наследования для узла *N*, следует вычислить и присоединить списки наследования всех его (от старших к младшим) потомков, поместив узел *N* в начало полученного результата.

Такое описание подразумевает, что список наследования для узла дерева, не имеющего потомков, состоит только из этого узла. Следовательно, список наследования для дерева получается путем вычисления списка наследования для его корневого узла. На первый взгляд, может показаться не совсем обычным, что этот алгоритм обращается к самому себе в своем описании. Такое описание называется рекурсивным (см. главы 12, “Своевременный стежок вычисляется впрок”, и 13, “Все дело в интерпретации”).

Ниже приведено генеалогическое дерево с рядом дополнительных членов семейства, чтобы продемонстрировать на его примере, каким образом выполняется рассматриваемый здесь алгоритм. При этом допускается, что у Генри было двое детей, Джек и Джилл, а также младшая сестра Мэри.



Чтобы применить рассматриваемый здесь алгоритм к этому дереву, начиная с Хьюго, следует вычислить список наследования для каждого из детей Хьюго. Начиная со старшего сына Чарльза, список наследования состоит только из него самого, поскольку у него нет детей. Более любопытная ситуация — со средним сыном Джоном. Чтобы вычислить список наследования для него, придется вычислить аналогичные списки для Генри и Мэри и присоединить их. В частности, список наследования для Генри состоит из него самого и двух его детей, а для Мэри — только из нее самой, поскольку у нее нет детей. Итак, мы вычислили следующие списки наследования.

Узел дерева	*Список наследования*
Чарльз	Чарльз
Джон	Джон → Генри → Джек → Джилл → Мэри
Генри	Генри → Джек → Джилл
Мэри	Мэри
Джек	Джек
Джилл	Джилл

Список наследования для узла дерева всегда начинается с самого узла, а список наследования для узла дерева без потомков состоит только из самого этого узла. Более того, списки наследования для Генри и Джона демонстрируют, что подобные списки для узлов дерева с потомками получаются путем присоединения списков наследования их потомков. Аналогичным образом вычисляются списки наследования для Роджера и Элизабет. И если присоединить списки наследования четырех потомков Хьюго, добавив впереди его самого, то в конечном счете получится следующий упорядоченный список наследников Хьюго:

Хьюго → Чарльз → Джон → Генри → Джек → Джилл → Мэри →
Роджер → Степлтон → Элизабет

Алгоритм наследования служит примером *обхода дерева*, т.е. систематического посещения всех узлов дерева. А поскольку такой обход производится сверху вниз, т.е. от корня к листьям дерева, когда узлы посещаются прежде их потоков, то он называется *обходом в прямом порядке* (preorder traversal). В качестве аналогии можно рассмотреть поведение белки, прочесывающей дерево в поисках орешков. Чтобы не пропустить ни одного орешка, белке приходится посетить и обследовать каждую ветку дерева. Она может сделать это с помощью разных стратегий. Одна из возможных стратегий состоит в том, чтобы продвигаться по уровням (высоты) дерева, обойдя все ветки одного уровня, прежде чем перейти к любой ветке на следующем уровне. Другая стратегия состоит в том, чтобы обойти каждую ветку до конца, на какой бы высоте она ни находилась, прежде чем перейти к другой ветке. Обе стратегии гарантируют, что каждая ветка будет посещена и все орешки будут найдены, а различаются они порядком обхода веток. Если для белки это различие не имеет особого значения, поскольку она так или иначе соберет все орешки, то порядок обхода узлов генеалогического дерева важен с точки зрения наследования, поскольку все наследство получит первый же найденный при обходе член семейства. Обход генеалогического дерева в прямом порядке относится ко второй стратегии, предполагающей проход по одной ветви до конца, прежде чем переходить к другой ветви.

Основное назначение типов данных и реализующих их структур данных состоит в том, чтобы собрать данные на одной стадии вычисления для того, чтобы воспользоваться ими на следующих стадиях. А поскольку поиск отдельного элемента в коллекции является настолько важной и часто используемой операцией, специалисты по информатике приложили немало труда, чтобы исследовать структуры данных, эффективно поддерживающие эту операцию. Более подробно этот вопрос обсуждается в главе 5, “Поиск идеальной структуры данных”.

Дальнейшее исследование

Примеры из повести *Собака Баскервилей* наглядно продемонстрировали, каким образом знаки действуют в качестве представлений, а типы и структуры данных организуют коллекции данных. Многие истории о Шерлоке Холмсе изобилуют знаками и их представлением, включая анализ отпечатков пальцев, следы ног и почерк. Даже в повести *Собака Баскервилей* имеется ряд дополнительных знаков, которые здесь не обсуждались. Например, на тросточке доктора Мортимера имеются царапины, указывающие на то, что она употреблялась не только по своему основному назначению, поскольку эти царапины — ни что иное, как следы от зубов, свидетельствовавшие о том, что у доктора Мортимера была собака. А исследовав шрифт анонимного послания, отправленного сэру Генри, Шерлок Холмс пришел к выводу, что его буквы вырезаны из газеты *Times*. По краям вырезок он заключил, что для этой цели использовались маленькие ножницы. Выводы сыщика могут различаться, если знаки могут иметь разную интерпретацию, а некоторые из них вообще игнорируются. Это наглядно показано в книге *Sherlock Holmes Was Wrong* (Шерлок Холмс был не прав) Пьера Байара (Pierre Bayard), где утверждается, что Шерлок Холмс неверно установил личность убийцы в повести *Собака Баскервилей*. Разумеется, во многих других детективных рассказах, повестях, фильмах и популярных телесериалах представлены примеры знаков и представлений, а также демонстрируется их использование при вычислении выводов о нераскрытых преступлениях.

В романе *Имя розы* Умберто Эко (Umberto Eco) повествуется о тайне убийства в монастыре, совершенного в XIV веке. В эту детективную историю введена соответствующая семиотика. Главный герой романа, Уильям Баскервиль, фамилия которого указывает на одноименную повесть о Шерлоке Холмсе и которому поручено расследовать убийство, без предубеждений интерпретирует знаки, приглашая тем самым читателя принять активное участие в анализе событий, происходящих в данной истории. А позднее Дэн Браун (Dan Brown) употребил немало знаков в своих популярных романах *Код да Винчи* и *Утраченный символ*.

В романе *Путешествия Гулливера* Джонатан Свифт (Jonathan Swift) повествует о проекте Великой академии Лагадо, намеревавшейся исключить из употребления слова, а значит, и знаки, лишь на том основании, что они обозначали

предметы. Вместо применения слов людям предлагалось носить с собой необходимые предметы всякий раз, когда им требовалось поговорить о них. Эта сатира Свифта наглядно показывает особый вклад знаков в смысловое значение языков. Аналогично в книгах *Алиса в Стране чудес* и *Алиса в Зазеркалье* Льюиса Кэрролла (Lewis Carroll) содержатся примеры искусной игры слов, которая порой бросает вызов традиционной роли слов выступать в качестве знаков.

Используя стек, можно обратить элементы в списке, сначала вводя их в стек, а затем извлекая их оттуда в обратном порядке по принципу "первым пришел, последним вышел". Характерными примерами использования стека, безусловно, служат поиск Гензелем и Гретель обратного пути домой, а также выхода из лабиринта героем Тезеем в древнегреческом мифе о Тезее и Минотавре и послушником Адсо в романе *Имя розы*. Стеком можно также воспользоваться, чтобы понять сюжет фильма *Memento* (Помни), который раскрывается в обратном хронологическом порядке. Размещая сцены фильма в мысленном стеке, а затем извлекая их оттуда в обратном порядке по отношению к тому, как они были показаны, можно восстановить события повествуемой истории в правильном порядке.

Возможно, использование дерева для представления родственных отношений, как мы сделали для повести *Собака Баскервилей*, кажется излишним, но семейства порой оказываются настолько многочисленными, что проследить в них родственные отношения без соответствующего представления очень трудно. В серии фантастических романов *Песнь льда и огня* Джорджа Р.Р. Мартина (George R.R. Martin) и известной его экранизации в телесериале *Игра престолов* присутствуют три крупных генеалогических дерева: домов Старков, Ланнистеров и Таргарьенов. Другими примерами крупных генеалогических деревьев служат сонмы богов в древнегреческой и скандинавской мифологии.

Решения задач и их ограничения

Индиана Джонс

Потери и находки

Где же та заметка, которую вы сделали несколько месяцев назад? Вы точно знаете, что написали ее о проекте, который снова всплыл на поверхность, на каком-то листе бумаги. Чтобы найти эту заметку, вы обыскали все места, где только могли лежать листы бумаги, по крайней мере вы так думали, но так и не смогли ее найти. При этом обнаружилось, что в некоторых местах вы искали потерянную заметку повторно, возможно, потому, что в первый раз вы искали, на ваш взгляд, недостаточно тщательно. Попутно вы обнаружили заметки, которые отчаянно, но безуспешно искали некоторое время назад. Что за наказание!

Знакомая ситуация, не так ли? Конечно, такое случалось и со мной, причем не один раз. Без сомнения, найти что-нибудь нелегко, а безуспешные поиски могут принести немалое разочарование. И даже в нынешние времена всесильного (по крайней мере, на первый взгляд) поискового механизма Google найти нужную информацию в Интернете непросто, если заранее неизвестны подходящие ключевые слова или критерии поиска, а общие ключевые слова дают слишком обширные его результаты.

К счастью, мы не обречены стать беспомощными жертвами бесконечного потока данных. Трудности поиска можно преодолеть, правильно организовав место для поиска. Ваша мать в конечном счете была права, когда регулярно велела вам убраться в вашей комнате [1]. Эффективная организация места поиска основывается на паре следующих принципов: 1) разделение места поиска на непересекающиеся области и расположение искомых элементов на этих участках и 2) расстановка искомых элементов на их участках в определенном порядке.

Например, принцип разделения может привести к размещению всех книг на книжных полках, всех документов — в картотечном ящике, а всех заметок — в скоросшивателе. Если вам потребуется затем найти заметку, вы будете знать, что искать ее следует только в скоросшивателе, а не в картотечном ящике или на книжной полке. Разделение важно потому, что оно позволяет эффективно ограничить место поиска более разумными, контролируемыми размерами. Данный принцип можно применять на нескольких уровнях, чтобы еще больше сузить поиск. Например, книги можно дополнительно сгруппировать по тематике, а документы — по году их составления.

Безусловно, разделение действует при двух важных условиях. Во-первых, категории, употребляемые для подразделения места поиска, должны быть различимы

для поиска нужного элемента. Допустим, что требуется найти повесть *Превращение* Франца Кафки. Находится ли эта книга в рубрике *Художественная литература* или *Философия*? А может быть, она присутствует среди книг по энтомологии или среди комиксов о превращении таких супергероев, как, например, Халк или один из людей Икс? И во-вторых, такая стратегия может оказаться пригодной лишь в том случае, если разделение пространств поиска поддерживается правильно и постоянно. Так, если извлечь заметку из скоросшивателя и оставить ее на столе или положить на книжную полку вместо того, чтобы вернуть обратно в скоросшиватель тотчас после применения по назначению, в следующий раз найти эту заметку в скоросшивателе вряд ли удастся.

Поддержание надлежащего порядка, т.е. второго принципа для обеспечения поиска, находит применение в самых разных ситуациях: от употребления печатных словарей до хранения колоды игральных карт. Упорядочение искомых элементов упрощает и ускоряет их обнаружение. Метод, используемый для легкого поиска элементов в отсортированной коллекции, называется *бинарным поиском.* В этом случае сначала выбирается элемент где-то посредине коллекции, а затем поиск продолжается влево или вправо от выбранного элемента в зависимости от того, является ли искомый элемент меньше или больше найденного соответственно. Кроме того, принципы разделения и упорядочения можно объединить, благодаря чему нередко обеспечиваются некоторые удобство и гибкость. Так, книги можно хранить на полке, упорядочив их по авторам, документы — по датам, а заметки — по их основным темам.

При более тщательном анализе обоих принципов обнаруживается, что они взаимосвязаны. В частности, принцип хранения предметов в определенном порядке подразумевает строгое и рекурсивное соблюдение принципа разделения пространства поиска. Каждый элемент разделяет пространство коллекции на два участка, содержащего все более мелкие и крупные элементы, причем каждый из этих участков организован тем же самым образом. Но поддержание элементов коллекции в организованном порядке требует известных усилий. С точки зрения вычисления возникает следующий вопрос: оправдывает ли ускорение поиска усилия, затрачиваемые на поддержание порядка? Ответ на него зависит от того, насколько крупной оказывается коллекция и как часто приходится совершать в ней поиск элементов.

Потребность в поиске возникает не только в учреждении. Те же самые усилия, разочарования и соответствующие решения характерны и для других условий (например, кухонь, гаражей или любительских мастерских). Поиск требуется и в тех ситуациях, которые обычно не воспринимаются как таковые. Ряд очевидных и не вполне очевидных примеров поиска дает обсуждаемая в главе 5, “Поиск идеальной структуры данных”, история поиска Священного Грааля из фильма *Индиана Джонс и последний крестовый поход.*

5 Поиск идеальной структуры данных

Типы данных, обсуждавшиеся в главе 4, “Записная книжка сыщика”, охватывают отдельные схемы доступа к коллекциям данных. А поскольку коллекции могут сильно разрастаться, с практической точки зрения очень важно управлять ими эффективно. Как упоминалось ранее, в разных структурах данных поддерживается эффективность тех или иных операций, и они предъявляют разные требования к свободному пространству для хранения данных. Если построение и преобразование коллекций являются важными задачами, то поиск элементов в коллекции, вероятно, относится к наиболее часто востребованным операциям.

Нам постоянно приходится что-то искать. И зачастую это происходит неосознанно, хотя иногда мы болезненно воспринимаем данное обстоятельство, например в ходе такой обыденной деятельности, как мучительный поиск ключей от автомашины. Чем больше имеется мест для поиска и элементов для обхода, тем труднее найти именно то, что требуется найти. С годами у нас накапливается немало артефактов, ведь, в конце концов, мы — наследники охотников и собирателей. Кроме таких элементов реальных коллекций, как марки, монеты или коллекционные карточки с изображениями известных спортсменов, мы с годами накапливаем целые собрания книг, картин или предметов одежды. Иногда это происходит как побочный эффект какого-нибудь увлечения или страсти. Мне, например, известен ряд страстных любителей домашних ремонтов, накопивших впечатляющие коллекции инструментов.

Если книжная полка организована по алфавиту или по теме, а коллекция картин является электронной и имеет врем*е*нн*ы*е отметки, то поиск конкретной

книги или изображения может упроститься. Если же коллекция насчитывает немало элементов, но не организована как следует, то поиск в ней может быть сильно затруднен.

Положение намного ухудшается, когда дело доходит до хранения данных в электронном виде. Ведь в данном случае имеется, по существу, неограниченный доступ к месту для хранения данных, и поэтому объем хранимых данных быстро растет. Например, по статистике веб-сайта YouTube каждую минуту на него выгружается около 300 часов видеозаписей [1].

Поиск является преобладающей задачей в реальной жизни и очень важным вопросом в информатике. Алгоритмы и структуры данных способны значительно ускорить процесс поиска. И то, что подходит для данных, иногда может помочь и в хранении и извлечении физических артефактов. Этот очень простой метод освобождает от повторного поиска ключей от автомашины, если, конечно, скрупулезно ему следовать.

Ключ к быстрому поиску

В истории из фильма *Индиана Джонс и последний крестовый поход* главный герой, Индиана Джонс, отправляется на поиски двух основных объектов. Сначала он пытается найти своего отца, сэра Генри Джонса, а затем вместе с ним — Священный Грааль. Будучи археологом, Индиана Джонс знает объекты своих поисков. На одной из своих лекций он фактически поясняет студентам следующее:

> *Археология — это поиск фактов.*

Каким же образом действует поиск археологического артефакта (или человека)? Если местоположение искомого объекта известно, то никакого поиска, безусловно, не требуется. В противном случае процесс поиска зависит от следующих двух факторов: *места поиска* и потенциальных *путеводных нитей*, или просто *ключей*, сужающих область поиска.

В истории из фильма *Индиана Джонс и последний крестовый поход* главный герой получает по почте из Венеции записную книжку своего отца со сведениями о Священном Граале, что вынуждает его начать поиски именно оттуда. И здесь происхождение записной книжки служит Индиане Джонсу ключом к разгадке, значительно сужающим первоначальную область поиска от целого земного шара до одного города. В данном примере область поиска имеет двумерную геометрическую форму буквально, но в общем этот термин обозначает нечто более абстрактное. Например, любую структуру данных, представляющую коллекцию элементов, можно рассматривать в качестве области поиска, в которой можно искать конкретные элементы. Например, такой областью поиска служит составленный Шерлоком Холмсом список подозреваемых, в котором можно найти конкретное имя.

Насколько списки пригодны для поиска? Как обсуждалось в главе 4, "Записная книжка сыщика", в худшем случае приходится предварительно просматривать все элементы в списке, прежде чем будет найден нужный элемент, или же выяснять, что таковой элемент в списке отсутствует. Это означает, что список не позволяет эффективно воспользоваться ключом для сужения области поиска.

Снаружи

Внутри

Чтобы стало понятнее, почему список не годится в качестве структуры данных для поиска, целесообразно рассмотреть подробнее принцип действия ключей. В двумерной области поиска ключ обеспечивает границу, отделяющую то, что находится "снаружи" и не содержит искомый элемент, от того, что находится "внутри" и может содержать искомый элемент [2]. Аналогично ключи обозначают в структуре данных границу, которая может разделять разные части этой структуры данных, а следовательно, ограничивать поиск пределами одной из ее частей. Кроме того, ключ представляет собой фрагмент информации, связанной с искомым объектом. В частности, ключ должен быть в состоянии выявить границу между элементами, относящимися к текущему поиску, и не имеющими к нему отношения.

В списке каждый элемент служит границей, отделяющей предшествующие ей элементы от следующих за ней. Но поскольку элементы в средине списка непосредственно недоступны и список вместо этого приходится всегда обходить из одного конца в другой, то его элементы, по существу, не способны отделить внешнее от внутреннего. Рассмотрим первый элемент в списке. Он не исключает никакой элемент из поиска, кроме него самого. Ведь если это не искомый элемент, то поиск придется продолжить среди всех остальных элементов списка. Но тогда при просмотре второго элемента в списке возникнет та же самая ситуация. Если же и второй элемент не окажется искомым, то придется снова продолжить поиск среди всех остальных элементов списка. Безусловно, второй элемент определяется как внешний по отношению к первому элементу списка, который не нужно проверять. Но тот факт, что он уже был проанализирован, еще не означает, что можно что-нибудь сэкономить на поиске. Это же справедливо для каждого элемента списка: чтобы достичь элемента в списке, необходимо проверить то, что определяется им как внешнее.

Приехав в Венецию, Индиана Джонс продолжает свои поиски в библиотеке. Поиск книги в библиотеке служит ярким примером, демонстрирующим применение понятий границы и ключа на практике. Когда книги располагаются на библиотечных полках упорядоченными по фамилиям авторов, каждая полка нередко обозначается фамилиями авторов первой и последней книг на этой полке. Фамилии этих двух авторов задают пределы, в которых можно искать книги указанной части авторов на полке. По существу, фамилии этих двух авторов определяют границу, отделяющую книги авторов в заданных пределах от всех остальных

книг. Допустим, что в библиотеке требуется найти повесть *Собака Баскервилей* Артура Конана Дойла. В качестве ключа можно воспользоваться фамилией данного автора, чтобы сначала найти полку с книгами в соответствующих пределах фамилий авторов. Как только такая полка будет найдена, поиск можно будет продолжить на самой полке. Такой поиск осуществляется в две стадии: на первой ищется полка, а на второй — книга на самой полке. Такая стратегия поиска оказывается вполне пригодной, поскольку разделяет область поиска на целый ряд небольших не перекрывающихся областей (полок), разделяемых фамилиями авторов как границами.

Поиск книг на библиотечных полках можно осуществлять по-разному. Один из способов состоит в том, чтобы проверять полки по очереди до тех пор, пока не будет найдена полка с фамилией "Дойл". В этом случае полки, по существу, выступают в качестве списка, а следовательно, им присущи такие же ограничения, как и спискам. Так, поиск по фамилии "Дойл" может отнять не очень много времени, но если искать книгу по фамилии "Ясперс", то ее поиск может отнять намного больше времени. Лишь немногие начнут поиск книги Ясперса, начиная не с буквы А, а с буквы Я, чтобы быстрее найти нужную полку. Такой способ основывается на предположении, что все библиотечные полки упорядочены по фамилиям авторов и что на каждую букву алфавита приходится одна полка, и поэтому вполне естественно начать поиск книг Ясперса с последней полки в библиотеке, а книги Дойла — с пятой полки.

Иными словами, библиотечные полки можно рассматривать в качестве элементов массива, проиндексированных буквами. Маловероятно, конечно, что в библиотеке есть столько же полок, сколько букв в алфавите, но рассматриваемый здесь способ можно расширить до любого количества полок, просто начав поиск книг Ясперса с последней одной тридцать второй части полок. А недостаток такого способа поиска заключается в том, что количество авторов, фамилии которых начинаются на одну и ту же букву, сильно разнится (например, авторов на букву "Д" намного больше, чем на букву "Я"). Иными словами, книги распределены неравномерно по всем полкам в библиотеке, и поэтому такая стратегия поиска неточна. Следовательно, посетителю библиотеки придется совершать поиск то в одном, то в другом направлении, чтобы найти целевую полку. Зная о неравномерном распределении фамилий авторов, можно было бы значительно повысить точность данной стратегии поиска, но даже столь простая стратегия оказывается вполне пригодной на практике, по существу, исключая как нечто "внешнее" большое количество полок, которые вообще не ст*о*ит просматривать.

Поиск книги на одной полке можно продолжать разными способами. Один из них состоит в том, чтобы просмотреть книги по очереди или прибегнуть к стратегии, аналогичной той, которая применяется для обнаружения полок, оценив местоположение книги по положению ключа в пределах фамилий авторов на

данной конкретной полке. В общем, двухэтапный способ поиска вполне пригоден и оказывается намного более быстродействующим, чем прямолинейный способ просмотра всех книг по очереди. Я провел эксперимент в поисках повести *Собака Баскервилей* в публичной библиотеке города Корваллис, шт. Орегон. И мне удалось найти нужную полку за пять шагов, а книгу на полке — еще за семь шагов. Это значительное улучшение по сравнению с прямолинейным способом поиска перебором, особенно если учесть, что на тот момент в данной библиотеке находилось 44 679 книг по художественной литературе для взрослых на 36 полках.

Решение Индианы Джонса отправиться в Венецию, чтобы найти своего отца, основано на аналогичной стратегии. В этом случае внешний мир можно рассматривать как массив элементов, соответствующих географическим регионам, проиндексированным по названиям городов. Обратный адрес на присланной по почте посылке с записной книжкой сэра Генри Джонса служит ключом для выбора элемента массива по индексу "Венеция", чтобы продолжить поиск. Поиск в Венеции приводит Индиану Джонса в библиотеку, где он ищет не книгу, а надгробие сэра Ричарда, одного из рыцарей, участвовавших в пятом крестовом походе. Он нашел надгробие после того, как разбил плитку на полу, помеченную знаком X, что не было лишено иронии, если принять во внимание следующее его прежнее заявление своим студентам:

Знак "X" едва ли вообще помечает место.

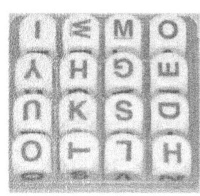
Ключом к быстрому поиску служит наличие структуры, позволяющей быстро сузить поиск. Чем меньше "внутреннее", обозначаемое ключом, тем лучше, поскольку это позволяет быстрее свести весь поиск к искомому результату. Так, если речь идет о поиске книг, то две стадии такого поиска разделяются на одном сужающем поиск главном шаге.

Выживание в боггле

Нередко поиск скрывается под какой-нибудь личиной в самых неожиданных обстоятельствах. К концу истории из фильма *Индиана Джонс и последний крестовый поход* главный герой, Индиана Джонс, прибывает в Храм Солнца, где ему предстоит преодолеть три препятствия, прежде чем войти, наконец, в помещение, где хранится Священный Грааль. Чтобы преодолеть второе препятствие, ему необходимо пересечь выложенный плиткой пол над пропастью. Трудность состоит в том, что лишь некоторые плитки пола надежны, а остальные осыпаются и ведут к ужасной смерти, если наступить на них. Пол состоит приблизительно из 50 плиток, неравномерно уложенных сеткой и обозначенных отдельными буквами алфавита. Доля археолога непроста: порой ему приходится ломать плитку пола, чтобы преуспеть, а иногда — избегать этого любой ценой.

Найти надежные плитки, на которые можно безопасно наступить, не так-то просто, поскольку в отсутствие каких-нибудь других ограничений вероятность попасть на такую плитку составляет один шанс из тысячи триллионов, т.е. число, состоящее из единицы с пятнадцатью нулями. Ключ к поиску жизнеспособной последовательности плиток, надежно ведущей к другой стороне пола, состоит в том, что буквы на плитках должны составлять имя *Iehova* (Иегова, т.е. “Сущий” — имя Бога в Ветхом Завете). И хотя эта информация, по существу, решает загадку, необходимо еще потрудиться, чтобы выявить правильную последовательность плиток. Как ни странно, для этого потребуется *поиск*, систематически сужающий область возможностей выбора.

Эта задача подобна игре в боггл, цель которой — найти строки связанных букв на сетке, образующей слова [3]. Задача Индианы Джонса кажется довольно простой, поскольку ему уже известно слово. Но в боггле последовательные символы должны находиться на соседних плитках; такого ограничения для задачи преодоления плиточного пола нет, а следовательно, имеется куда больше возможностей, которые необходимо рассмотреть Индиане Джонсу, что существенно затрудняет решение данной задачи.



Чтобы продемонстрировать решение задачи поиска, допустим ради простоты, что пол состоит из шести рядов, а каждый из них — из восьми плиток с разными буквами, что в итоге составляет 48 плиток. Если правильный путь состоит из одной плитки в каждом ряду, то всего имеется 8×8×8×8×8×8 = 262 144 возможных путей через все возможные комбинации плиток в шести рядах. И лишь один из этих путей оказывается правильным.

Как же тогда Индиане Джонсу найти правильный путь? Руководствуясь ключевым словом, он находит плитку с буквой *I* в первом ряду и становится на нее. Это само по себе обозначает процесс поиска, состоящий из нескольких стадий. Если буквы на плитках расположены не в алфавитном порядке, то Индиане Джонсу придется просматривать их по очереди до тех пор, пока не будет найдена плитка с буквой *I*. Став на нее, он продолжает поиск плитки с буквой *e* во втором ряду и т.д.

Если поочередный поиск напоминает вам поиск элемента в списке, вы совершенно правы. Именно это и происходит в данном случае. Отличие состоит в том, что поиск в списке повторяется в каждом ряду, т.е. каждого символа в ключевом слове. И в этом состоит истинная сила данного метода поиска. Рассмотрим, что происходит в области поиска, где насчитывается всего 262 144 возможных путей через сетку плиток. Каждая плитка в первом ряду обозначает разную отправную точку пути, который может быть продолжен 8×8×8×8×8 = 32 768 разными способами выбора разных сочетаний букв из пяти оставшихся рядок плиток. Если Индиана Джонс найдет первую плитку, обозначенную, скажем, буквой *K*, он,

конечно, не станет на нее, поскольку она не соответствует требуемой букве *I*. Этим единственным решением одним махом отметается в общем 32 768 остальных путей из области поиска, а именно — тех путей, которые могут быть сформированы, начиная с плитки, обозначенной буквой *K*. И то же самое сокращение возможных путей происходит с каждой последующей отвергаемой плиткой в первом ряду.

Как только Индиана Джонс найдет правильную плитку, область поиска сократится еще больше. Ведь как только он станет на плитку, процесс принятия решения в первом ряду завершится, и область поиска сразу же сократится до 32 768 путей, которые могут быть сформированы в оставшихся пяти рядах. Приняв в общем не больше семи решений, которые требуются, если плитка с буквой *I* оказывается последней в первом ряду, Индиана Джонс сокращает область поиска в восемь раз. И так он продолжает действовать во втором ряду плиток, чтобы найти букву *e*. И снова область поиска сокращается в восемь раз, т.е. до 4096 путей, с помощью не более семи решений. Как только Индиана Джонс достигнет последнего ряда, у него останется восемь возможных путей, и снова ему потребуется принять не более семи решений, чтобы завершить свой путь. В худшем случае для поиска ему потребуется сделать 6×7 = 42 "шага" (и лишь шесть буквальных шагов), и это замечательно эффективный способ поиска одного из 262 144 возможных путей.

Трудная задача, которую удалось решить Индиане Джонсу, имеет некоторое, хотя и не совсем очевидное, сходство с задачей Гензеля и Гретель. В обоих случаях главному герою приходится искать безопасный путь, который состоит из последовательного ряда отмеченных мест: в истории Гензеля и Гретель — камешками, а в истории Индианы Джонса — буквами на плитках пола, но поиск следующего местоположения на обоих путях разнится. В частности, Гензелю и Гретель достаточно найти любой камешек, хотя они должны следить за тем, чтобы не обходить повторно те же самые камешки, тогда как Индиане Джонсу требуется найти плитку с конкретной буквой. Оба эти примера еще раз подчеркивают роль представления в вычислении. Трудная задача, которую приходится решать Индиане Джонсу, демонстрирует, в частности, что последовательность означающих (букв) для отдельных плиток сама по себе служит еще одним означающим (слово *Iehova*) для пути. А тот факт, что слово *Iehova* обозначает искомый путь, наглядно показывает, насколько значимым и важным в реальном мире становится вычисление, которое просто состоит в поиске слова.

Способ, которым Индиана Джонс отыскивает ключевое слово на сетке плиток, точно соответствует тому, как можно эффективно найти слово в словаре, сузив поиск сначала до ряда страниц, содержащих слово, начинающееся на ту же самую букву, что и ключевое слово, а затем до еще более узкого ряда страниц, соответствующих первой и второй буквам искомого слова, и так далее до тех пор, пока слово не будет найдено.

Чтобы лучше понять структуру данных, скрывающуюся за сеткой плиток и делающей поиск Индианы Джонса столь эффективным, следует рассмотреть еще один способ представления слов и словарей, в которых для организации области поиска применяются деревья.

Словарный подсчет

В главе 4, "Записная книжка сыщика", была описана структура данных дерева. В частности, генеалогическое дерево послужило примером для того, чтобы продемонстрировать порядок вычисления списка наследников и приоритета их наследования. В ходе этого вычисления был выполнен обход всего дерева, при котором пришлось посетить каждый его узел. Деревья отлично подходят и для поддержки поиска элементов в коллекции. В этом случае узлы дерева используются с целью направить поиск по единственно верному пути и найти требуемый элемент.

Рассмотрим еще раз трудную задачу, которую пришлось решать Индиане Джонсу. В фильме *Индиана Джонс и последний крестовый поход* его первый шаг приводит к драматичной сцене, в которой он едва не погиб, став на ненадежную плитку. Используя написание *Jehova* ключевого слова, он становится на плитку с буквой *J*, которая ломается под его ступнями. Это ясно показывает, что рассматриваемая здесь трудная задача фактически оказывается еще более трудной, чем кажется на первый взгляд, поскольку у Индианы Джонса нет полной уверенности в правильности написания ключевого слова. Помимо букв *I* и *J*, ключевое слово может начинаться и с буквы *Y*. Кроме того, имеются другие возможные имена Бога, которые в принципе подходят для решения данной задачи, например *Yahweh* (Яхве) и *God* (Бог). Допустим, что все эти слова составляют возможные варианты выбора и что Индиана Джонс и его отец уверены, что одно из этих слов действительно указывает безопасный путь через плиточный пол.

Какая же стратегия подходит для выбора плитки, на которую следует стать? Если все упомянутые выше ключевые слова равнозначны, то Индиана Джонс мог бы улучшить свое положение, выбрав букву, присутствующую в нескольких словах. Допустим, что он выбрал букву *J*, как это и было на самом деле. Буква *J* присутствует только в одном из имен Бога, и поэтому есть лишь один из пяти шансов (т.е. вероятность 20%), что плитка с этой буквой устоит. С другой стороны, вероятность безопасно стать на плитку с буквой *v* составляет 60%, поскольку эта буква присутствует в трех именах Бога. И если какое-то из этих слов правильное, то плитка с буквой *v* окажется надежной, а поскольку все ключевые слова считаются в равной степени правильными, то шансы выжить возрастают для плитки с буквой *v* до трех из пяти [4].

Таким образом, подходящая стратегия состоит в том, чтобы вычислить сначала частотность букв в пяти словах, а затем попробовать стать на плитки с наибольшей встречаемостью. Такое соответствие букв их частотности называется *гистограммой.* Чтобы вычислить гистограмму букв, Индиане Джонсу придется организовать ведение подсчета частотности каждой буквы. Просмотрев все слова, он может инкрементировать подсчет каждого встретившегося слова. Ведение подсчета разных букв является задачей, решаемой с помощью типа данных словаря (см. главу 4, "Записная книжка сыщика"). В данном случае ключи служат отдельными буквами, а сведения, сохраняемые с каждым ключом, — частотностью буквы ключа.

С одной стороны, чтобы реализовать такой словарь, можно было бы воспользоваться массивом с буквами в качестве индексов, но это означало бы напрасно израсходовать более половины свободного места в массиве, поскольку подсчитать требуется всего одиннадцать разных букв. С другой стороны, для этой цели можно было бы воспользоваться списком, но и это не очень эффективно. Чтобы убедиться в этом, допустим, что слова просматриваются в алфавитном порядке. Следовательно, просмотр начинается со слова *God*, каждая буква которого вводится в список вместе с начальным подсчетом, равным 1. В итоге получается следующий список:

G:1 → o:1 → d:1

Следует, однако, иметь в виду, что если для ввода буквы *G* в список потребовался один шаг, то для ввода буквы *o* — два шага, поскольку ее пришлось ввести после буквы *G*, а для ввода буквы *d* — уже три шага, потому что она должна быть введена в список после обеих букв, *G* и *o*. Но нельзя ли вместо этого вводить новые буквы в начале списка? К сожалению, нельзя, поскольку необходимо убедиться в отсутствии буквы в списке, прежде чем ее вводить. Следовательно, придется просмотреть все присутствующие в списке буквы, прежде чем вводить в него новую букву. Если буква уже присутствует в списке, то вместо ее ввода следует просто увеличить ее счетчик. Таким образом, на первое слово уже затрачено 1+2+3 = 6 шагов.

Если перейти к следующему слову, *Iehova*, то для ввода букв *I*, *e* и *h* в список потребуется 4, 5 и 6 шагов соответственно, а в итоге — 21 шаг. Далее следует буква *o*, которая уже присутствует в списке. Чтобы найти эту букву и обновить ее счетчик до 2, потребуются лишь два шага. Буквы *v* и *a* являются новыми для данного списка, и поэтому для их добавления в конец списка потребуется 7+8 = 15 шагов. Итак, для заполнения данного списка потребовалось всего 38 шагов, а выглядит он следующим образом:

G:1 → o:2 → d:1 → I:1 → e:1 → h:1 → v:1 → a:1

Для обновления в словаре счетчиков букв из слова *Jehova* потребуется еще 9 шагов на ввод буквы *J* и далее 5, 6, 2, 7 и 8 шагов для увеличения счетчиков остальных букв, уже присутствующих в списке, что в итоге составит 75 шагов. Для обработки двух последних слов, *Yahweh* и *Yehova*, требуется 40 и 38 шагов соответственно, а в итоге получается следующий список, на заполнение которого потребовалось 153 шага:

G:1 → o:4 → d:1 → I:1 → e:4 → h:4 → v:3 → a:4 → J:1 → Y:2 → w:1

Следует, однако, иметь в виду, что вводить вторую букву *h* из слова *Yahweh* не нужно, поскольку двойной ее учет для одного слова приведет (причем неверно) к увеличению вероятности встретить эту букву (в нашем случае — с 80% до 100%) [5]. Окончательный просмотр данного списка показывает, что буквы *o*, *e*, *h* и *a* оказываются самыми надежными, а вероятность встретить их в правильном слове составляет 80%.

Скудость — не всегда добродетель

Главный недостаток реализации словаря в виде структуры данных списка заключается в больших затратах на повторный доступ к элементам, находящимся ближе к концу списка. В структуре данных *бинарного дерева поиска* предпринимается попытка преодолеть подобный недостаток путем более равномерного разделения области поиска для поддержки ускоренного поиска. *Бинарным* называется такое дерево, у каждого узла которого имеются хотя бы два потомка. Как упоминалось ранее, узлы без потомков называются *листьями*, узлы с потомками — *внутренними узлами*, а узел, не имеющий родителя, — *корневым.*

Рассмотрим ряд примеров бинарного дерева. Так, на рис. 5.1, *слева*, показано дерево с единственным узлом, т.е. самое простое дерево, какое только можно себе

представить, — это корень, одновременно являющийся листом этого дерева (помечен буквой *G*). На самом деле это дерево не очень похоже на подлинное дерево, подобно тому как список с единственным элементом — на настоящий список. На рис. 5.1 посредине показано дерево, состоящее из двух узлов: корневого и дочернего узла, помеченного буквой *o*, а на рис. 5.1, *справа*, — то же дерево, но содержащее дополнительный дочерний узел, помеченный буквой *d* и имеющий два потомка — листья *a* и *e*. Данный пример наглядно показывает, что если отсечь связь узла с его родителем, он станет корневым узлом отдельного дерева, называемого *поддеревом* своего родительского дерева. В данном случае дерево с корневым узлом *d* и двумя его потомками, *a* и *e*, является поддеревом узла *G*, как, впрочем, и дерево с единственным (корневым) узлом *o*. Это означает, что дерево, по существу, является

рекурсивной структурой данных и может быть определено следующим образом: дерево состоит из единственного узла или же из узла с одним или двумя поддеревьями, корневые узлы которых являются потомками данного узла. На подобный рекурсивный характер структуры деревьев указывалось в главе 4, “Записная книжка сыщика”, в которой был представлен рекурсивный алгоритм для обхода генеалогического дерева.

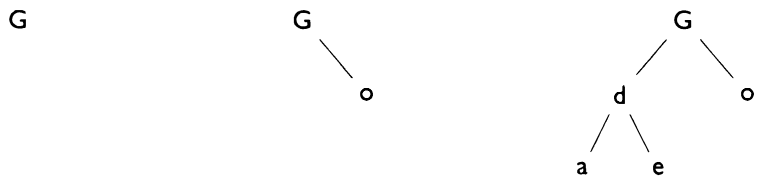


Рис. 5.1. Три примера двоичных деревьев. Слева: дерево с единственным узлом. Посредине: дерево с корневым узлом и правым поддеревом, которое представляет собой единственный узел. Справа: дерево с корневым узлом и двумя поддеревьями. Все три дерева обладают свойством бинарного поиска, которое означает, что узлы в левых поддеревьях содержат значения, меньшие значений в корневом узле, а узлы в правых поддеревьях — значения, большие, чем в корневом узле

Принцип организации бинарного дерева поиска состоит в том, чтобы использовать внутренние узлы в качестве границ, разделяющих все дочерние узлы на два класса: узлы со значением, меньшим, чем у граничного узла, содержатся в левом поддереве, а узлы с б*о*льшим значением — в правом поддереве.

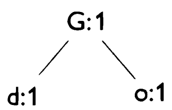
Таким расположением узлов дерева можно воспользоваться для поиска. Так, если требуется найти конкретное значение в дереве, оно сравнивается со значением в корневом узле дерева. Если обнаружено совпадение, то искомое значение найдено, и на этом поиск прекращается. В противном случае, если искомое значение оказывается меньше, чем значение в корневом узле дерева, дальнейший поиск можно ограничить только левым поддеревом. Это избавляет от необходимости искать значение в любых узлах правого поддерева, поскольку заранее известно, что все его узлы содержат большее значение, чем в корневом узле, а следовательно, оно больше искомого значения. Но в любом случае внутренний узел определяет границу, разделяющую то, что находится “внутри” (т.е. в левом поддереве), от того, что находится “снаружи” (т.е. в правом поддереве), подобно тому, как дневник о Святом Граале определяет то, что находится “внутри”, на следующем шаге поиска Индианой Джонсом своего отца.

Все три дерева, показанные на рис. 5.1, являются бинарными деревьями поиска. В качестве значений они хранят буквы, которые можно сравнивать по их положению в алфавите. Чтобы найти значение в таком дереве, придется неоднократно сравнивать искомое значение со значениями в отдельных узлах дерева, опускаясь соответственно по правому или левому поддереву.

Допустим, что требуется выяснить, где находится буква *e* в правом дереве на рис. 5.1. Начнем поиск с корневого узла дерева и сравним буквы *e* и *G*. Поскольку буква *e* предшествует в алфавите букве *G*, она оказывается "меньше" буквы *G*, и поиск продолжается в левом поддереве, где находятся буквы, значения которых меньше значения буквы *G*. Сравнив далее букву *e* с буквой *d* в корневом узле левого поддерева, мы обнаружим, что буква *e* оказывается "больше" буквы *d*, а следовательно, поиск продолжается в правом поддереве, которое содержит единственный узел. Сравнение буквы *e* со значением в этом узле завершает поиск. Если бы мы искали букву *f*, то ее поиск привел бы к тому же самому узлу *e*, но поскольку у узла *e* правое поддерево отсутствует, то поиск завершился бы в узле *e* неудачей — выводом, что буква *f* в дереве отсутствует.

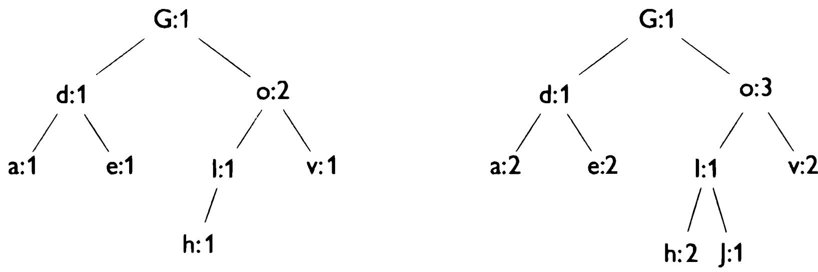
Любой поиск следует по некоторому пути в дереве, и поэтому время поиска элемента (или вывода об его отсутствии в дереве) никогда не превышает время следования по самому длинному пути от корня к листу дерева. Дерево, показанное на рис. 5.1, *справа*, состоит из пяти элементов, а пути к его листьям имеют длину, равную только двум или трем элементам. Это означает, что любой элемент можно найти не больше чем за три шага. Сравните это со списком из пяти элементов, в котором для поиска последнего элемента всегда требуется пять шагов.

Теперь мы готовы вычислить гистограмму букв, используя для представления словаря бинарное дерево поиска. И в этом случае мы также начнем со слова *God*, вводя каждую его букву в дерево вместе с начальным счетчиком этой буквы, равным 1. В итоге мы получим приведенное ниже дерево, отражающее порядок появления букв в дереве.



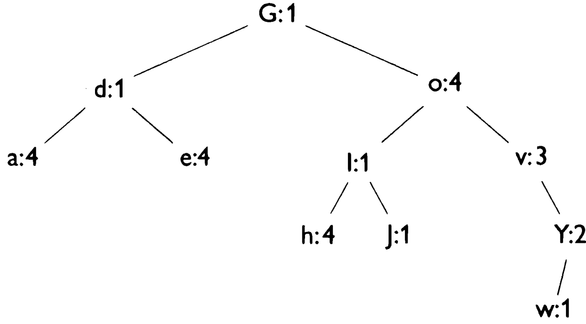
В данном случае для ввода буквы *G* в дерево потребовался один шаг, как и для ввода в список, но для ввода букв *o* и *d* потребовалось только два шага, поскольку их узлы являются потомками узла с буквой *G*. Таким образом, мы сэкономили один шаг на первом слове (1+2+2 = 5 шагов для его ввода в дерево в сравнении с 1+2+3 = 6 шагами для ввода в список).

Для ввода следующего слова, *Iehova*, требуются по три шага на каждую его букву, кроме букв *o* и *h*, для ввода которых требуются два и четыре шага соответственно. Как и при вводе в списки, обработка буквы *o* не приводит к созданию нового элемента, а лишь увеличивает на 1 счетчик существующего элемента. Следовательно, для обработки данного слова потребуется 18 шагов, а вместе с предыдущим словом — 24 шага по сравнению с 38 шагами, потребовавшимися для структуры данных списка. Слово *Jehova* лишь незначительно изменяет структуру дерева, вводя узел для буквы *J*, на что требуется четыре шага. Еще за 3+4+2+3+3 = 15 шагов обновляются счетчики существующих букв, что в итоге дает 39 шагов (сравните с 75 шагами, требовавшимися для списка).



После ввода слова *lehova* | После ввода слова *Jehova*

И наконец, для ввода каждого из слов *Yahwe(h)* и *Yehova* потребовалось по 19 шагов, так что в итоге полностью дерево было построено за 76 шагов. И это составляет лишь половину от 153 шагов, необходимых для построения списка.



Приведенное выше бинарное дерево поиска представляет тот же самый словарь, что и список, но в форме, поддерживающей ускоренный поиск и обновление, по крайней мере в общем случае. Пространственные формы списков и деревьев поясняют некоторые различия их эффективности. Длинная и узкая структура списка нередко вынуждает поиск следовать по долгому пути и просматривать ненужные элементы. Широкая же и плоская форма дерева эффективно направляет поиск и ограничивает количество рассматриваемых элементов и длину пройденного расстояния. Однако для беспристрастного сравнения списков и бинарных деревьев поиска требуется принять во внимание ряд дополнительных особенностей.

Балансирующая эффективность

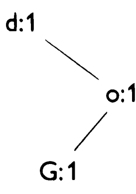
В информатике разные структуры данных обычно не сравниваются путем подсчета точного количества шагов для конкретных списков или деревьев, поскольку это может создать неверное впечатление (особенно это касается небольших структур данных). Но даже в столь простом анализе сравнивались операции, имеющие

неодинаковую сложность и требующие разного времени на выполнение. Например, чтобы выполнить сравнение в списке, достаточно проверить равенство двух элементов, тогда как в бинарном дереве поиска требуется определить, какой из сравниваемых элементов больше, чтобы направить поиск в соответствующее поддерево. Это означает, что выполнить большинство из 153 операций над списком оказывается быстрее и проще, чем некоторые из 76 операций над деревом, и поэтому не следует придавать особого значения непосредственному сравнению двух числовых показателей.

Вместо этого мы рассмотрим увеличение времени выполнения операций по мере укрупнения и усложнения структур данных. Соответствующие примеры были приведены в главе 2, "От слов к делу: когда действительно происходит вычисление". Что касается списков, то, как известно, время выполнения операции ввода новых элементов или поиска существующих элементов линейно, т.е. время ввода или поиска элементов в худшем случае пропорционально длине списка. Это совсем не так плохо, но если выполнять такую операцию многократно, то время ее выполнения будет накапливаться по квадратичному закону, что может оказаться недопустимым (вспомните пример, когда Гензелю приходилось за каждым новым камушком возвращаться домой).

Каково же в сравнении с этим время выполнения операций ввода и поиска элементов в дереве? Наше окончательное дерево поиска состоит из 11 элементов; и поиск, и ввод в него элементов отнимает от трех до пяти шагов. В действительности время поиска или ввода элемента ограничивается высотой дерева, которая зачастую оказывается намного меньше количества его элементов. В сбалансированном дереве, т.е. в дереве, в котором все пути от корня к листу имеют одинаковую длину (±1), высота дерева находится в *логарифмической* зависимости от его размера. Это означает, что высота вырастает лишь на 1, когда количество узлов в дереве удваивается. Например, сбалансированное дерево с 15 узлами имеет высоту 4, сбалансированное дерево с 1000 узлов — высоту 10, а с 1 000 000 узлов — высоту 20. Логарифмическое время выполнения оказывается намного лучше, чем линейное, и по мере увеличения размера словаря структура данных дерева становится все лучше в сравнении со структурой данных списка.

Такой анализ основывается на *сбалансированных* бинарных деревьях поиска. А можно ли гарантировать сбалансированный характер двоичных деревьев при их построении? И если нельзя, то каким будет время выполнения, если деревья окажутся несбалансированными? Окончательное дерево в рассматриваемом здесь примере *не* сбалансировано, поскольку длина пути к листу *e* равна 3, тогда как длина пути к листу *w* равна 5. Порядок, в котором буквы вводятся в дерево, имеет значение и может привести к построению разных деревьев. Так, если вводить буквы в том порядке, в каком они следуют в слове *doG*, то получится приведенное ниже бинарное дерево поиска, которое вообще не сбалансировано.



Такое дерево совершенно не сбалансировано и, по сути, мало чем отличается от списка. И это не какое-то редкое исключение. Две (*God* и *Gdo*) из шести возможных последовательностей, состоящих из трех букв, приводят к сбалансированному дереву, а четыре другие последовательности — к спискам. Правда, существуют методы балансирования бинарных деревьев поиска, теряющих свойство сбалансированности после операций добавления узлов. И хотя они и увеличивают время выполнения операции добавления узла, оно по-прежнему сохраняет свой логарифмический характер.

Наконец, бинарные деревья поиска пригодны для хранения лишь таких разновидностей элементов, которые всегда упорядочены, т.е. о двух любых элементах всегда можно сказать, какой из них больше, а какой меньше. Такое сравнение требуется для того, чтобы решить, в каком направлении двигаться по дереву, чтобы найти или сохранить в нем элемент. Но для некоторых видов данных такие сравнения недопустимы. Предположим, что требуется вести заметки о рисунках стеганых изделий и для каждого рисунка записывать требующиеся ткани и инструменты, особенности и время выполнения стежки. Чтобы сохранить сведения о рисунках стеганых изделий в бинарном дереве поиска, необходимо каким-то образом решить, какой из рисунков больше (или меньше) другого. А поскольку рисунки различаются количеством, формой и цветами фрагментов, совершенно не очевидно, как правильно упорядочить эти рисунки. Это не такая уж невыполнимая задача, поскольку отдельный рисунок можно сначала разложить на составляющие его фрагменты и описать с помощью перечня свойств (например, количества фрагментов, их форм и цветов), а затем сравнить рисунки по перечням их свойств (хотя это потребует определенных усилий и может быть не очень практичным). Таким образом, бинарное дерево поиска может оказаться не совсем пригодным для хранения словаря с рисунками стеганых изделий. Тем не менее для решения подобной задачи можно было бы воспользоваться списком, поскольку для списка достаточно выяснить, являются ли два рисунка одинаковыми, что куда проще, чем их упорядочивать.

Таким образом, бинарные деревья поиска используют стратегию, которая естественно и без особых усилий применяется людьми для разложения задач поиска на более мелкие подзадачи. Фактически бинарные деревья поиска основаны на систематизированном и более совершенном принципе и как следствие для представления словарей могут быть намного более эффективными, чем списки. Однако они требуют больше усилий, чтобы гарантировать сбалансированность, и пригодны не для всех разновидностей данных. Все это отвечает вашему обычному опыту поиска предметов на рабочем столе, на кухне или в гараже. Так, если вы регулярно тратите усилия на поддержание порядка и, попользовавшись, кладете документы или инструменты на место, то вы сможете намного легче и быстрее

найти нужные предметы, чем при поиске их в беспорядочной свалке, которую вы гордо называете рабочим столом.

Префиксное дерево

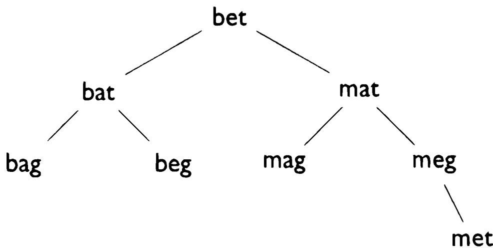
Бинарные деревья поиска служат эффективной альтернативой спискам, когда речь идет о сравнении гистограммы частотности букв в словах. Но это лишь одна из разновидностей вычислений, помогающих Индиане Джонсу решить трудную задачу пересечения плиточного пола. Другая разновидность вычисления состоит в том, чтобы выявить последовательность плиток, образующих конкретное слово из букв и ведущих по безопасному пути через плиточный пол.

Как мы уже определили, сетка состоит из шести рядов, а каждый из них — из восьми букв, образуя 262 144 разных пути по шести плиткам. А поскольку каждый путь соответствует одному слову, связь слов с путями можно было бы представить с помощью словаря. Список оказался бы неэффективным преставлением, поскольку ключевое слово *Iehova* могло бы располагаться ближе к концу списка, а следовательно, для его обнаружения потребовалось бы немало времени. Намного лучше было бы воспользоваться сбалансированным бинарным деревом поиска, поскольку его высота была бы равна 18, гарантируя относительно быстрое отыскание ключевого слова. Но у нас нет такого дерева, а его построение — дело долгое. Как и в дереве поиска, содержащем буквы, нам придется ввести каждый элемент по отдельности, а для этого мы должны будем многократно проходить в дереве пути от его корня к листьям. Даже без подробного анализа данного процесса должно быть ясно, что для построения такого дерева потребуется время, большее линейного [6].

Тем не менее последовательность плиток можно довольно эффективно выявить (не больше, чем за 42 шага) без всякой дополнительной структуры данных. Как же такое возможно? Дело в том, что пол, уложенный плитками с буквами, сам по себе является структурой данных, особенно хорошо поддерживающей вид поиска, совершаемого Индианой Джонсом. Такая структура данных называется *префиксным деревом* (trie) и в какой-то степени подобна бинарному дереву поиска, хотя и заметно отличается от него [7].

Напомним, что, становясь на очередную плитку, Индиана Джонс сокращает область поиска в восемь раз. Нечто подобное происходит при перемещении вниз по бинарному дереву поиска, в котором область поиска сокращается вдвое, поскольку в результате выбора одной из двух ветвей дерева половина узлов исключается из поиска. Аналогично, если не выбрать плитку, область поиска сократится на одну восьмую, а если выбрать плитку — на одну восьмую ее прежнего размера. Ведь выбор одной плитки означает, что не выбраны семь остальных плиток, а следовательно, область поиска сокращается на семь восьмых.

Но здесь происходит нечто иное, что и отличает происходящее от принципа действия бинарного дерева поиска. В бинарном дереве поиска каждый элемент хранится в отдельном узле, тогда как в узле префиксного дерева хранятся только отдельные буквы, а слова представлены как пути, соединяющие разные буквы. Чтобы понять отличие бинарного дерева поиска от префиксного дерева, рассмотрим следующий пример. Допустим, что требуется представить ряд слов *bat* (палка), *bag* (сумка), *beg* (мольба), *bet* (ставка), *mag* (сплетня), *mat* (коврик), *meg* (баба) и *met* (металл). Сбалансированное бинарное дерево поиска, содержащее эти слова, выглядит так, как показано ниже.



Чтобы найти слово *bag* в этом дереве, мы сравниваем его со словом *bet* в корневом узле. Результат этого сравнения подсказывает нам, что надо продолжить поиск в левом поддереве. Для такого сравнения потребуются два шага, в течение которых сравниваются две первые буквы обоих слов. Сравним далее слово *bag* со словом *bat* в корневом узле левого поддерева. Результат этого сравнения снова подскажет нам продолжить поиск в левом поддереве, но при этом нам потребуются уже три шага сравнения, поскольку придется сравнивать все три буквы обоих слов. И, наконец, сравнение слова *bag* со словом в крайнем слева узле дерева приведет к успешному завершению поиска. Для этого последнего сравнения также потребуются три шага, а для всего поиска в целом — восемь шагов сравнения.

b	m
a	e
g	t

Но ту же самую коллекцию слов можно представить как пол, уложенный плитками 2×3, каждый ряд которого содержит плитки с буквами на соответствующих местах в любом из слов. Так, если каждое слово начинается с буквы *b* или *m*, то в первом ряду должны находиться две плитки с этими буквами. Аналогично во втором ряду должны быть две плитки с буквами *a* и *e*, а в третьем ряду — плитки с буквами *g* и *t*.

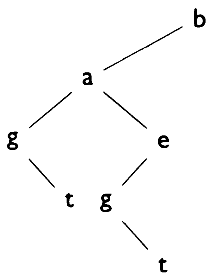
Поиск слова на плиточном полу осуществляется путем систематического обхода плиточного пола ряд за рядом. Для поиска каждой буквы слова обход соответствующего ряда плиток осуществляется слева направо до тех пор, пока буква

не будет найдена. Например, чтобы найти слово *bag* на таком плиточном полу, начнем поиск первой буквы *b* в первом ряду. Нужная плитка обнаруживается за один шаг. Далее мы ищем вторую букву *a* во втором ряду, для чего также требуется один шаг. И наконец, мы можем успешно завершить поиск, найдя букву *g* в третьем ряду, для чего также требуется лишь один шаг. А в целом для данного поиска требуются только три шага сравнения.

Для поиска на плиточном полу требуется меньше шагов, чем в бинарном дереве поиска, поскольку каждую букву приходится сравнивать лишь один раз, тогда как поиск в двоичном дереве приводит к повторяющимся сравнениям первоначальных частей слова. Слово *bag* представляет наиболее благоприятный вариант поиска на плиточном полу, поскольку каждую его букву можно найти на первой плитке. А для поиска слова *met* потребуется шесть шагов, поскольку каждая его буква находится на последней плитке в ряду. Но хуже этого варианта поиска быть не может, поскольку каждую плитку приходится проверять хотя бы один раз. (Для сравнения, чтобы найти слово *met* в бинарном дереве поиска, потребуется 2+3+3+3 = 11 шагов.) Наилучшим вариантом поиска в двоичном дереве является слово *bet*, поскольку для него требуются только три шага сравнения. Тем не менее по мере роста расстояния от корня дерева бинарный поиск приводит к постепенному увеличению шагов сравнения. А поскольку большинство слов располагаются ближе к листьям дерева, т.е. на большем расстоянии от его корня [8], в общем случае сравнение начальных частей слова приходится повторять многократно. Это означает, что, как правило, поиск в префиксном дереве осуществляется быстрее, чем в бинарном дереве поиска.

Аналогия с плиточным полом предполагает, что префиксное дерево может быть представлено в виде таблицы, хотя это верно не всегда. Рассматриваемый здесь пример достигает своей цели только потому, что все слова имеют одинаковую длину и на каждой позиции в слове могут находиться одни и те же буквы. Допустим, однако, что требуется также представить слова *bit* (кусочек) и *big* (большой). В этом случае во втором ряду потребуется дополнительная плитка для буквы *i*, что уже нарушает прямоугольную форму пола. Дополнение данного примера этими двумя словами демонстрирует еще одну закономерность, которая в общем случае не проявляется. Примеры слов выбраны таким образом, чтобы у разных префиксов одинаковой длины были одни и те же возможные суффиксы, что допускает совместное использование информации. Например, оба префикса, *ba* и *be*, могут быть дополнены суффиксом *g* или *t*, а это означает, что возможное продолжение обоих префиксов может быть описано одним рядом плиток с буквами *g* и *t*. Но этого больше нельзя утверждать, если добавить слова *bit* и *big*. Если начало слова с буквы *b* можно продолжить тремя возможными буквами, а именно: *a*, *e* и *i*, то начало слова с буквы *m* — только буквами *a* и *e*. Это означает, что в последнем случае потребуются два разных продолжения.

Таким образом, префиксное дерево, как правило, представлено особого рода двоичным деревом, в узлах которого находятся одиночные буквы, левые поддеревья представляют продолжения слов, а правые поддеревья — альтернативные варианты букв (плюс их продолжения). Например, префиксное дерево для хранения слов *bat*, *bag*, *beg* и *bet* выглядит так, как показано ниже.



У корневого узла дерева отсутствует правое ребро, и поэтому все слова начинаются с буквы *b*. А левое ребро, направленное от корневого узла *b* к поддереву с корневым узлом *a*, приводит к возможным продолжениям, которые начинаются с буквы *a* или буквы *e*, которая служит возможной альтернативой букве *a* и обозначается правым от нее ребром. Возможность продолжить префиксы *ba* и *be* буквой *g* или *t* представлена левым ребром, направленным от буквы *a* к поддереву с корневым узлом *g*, справа от которого находится его дочерний узел *t*. А то, что левые поддеревья узлов *a* и *e* одинаковы, означает, что они могут быть использованы совместно. Именно это обстоятельство используется в плиточном поле, представляющем рассматриваемое здесь дерево в виде единственного ряда плиток. Благодаря такому совместному использованию узлов в префиксном дереве обычно требуется хранить меньше информации, чем в бинарном дереве поиска.

Если в бинарном дереве поиска каждый ключ полностью содержится в отдельном узле, то в префиксном дереве хранящиеся ключи распределяются по всем его узлам. Любая последовательность узлов, исходящих из корня префиксного дерева, служит префиксом какого-нибудь ключа. Так, если речь идет о плиточном поле, то выбранные плитки служат префиксом для конечного пути. Именно поэтому префиксное дерево так и называется. Структура данных, представляющая в виде префиксного дерева плиточный пол, представший перед Индианой Джонсом, показана на рис. 5.2.

Как и у бинарных деревьев поиска и списков, у префиксных деревьев имеются свои преимущества и недостатки. В частности, они пригодны для хранения только тех ключей, которые могут быть разложены на последовательности составляющих элементов. Как в истории Индианы Джонса, которому так и не удалось получить Священный Грааль, так и у структур данных не существует своего Священного Грааля. Выбор наиболее подходящей структуры данных всегда зависит от подробностей ее применения.

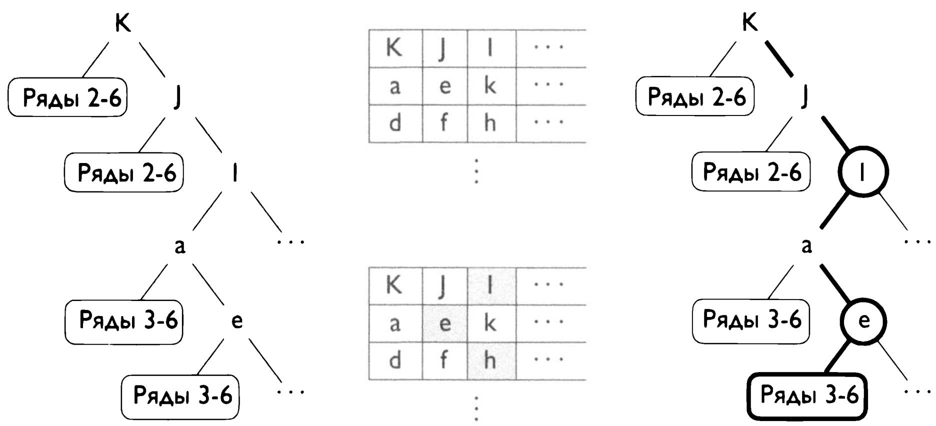


Рис. 5.2. Структура данных префиксного дерева и порядок поиска в ней. Слева: представление префиксного дерева, левые поддеревья которого обозначают (закругленными прямоугольниками) возможные продолжения слова от буквы в родительском узле, а правые поддеревья — их альтернативные варианты. Посредине сверху: представление левой части префиксного дерева в виде плиточного пола, общие поддеревья которого разделяют единые ряды плиток. Посредине внизу: три выбранные плиты, обозначающие начальные буквы слова Iehova. Справа: путь по префиксному дереву, помеченный выделенными полужирным ребрами, ведущими к выбранным плиткам, которые обозначены очерченными кругом узлами

Наблюдая упоминаемую здесь сцену в фильме *Индиана Джонс и последний крестовый поход*, мы понимаем трудность задачи поиска пути по ключевому слову или точного перемещения по целевым плиткам, но едва ли обращаем внимание на распознавание плиток, соответствующих буквам ключевого слова. Ведь это кажется нам настолько очевидным, что мы об этом вообще не думаем. И это обстоятельство еще раз показывает, насколько естественным является для нас выполнение эффективных алгоритмов. Как и в историях Гензеля и Гретель и Шерлока Холмса, спасение Индианы Джонса в его приключениях основывается на базовых принципах вычислений.

И наконец, если вас все еще интересует, как никогда не терять ключи от своей автомашины, знайте, что сделать это совсем не трудно. Придя домой, всегда кладите ключи в одно и то же отведенное для них место. Это место и будет служить ключом для поиска ключей от автомашины. Хотя об этом вам, вероятно, уже давно известно.

Приведение дел в порядок

Если вы преподаватель, то значительная часть вашей работы уходит на проверку сочинений или проведение экзаменов для учащихся в ваших классах. Ваша задача состоит не только в том, чтобы прочитать сочинения отдельных учащихся и принять у них экзамены. Например, прежде чем ставить окончательные оценки, вам потребуется иметь какое-то представление об уже выставленных оценках и кривой их распределения. После выставления оценок от вас потребуется внести их в классный журнал и, наконец, вернуть учащимся их работы. В некоторых случаях каждую из этих задач можно решить более эффективно, выполнив ту или иную сортировку.

Рассмотрим задачу внесения оценок в классный журнал. Даже для решения столь простой задачи можно применить три разных алгоритма с разными показателями времени выполнения. Во-первых, можно просмотреть всю стопку экзаменационных работ и внести оценку за каждую из них в классный журнал. Каждая экзаменационная работа извлекается из стопки за константное время, но поиск фамилии учащегося в классном журнале осуществляется за логарифмическое время, при условии, что он отсортирован по фамилиям, а сам поиск носит бинарный характер. Таким образом, время выполнения задачи по вводу всех оценок в классный журнал приобретает линейно-логарифмический характер [1], что намного лучше, чем квадратичный, хотя и не так хорошо, как линейный.

Во-вторых, можно просматривать классный журнал по фамилиям учащихся, а потом искать экзаменационную работу очередного ученика в стопке. В этом случае перейти к фамилии следующего учащегося в классном журнале можно за константное время, но поиск работы конкретного учащегося в стопке требует просмотра в среднем половины стопки, т.е. квадратичного времени в среднем [2]. Пользоваться таким алгоритмом явно не следует.

В-третьих, можно сначала отсортировать экзаменационные работы по фамилиям, а затем просматривать классный журнал и параллельно — стопку экзаменационных работ. Поскольку стопка отсортирована, а классный журнал упорядочен, для внесения каждой оценки потребуется лишь константное время. Все вместе приводит к тому, что к линейному времени проверки и выставления оценок добавляется время сортировки списка работ, которую можно выполнить за

линейно-логарифмическое (*линарифмическое*[1]) время. В общем времени выполнения данного алгоритма преобладает линейно-логарифмическая составляющая времени, требующегося для сортировки. Таким образом, общее время выполнения этого алгоритма также принимает линейно-логарифмический характер. Получается, такой алгоритм ничуть не лучше первого, так зачем вообще идти на дополнительные хлопоты, связанные с сортировкой экзаменационных работ? Дело в том, что иногда возникают ситуации, когда сортировку можно выполнить быстрее, чем за линейно-логарифмическое время, и тогда третий алгоритм способен повысить производительность и оказаться наиболее эффективным.

Для решения второй задачи — построения кривой распределения экзаменационных оценок — потребуется создание графика распределения баллов, т.е. отображения набранных баллов на количество учащихся, получивших эти баллы. Эта задача подобна задаче Индианы Джонса по вычислению гистограммы букв, поскольку распределение оценок также является гистограммой. Для вычисления такой гистограммы с помощью списка потребуется квадратичное время, а с помощью бинарного дерева поиска — линейно-логарифмическое. Эту гистограмму можно также вычислить, отсортировав сначала стопку экзаменационных работ по баллам, а затем просто просмотрев отсортированную стопку и подсчитав, как часто повторяется та или иная оценка. Такой способ вполне пригоден, поскольку в отсортированном списке за всеми экзаменационными работами с одинаковыми баллами следуют экзаменационные работы с другими, но тоже одинаковыми баллами. Как и при внесении оценок в классный журнал, сортировка не увеличивает время выполнения алгоритма, но потенциально может сократить его. Но совсем другое дело, если диапазон оценок небольшой. В этом случае обновление массива оценок в баллах, вероятно, окажется самым быстрым способом.

И наконец, раздача экзаменационных работ классу может стать чрезвычайно медленным делом. Представьте, что вы стоите перед учащимся, лихорадочно пытаясь найти его экзаменационную работу в большой стопке. Эта задача отнимет немало времени, даже если воспользоваться для ее решения бинарным поиском. Повторение поиска экзаменационных работ большого числа учащихся чрезмерно замедлит весь процесс, даже если он будет ускоряться по мере уменьшения стопки. В качестве альтернативы можно отсортировать экзаменационные работы по местоположению учащихся в классе. Если места учащихся пронумерованы и имеется схема их расположения в классе (или же просто известно, где они сидят), то экзаменационные работы можно отсортировать по номерам мест. В таком случае возврат экзаменационных работ превращается в линейный алгоритм: как и при внесении оценок в классный журнал, можно обойти места со стопкой в руках и

---

[1] Термин, построенный из слов "линейный" и "логарифмический", по аналогии с использующимся в англоязычной литературе "linearithmic", полученным из слов "linear" и "logarithmic". — *Примеч. ред.*

вручить каждую экзаменационную работу за один шаг. И даже если сортировка отнимет линейно-логарифмическое время, она того стоит, поскольку экономится драгоценное время урока. Это характерный пример *предварительного вычисления*, когда некоторые данные, требующиеся для алгоритма, вычисляются до выполнения этого алгоритма. Именно так поступает почтальон, прежде чем разносить почту, или вы сами, когда составляете свой список покупок в соответствии с расположением товаров на полках магазина.

Сортировка оказывается более распространенным явлением, чем может показаться на первый взгляд. Помимо сортировки наборов предметов, нам регулярно приходится решать задачи сортировки в зависимости от обстоятельств. Рассмотрим в качестве примера процесс одевания, когда вы быстро понимаете, что следует придерживаться определенного порядка, надевая трусы до штанов, а носки — до обуви. Сборка мебели, ремонт и техническое обслуживание машины, оформление документов — б*о*льшая часть этих видов деятельности требует выполнения некоторых действий в правильном порядке. Даже дошкольникам дают задания отсортировать картинки в логичный ряд событий в сказках.

6 Сортировка алгоритмов сортировки

Сортировка служит важным примером вычисления. Сортировка не только применяется по прямому назначению, но и помогает объяснить некоторые основополагающие понятия информатики. Во-первых, разные алгоритмы сортировки с разными показателями времени выполнения и требованиями к свободному пространству демонстрируют влияние эффективности на выбор способа решения задачи путем вычисления. Во-вторых, сортировка — это задача, минимальная сложность которой известна заранее. Иными словами, нам точно известна *нижняя граница* количества шагов, которые должен предпринять алгоритм. Таким образом, сортировка наглядно показывает, что в области информатики накладываются свои вполне определенные принципиальные ограничения на скорость вычислений. Знание этих ограничений расширяет наши возможности, поскольку помогает более продуктивно проводить исследования. В-третьих, умение отличать сложность задачи от сложности ее решения помогает лучше уяснить понятие *оптимального решения*. И наконец, в-четвертых, ряд алгоритмов сортировки служат примерами алгоритмов, действующих по принципу *разделяй и властвуй*. Такие алгоритмы разбивают свои входные данные на более мелкие части, которые рекурсивно обрабатываются, и получаемые в итоге решения объединяются в единое решение исходной задачи. Изящество принципа "разделяй и властвуй" объясняется его рекурсивным характером и взаимосвязью с близким ему принципом математический индукции. Этот весьма эффективный способ решения задач наглядно раскрывает истинный потенциал разложения задачи.

Как упоминалось в главе 5, “Поиск идеальной структуры данных”, одним из применений сортировки является поддержка и ускорение поиска. Например, поиск элемента в неотсортированном массиве или списке требует линейного времени, тогда как в отсортированном массиве это можно сделать за логарифмическое время с помощью бинарного поиска. Таким образом, единожды выполненное вычисление (в данном случае — сортировка) может быть сохранено (в виде отсортированного массива) для последующего применения, чтобы ускорить другие вычисления (например, поиск). В более общем случае вычисление является ресурсом, который может быть сохранен и повторно использован через структуры данных. Такая взаимозаменяемость структур данных и алгоритмов наглядно показывает, насколько они взаимосвязаны.

Обо всем по порядку

Основная работа Индианы Джонса — преподавание в колледже, а это означает, что ему приходится решать вопросы, связанные с приемом экзаменов. Следовательно, для его работы вполне уместна сортировка. Будучи археологом, он обязан хранить в организованном порядке коллекции физических артефактов, а также заметки и наблюдения, сделанные им в походах. Подобно документам в учреждении, они немало выиграют от сортировки, поскольку порядок делает поиск конкретных предметов более эффективным. Приключения Индианы Джонса демонстрируют еще один пример применения сортировки, потребность в которой возникает всякий раз, когда он составляет план решения сложной задачи. Такой план состоит в расположении предполагаемых действий в нужном порядке.

Задача Индианы Джонса в фильме *В поисках утраченного ковчега* состоит в том, чтобы найти ковчег с Десятью Заповедями Господними. По слухам, ковчег был спрятан в потайной комнате древнего города Таниса. Чтобы найти ковчег, Индиане Джонсу сначала нужно отыскать потайную комнату, называемую “Кладезем Душ”. Местоположение потайной комнаты может быть обнаружено по модели города Таниса, которая находится в картографическом зале. Если расположить специальный золотой диск в особом месте картографического зала, то можно сфокусировать солнечный свет на местоположении “Кладезя Душ” в модели города Таниса. Золотым диском первоначально обладал бывший учитель и наставник Индианы Джонса, профессор Рэйвенвуд, но впоследствии он передал этот диск своей дочери Мэрион Рэйвенвуд. Следовательно, чтобы найти ковчег, Индиане Джонсу придется выполнить несколько заданий, включая следующие.

- Найти “Кладезь Душ” (`Well`).
- Найти картографический зал (`Map`).
- Заполучить золотой диск (`Disc`).

- Воспользоваться золотым диском, чтобы сфокусировать луч солнца (`Sunbeam`).
- Найти обладателя золотого диска (`Marion`).

Помимо выполнения этих заданий, Индиана Джонс должен посетить разные места: Непал, чтобы найти Мэрион Рэйвенвуд, а также древнеегипетский город Танис, где ковчег спрятан в “Кладезе Душ”.

Правильно упорядочить этот конкретный ряд заданий нетрудно. Но с точки зрения вычисления возникает следующий любопытный вопрос: какие разновидности алгоритмов существуют для решения этой задачи и каковы показатели их времени выполнения? Большинство людей обычно применяют два метода для упорядочения какой-нибудь последовательности. Оба метода начинаются с неотсотированного списка, элементы которого затем перемещаются неоднократно до тех пор, пока список не будет отсортирован. Отличие обоих методов лучше всего показать, описав порядок перемещения элементов из неотсотированного списка в отсортированный.

Первый метод, наглядно показанный на рис. 6.1, состоит в многократно повторяемом поиске наименьшего элемента в неотсортированном списке и последующем его присоединении к отсортированному списку. При сравнении действий одно из них будем считать меньшим, если оно может быть выполнено прежде другого. Следовательно, наименьшим оказывается такое действие, которому не предшествует никакое другое действие. Сначала должны быть обработаны все элементы неотсортированного списка, а отсортированный список пуст. Первое задание состоит в посещении Непала, а это означает, что `Nepal` является наименьшим элементом неотсортированного списка и поэтому должен быть выбран первым и присоединен к отсортированному списку. На следующем шаге необходимо найти обладателя золотого диска, т.е. Мэрион. Далее Индиана Джонс должен заполучить диск, потом отправиться в древний город Танис и обнаружить картографический зал, что и отражено в строках со второй до предпоследней на рис. 6.1. И наконец, Индиана Джонс должен сфокусировать солнечный луч с помощью золотого диска, чтобы обнаружить местоположение “Кладезя Душ”, где он может затем найти ковчег. Отсортированный в итоге список представляет план действий Индианы Джонса в его приключении.

Как показано на рис. 6.1, алгоритм завершается, когда список неотсортированных элементов становится пустым, поскольку в этом случае все элементы перемещены в отсортированный список. Такой алгоритм сортировки называется *сортировкой выбором*, поскольку основан на многократном выборе наименьшего элемента из неотсортированного списка. Следует, однако, иметь в виду, что такой метод может действовать и в обратном направлении, многократно обнаруживая *наибольший* элемент и перенося его в *начало* отсортированного списка.

Рис. 6.1. При сортировке выбором очередной элемент переносится из неотсортированного списка (слева от вертикальной черты) в отсортированный список (справа)

Возможно, это и очевидно, но как же найти наименьший элемент в списке? Для этого проще всего запомнить значение первого элемента и сравнить его со значением второго элемента, третьего и так далее до тех пор, пока не будет найден элемент, меньший первого. В таком случае необходимо взять значение этого элемента и продолжить сравнение и запоминание значения наименьшего в настоящий момент элемента до тех пор, пока не будет достигнут конец списка. Такой метод сортировки занимает линейное время, соответствующее в худшем случае длине списка, поскольку наименьший элемент может быть последним в списке.

Б*о*льшая часть усилий при сортировке выбором расходуется на поиск наименьшего элемента в неотсортированном списке. Несмотря на то что неотсортированный список на каждом шаге данного алгоритма сортировки сокращается на один элемент, общее время выполнения такого алгоритма по-прежнему остается квадратичным, поскольку в среднем приходится обходить список, содержащий половину элементов. Такой же шаблон был выявлен и при анализе алгоритма Гензеля в главе 2 для выбора следующего камешка, принесенного из дому: сумма первых n чисел пропорциональна квадрату n [1].

Еще один распространенный метод сортировки состоит в том, чтобы извлечь произвольный (обычно первый) элемент из неотсортированного списка и расположить его в правильной позиции в отсортированном списке. Такой элемент вставляется *после* того наибольшего элемента в отсортированном списке, который оказывается *меньше* вставляемого элемента. Иными словами, на шаге вставки осуществляется обход отсортированного списка с целью найти последний элемент, который должен предшествовать вставляемому.

Усилия в данном алгоритме сортировки расходуются на вставку элементов, а не на их выбор, и поэтому данный алгоритм называется *сортировкой вставками* (рис. 6.2). Сортировку вставками предпочитают многие игроки в карты. Имея перед собой сданные карты, они выбирают их одну за другой и вставляют в уже отсортированные карты на руках.

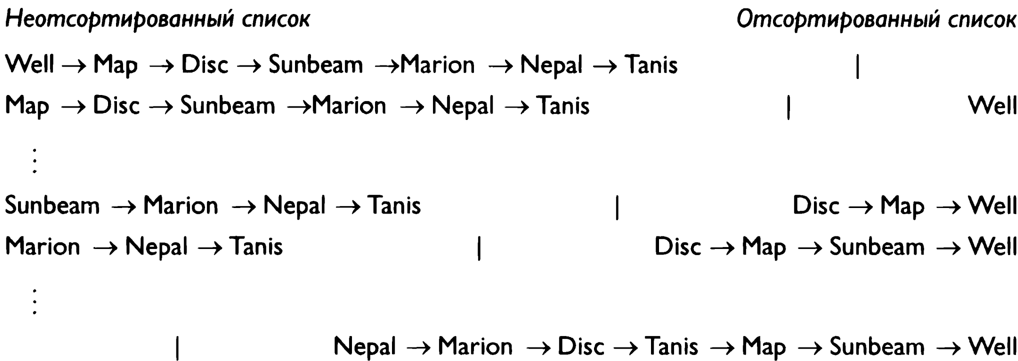


Рис. 6.2. При сортировке вставками следующий элемент вставляется из неотсортированного списка (слева от вертикальной черты) в отсортированный список (справа)

Отличие сортировки вставками от сортировки выбором лучше всего видно, когда элемент `Sunbeam` перемещается из неотсортированного списка в отсортированный. Как показано на рис. 6.2, этот элемент просто удаляется из неотсортированного списка, не требуя никакого поиска, а затем вставляется в отсортированный список путем обхода последнего до тех пор, пока не будет найдено подходящее для него место между элементами `Map` и `Well`.

Данный пример демонстрирует также незначительное отличие во времени выполнения обоих алгоритмов сортировки. Если при сортировке выбором приходится полностью обходить один из двух списков для каждого выбранного элемента, то при сортировке вставками это делается только в том случае, когда вставляемый элемент больше всех элементов, имеющихся в настоящий момент в отсортированном списке. В худшем случае, когда сортируемый список уже отсортирован изначально, каждый элемент будет вставляться в конец отсортированного списка. А если список изначально отсортирован в обратном порядке, т.е. от наибольшего к наименьшему элементу, то каждый элемент будет вставляться в начало отсортированного списка, что в итоге приведет к линейному времени выполнения сортировки вставками. Можно показать, что в среднем время выполнения сортировки вставками по-прежнему носит квадратичный характер. Иногда, в зависимости от конкретных входных данных, сортировка вставками выполняется намного быстрее, чем сортировка выбором, но она никогда не выполняется медленнее.

Почему же показатель времени выполнения сортировки вставками иногда оказывается лучше, чем у сортировки выбором, несмотря на то что оба эти алгоритма действуют сходным образом? Главное отличие состоит в том, что при сортировке вставками используется результат ее собственного вычисления. А поскольку новые элементы вставляются в уже отсортированный список, для их вставки не всегда приходится обходить весь список. При сортировке же выбором элемент всегда добавляется в конец отсортированного списка, а поскольку в

процессе выбора нельзя употреблять результат сортировки, приходится постоянно просматривать весь неотсортированный список. Такое сравнение обоих алгоритмов сортировки наглядно показывает очень важный для информатики принцип *повторного использования.*

Эти алгоритмы сортировки различаются не только эффективностью, но и пригодностью для решения задачи упорядочения плана действий. Ни один из них не идеален, поскольку основная трудность в данном примере задачи состоит не в самой сортировке, а в том, чтобы решить, какие действия должны предшествовать другим действиям. Если нет уверенности, как именно следует упорядочить элементы в списке, то сортировка выбором кажется менее привлекательной, поскольку уже на первом шаге приходится решать, как сравнить предположительно наименьший элемент со всеми остальными. В этом отношении сортировка вставками лучше, поскольку на первом шаге выбирается произвольный элемент без всяких сравнений, а значит, можно сразу же получить отсортированный одноэлементный список. Но на последующих шагах каждый выбранный элемент придется сравнивать со все б*о*льшим и б*о*льшим числом элементов в растущем отсортированном списке, чтобы найти для него подходящее место в этом списке. Хотя поначалу количество трудных сравнений может оказаться для сортировки вставками меньшим, чем для сортировки выбором, этот алгоритм может заставить принимать решения, которые, возможно, пока еще не могут быть приняты. Нет ли алгоритма, дающего сортирующему больший контроль над сравнением элементов?

Разбивайте, не стесняясь

Желательно, чтобы алгоритм сортировки позволял откладывать трудные решения и начинать с решений простых. Например, из плана Индианы Джонса отыскать потерянный ковчег Завета ясно, что "Кладезь Душ" и картографический зал находятся в древнем городе Танис, и поэтому каждое действие, связанное с этими двумя местами, должно следовать после стадии поездки в город Танис, а все остальное должно ему предшествовать.

Разделяя элементы списка на последующие и предшествующие выделенному элементу, называемому *опорным*, мы должны разбить один неотсортированный список на два других неотсортированных списка. А чего мы этим добьемся? И хотя это действие еще не означает никакой сортировки, оно все же позволяет достичь двух важных целей. Во-первых, мы разложили задачу сортировки одного длинного списка на два более коротких списка для последующей сортировки. Ведь упрощение задачи нередко оказывается важным шагом в сторону ее решения. И во-вторых, как только будут решены две подзадачи, т.е. отсортированы два неотсортированных списка, их можно просто соединить вместе, чтобы получить единый отсортированный список. Иными словами, разложение на подзадачи упрощает составление решения общей задачи из решений двух подзадач.

Рассмотрим последовательность формирования двух неотсортированных списков. Обозначим меткой *S* список элементов, которые меньше элемента `Tanis`, а меткой *L* — список тех элементов, которые больше элемента `Tanis`. Теперь нам известно, что все элементы в списке *S* меньше всех элементов в списке *L*, хотя оба списка *S* и *L*, пока еще не отсортированы. Но как только списки *S* и *L* будут отсортированы, то отсортированным окажется и список, полученный в результате объединения списка *S*, элемента `Tanis` и списка *L*. Поэтому остается лишь решить задачу сортировки списков *S* и *L*. И как только это будет сделано, останется лишь объединить полученные результаты. Как же отсортировать эти более мелкие списки? Для этой цели можно выбрать любой алгоритм сортировки. Выбранный метод разделения и соединения можно применять рекурсивно, а если списки достаточно малы, воспользоваться простым алгоритмом вроде сортировки выбором или вставками.

В 1960 году британский специалист по информатике Тони Хоар (Tony Hoare)[1] изобрел основанный на этом принципе алгоритм *быстрой сортировки*. На рис. 6.3 принцип действия быстрой сортировки наглядно показан на примере составления Индианой Джонсом плана поиска утерянного ковчега Завета. На первом шаге данного алгоритма неотсортированный список разбивается на два других списка, разделяемых опорным элементом. Каждый из них состоит только из трех элементов, и поэтому их легко отсортировать, используя любой другой алгоритм.

Well → Map → Disc → Sunbeam →Marion → Nepal → Tanis

Disc → Marion → Nepal | Tanis | Well → Map → Sunbeam

Nepal → Marion → Disc | Tanis | Map → Sunbeam → Well

Nepal → Marion → Disc → Tanis → Map → Sunbeam → Well

Рис. 6.3. При быстрой сортировке исходный список сначала разбивается на два подсписка элементов, которые соответственно меньше и больше выбранного опорного элемента. Затем оба подсписка сортируются, а полученные результаты объединяются с опорным элементом для формирования результирующего отсортированного списка

Ради наглядности отсортируем подсписок элементов, меньших элемента `Tanis`, используя быструю сортировку. Если выбрать элемент `Nepal` в качестве разделяющего, то после собирания элементов, б*о*льших элемента `Nepal`, получится неотсортированный список элементов `Disc` → `Marion`. Соответственно, список элементов, меньших элемента `Tanis`, окажется пустым, т.е. заведомо отсортированным. Чтобы отсортировать этот двухэлементный список, достаточно сравнить два его элемента и переставить их местами, получив в итоге отсортированный список элементов `Marion` → `Disc`. Затем пустой список, элемент `Nepal` и отсортированный список элементов `Marion` → `Disc` следует объединить, чтобы

[1] Официально — сэр Чарльз Энтони Ричард Хоар (Sir Charles Antony Richard Hoare).

получить отсортированный подсписок элементов `Nepal` → `Marion` → `Disc`. Если выбрать любые другие элементы в качестве опорных, сортировка действует аналогично. Так, если выбрать опорным элемент `Disc`, то и в этом случае получатся пустой список и двухэлементный список, который требуется отсортировать. А если выбрать опорным элемент `Marion`, то в итоге получатся два одноэлементных списка, которые уже отсортированы. Точно так же сортируется и подсписок элементов, б*о*льших элемента `Tanis`.

Как только подсписки будут отсортированы, их можно объединить с элементом `Tanis` посредине и получить конечный результат. Как показано на рис. 6.3, быстрая сортировка сходится удивительно быстро. Но всегда ли это так? Каково общее время выполнения быстрой сортировки? Нам, по-видимому, повезло в выборе элемента `Tanis` в качестве опорного на первом шаге, поскольку элемент `Tanis` разделяет исходный список на два других списка равной длины. Если бы можно было всегда выбирать опорный элемент с таким свойством, то подсписки неизменно разделялись бы надвое, а это означает, то количество элементов на уровне разделения будет пропорционально логарифму длины исходного списка. А что это означает для общего времени выполнения быстрой сортировки в данном и общем случаях?

Первый шаг цикла, на котором исходный список разбивается на два других списка, отнимает линейное время, поскольку на этом шаге приходится проверять все элементы в списке. На следующем шаге цикла приходится разбивать два подсписка, что в целом отнимет снова линейное время, поскольку общее количество элементов в обоих списках на единицу меньше, чем в исходном списке, независимо от места разбиения и длины каждого из двух подсписков [2]. И по такой схеме рассматриваемый здесь алгоритм действует дальше: на каждом уровне оказывается не больше элементов, чем на предыдущем уровне, а значит, для его разбиения требуется линейное время. В целом на каждом уровне разбиения накапливается линейное время, а это означает, что время выполнения быстрой сортировки зависит от количества уровней, требующихся для разбиения подсписков. Как только исходный список будет полностью разложен на отдельные элементы, процесс разбиения обеспечит правильный порядок расположения всех этих элементов, которые останется лишь соединить в отсортированный конечный список. На это снова потребуется линейное усилие, а следовательно, общее время выполнения быстрой сортировки составит сумму затрат линейного времени на каждом уровне разбиения.

Если разбивать каждый подсписок приблизительно пополам, то в лучшем случае получится логарифмическое количество уровней. Например, список из 100 элементов разбивается на семь уровней, список из 1000 элементов — на десять уровней, а список из 1 000 000 элементов может быть полностью разложен на 20 уровней [3]. С точки зрения общего времени выполнения это означает,

что для просмотра списка из 1 000 000 элементов придется выполнить линейное действие лишь 20 раз. Таким образом, линейное время выполнения оказывается порядка десятка миллионов шагов, что намного лучше квадратичного времени выполнения, которое составляет от сотен миллиардов до триллионов шагов. Такая зависимость времени выполнения, когда оно растет пропорционально объему входных данных, умноженному на логарифм этого объема, называется *линейно-логарифмической* (*линарифмической*). И хотя такая зависимость времени выполнения хуже линейной, она все же куда лучше квадратичной (рис. 6.4).

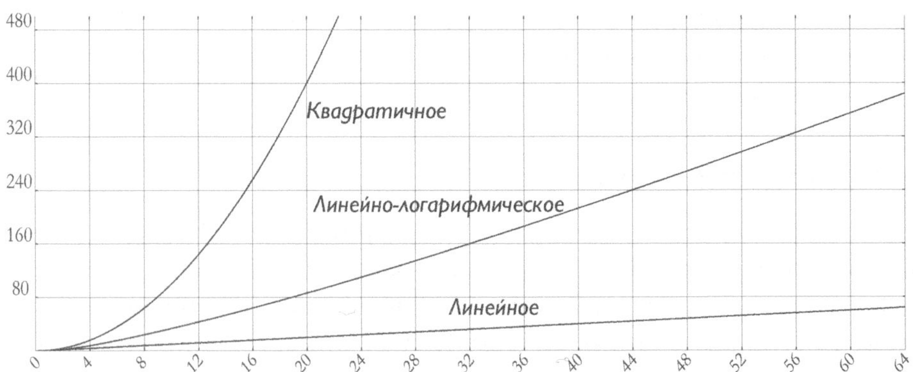


Рис. 6.4. Сравнение квадратичного, линейно-логарифмического и линейного времен выполнения

В лучшем случае время выполнения быстрой сортировки имеет линейно-логарифмический характер. Но если выбирать опорный элемент неблагоразумно, то возникнет совсем иная ситуация. Так, если выбрать первым опорным элемент `Nepal`, а не `Tanis`, то подсписок меньших элементов окажется пустым, а подсписок бо́льших элементов будет состоять из всех элементов, кроме `Nepal`. Если далее выбрать опорным элемент `Marion` в данном подсписке, то возникнет та же самая ситуация, когда один подсписок оказывается пустым, а другой — лишь на один элемент короче разбиваемого подсписка. Как видите, в данной ситуации быстрая сортировка, по существу, ведет себя таким же образом, как и сортировка выбором, когда из неотсортированного списка многократно удаляется наименьший элемент. И подобно сортировке выбором, быстрая сортировка потребует в данном случае квадратичного времени выполнения. Эффективность быстрой сортировки зависит от выбора опорного элемента. Так, если опорный элемент всегда оказывается посредине, то в результате разбиения получаются равномерно разделенные списки. Но как найти подходящий опорный элемент? Несмотря на то, что гарантировать обнаружение подходящих опорных элементов нелегко, в среднем выбор случайного элемента (или медианы из первого, среднего и последнего элементов) списка оказывается вполне подходящим.

Важность и влияние опорного элемента тесно связаны с понятием границы, которое было введено в главе 5, “Поиск идеальной структуры данных”, для объяснения сущности эффективного поиска. Назначение границы при поиске состоит в том, чтобы разделить область поиска на внешнюю и внутреннюю части и сделать внутреннюю часть как можно меньшей, упростив тем самым поиск. А при сортировке граница должна разделять области сортировки на равные части, чтобы в достаточной мере упростить сортировку каждой части через разложение на составляющие. Так, если опорный элемент выбран неудачно, то время выполнения быстрой сортировки заметно ухудшится. Но, несмотря на то что время выполнения быстрой сортировки приобретает в худшем случае квадратичный характер, в среднем оно имеет линарифмический характер, что вполне удовлетворяет требованиям практики.

Лучшее еще впереди

Имеются ли более быстродействующие алгоритмы, чем быстрая сортировка, например алгоритм с линейно-логарифмическим или еще лучшим временем выполнения в худшем случае? Да, имеются. Один из таких алгоритмов называется *сортировкой слиянием*, которая была изобретена венгерско-американским математиком Джоном фон Нейманом (John von Neumann) в 1945 году [4]. При сортировке слиянием неотсортированный список разбивается на два других списка, так что решаемая задача разлагается на более мелкие части, подобно тому, как это дается при быстрой сортировке. Но на данном шаге элементы при сортировке слиянием не сравниваются, а осуществляется только разделение списка на две равные части. Как только оба подсписка будут отсортированы, их можно слить в один отсортированный список, обходя их параллельно и поочередно сравнивая первые элементы двух подсписков. При сравнении первых элементов двух подсписков выбирается наименьший из них. Благодаря тому что оба подсписка отсортированы, таковым оказывается и объединенный список. Но каким образом два подсписка, получающихся на стадии разбиения, оказываются отсортированными? Это достигается путем рекурсивного применения алгоритма сортировки слиянием к обоим спискам. Принцип действия сортировки слиянием наглядно показан на рис. 6.5.

Несмотря на то что алгоритмы быстрой сортировки и сортировки слиянием вполне пригодны для программирования электронных вычислительных машин, ими, не прибегая к посторонней помощи, нелегко пользоваться людям с их слабой памятью — поскольку эти алгоритмы требуют немало служебных операций. В частности, для сортировки крупных списков в обоих алгоритмах приходится поддерживать потенциально большую совокупность мелких списков, а при быстрой сортировке эти списки должны еще и храниться в правильном порядке.

Поэтому Индиана Джонс, как и большинство из нас, выбрал бы, вероятно, какую-нибудь форму сортировки вставками, скорее всего, усиленную рациональным выбором элементов, если только длина сортируемого списка не потребовала бы более эффективного алгоритма. Например, всякий раз, когда я веду урок в крупном классе колледжа, то пользуюсь вариантом *блочной сортировки*, чтобы отсортировать экзаменационные работы по фамилиям учащихся. С этой целью я распределяю экзаменационные работы по разным папкам, называемым *блоками*, исходя из первой буквы в фамилии учащегося, а каждый блок поддерживаю в отсортированном порядке с помощью сортировки вставками. Как только все экзаменационные работы будут распределены по своим блокам, эти блоки затем располагаются в алфавитном порядке, образуя упорядоченный список. Блочная сортировка подобна рассматриваемой далее сортировке подсчетом.

[1] Well → Map → Disc → Sunbeam → Marion → Nepal → Tanis

[2] Well → Map → Disc → Sunbeam | Marion → Nepal → Tanis

[3] (Well → Map | Disc → Sunbeam) | (Marion → Nepal | Tanis)

[4] ((Well | Map) | (Disc | Sunbeam)) | ((Marion | Nepal) | Tanis))

[5] (Map → Well | Disc → Sunbeam) | (Nepal → Marion | Tanis)

[6] Disc → Map → Sunbeam → Well | Nepal → Marion → Tanis

[7] Nepal → Marion → Disc → Tanis → Map → Sunbeam → Well

Рис. 6.5. При сортировке слиянием исходный список сначала разбивается на два подсписка равной длины, затем они сортируются и сливаются в один отсортированный список. В скобках указан порядок слияния списков. В строке 4 разложение на составляющие завершается (получаются только одноэлементные списки). В строке 5 три пары одноэлементных списков были объединены в три отсортированных двухэлементных списка. В строке 6 эти списки снова объединяются в четырехэлементный и трехэлементный списки, а на последнем шаге эти списки объединяются для получения конечного результата

На первый взгляд, сортировка слиянием выглядит более сложной, чем быстрая сортировка, но это впечатление можно объяснить тем, то в приведенном выше описании быстрой сортировки были опущены некоторые шаги. Тем не менее повторяющееся слияние все более длинных списков может показаться неэффективным, хотя на самом деле это впечатление обманчиво. Разложение на составляющие производится систематически, разделяя надвое длину списков на каждом шаге, и поэтому общий показатель времени выполнения сортировки слиянием оказывается довольно неплохим: в худшем случае время выполнения сортировки слиянием приобретает линейно-логарифмический характер. Это хорошо видно из следующего. Во-первых, списки всегда разбиваются надвое, и поэтому количество требующихся разбиений списков растет по логарифмическому закону. Во-вторых, слияние на каждом уровне отнимает лишь линейное время,

поскольку каждый элемент списка приходится обрабатывать только один раз (см. рис. 6.5). И наконец, слияние на каждом уровне происходит лишь однажды, и поэтому общее время выполнения данного вида сортировки получается линейно-логарифмическим.

Сортировка слиянием в чем-то похожа на быструю сортировку. В частности, оба алгоритма сортировки включают в себя стадию разбиения списков, после которой следует стадия рекурсивной сортировки каждого из более мелких списков, и наконец, стадия объединения отсортированных подсписков в более крупные отсортированные списки. В действительности сортировка слиянием и быстрая сортировка служат примерами алгоритмов разбиения, а те, в свою очередь, — примерами следующей общей схемы.

Если задача простая, решить ее непосредственно.

Иначе:

1. Разложить задачу на подзадачи.
2. Решить подзадачи.
3. Объединить решения подзадач в единое решение задачи.

Решение нетривиальной задачи демонстрирует применение метода "разделяй и властвуй". Первым выполняется шаг *разделения*, на котором степень сложности задачи снижается. На втором шаге отдельные подзадачи решаются рекурсивно. Если полученные таким образом подзадачи достаточно малы, они решаются непосредственно. В противном случае они разлагаются далее до тех пор, пока не станут достаточно мелкими для их непосредственного решения. И наконец, третьим выполняется шаг *слияния*, на котором решение исходной задачи составляется из решений отдельных подзадач.

Б*о*льшая часть быстрой сортировки выполняется на шаге разделения, на котором происходит сравнение всех элементов. Благодаря тому что элементы в обоих списках разделяются опорным элементом, на шаге слияния остается лишь соединить эти списки. А при сортировке слиянием шаг разделения оказывается очень простым и не содержит никаких операций сравнения. Б*о*льшая часть сортировки слиянием выполняется на шаге слияния, на котором отсортированные списки объединяются, как с помощью застежки "молния".

Поиск завершен: лучших алгоритмов сортировки нет

Быстрая сортировка является самым эффективным из всех обсуждавшихся до сих пор алгоритмов сортировки, но нельзя ли сортировать еще быстрее? И да, и нет. Хотя в общем случае сортировать быстрее нельзя, при некоторых допущениях относительно сортируемых элементов положение можно улучшить. Так, если

заранее известно, что сортируемый список содержит только числа в пределах от 1 до 100, можно создать массив из 100 элементов: по одному на каждое потенциальное число в списке. Такой подход подобен блочной сортировке, при которой на каждую букву алфавита приходится одна стопка, а здесь элемент массива соответствует стопке, содержащей отдельное число. Элементы массива индексируются от 1 до 100, причем каждый массив с индексом *i* используется для подсчета количества появлений *i*-го элемента в списке. Сначала в каждом элементе массива содержится нулевое значение, поскольку неизвестно, какие именно числа должны быть в отсортированном списке. Затем выполняется обход списка и для каждого *i*-го элемента число, хранящееся в элементе массива по индексу *i*, увеличивается. И наконец, выполняется обход массива в порядке увеличения индексов и каждый индекс вносится в итоговый список столько раз, сколько указано в счетчике элемента. Так, если требуется отсортировать список 4→2→5→4→2→6, то после его обхода получится следующий массив.

0	2	0	2	1	1	0	···	*Счетчик*
1	2	3	4	5	6	7	···	*Индекс*

Обойдя массив, можно обнаружить, что число 1 отсутствует в списке, поскольку его счетчик остался равным нулю. Следовательно, число 1 не входит в отсортированный итоговый список. Напротив, число 2 дважды встречается в списке и поэтому столько же раз вводится в отсортированный итоговый список и т.д. Таким образом, итоговый список будет выглядеть как 2→2→4→4→5→6. Такой алгоритм называется *сортировкой подсчетом*, поскольку массив содержит счетчик, указывающий, как часто элемент встречается в сортируемом списке. Время выполнения сортировки подсчетом складывается из затрат на обход списка и массива. А поскольку каждый шаг данного алгоритма линейно зависит от размера соответствующей структуры данных (списка и массива), то сортировка подсчетом выполняется в течение линейного времени, пропорционального размеру списка или массива, в зависимости от того, какой из них больше.

Недостаток сортировки подсчетом заключается в том, что она может напрасно расходовать немало свободного пространства. В рассмотренном выше примере элементы массива с индексами от 7 и до 100 вообще не использовались. Кроме того, данный алгоритм сортировки действует лишь в том случае, если элементы списка могут быть использованы для индексации массива и если диапазон этих элементов заранее известен и не особенно велик. Например, с помощью сортировки подсчетом нельзя отсортировать список имен, поскольку имена представляют собой последовательности символов и не могут быть использованы для индексации массива [5]. Впрочем, имеются другие алгоритмы, специально предназначенные для сортировки списков символьных строк. Например, для сортировки списков символьных строк может быть использовано префиксное дерево

(см. главу 5, “Поиск идеальной структуры данных”), но такие алгоритмы используют другие допущения относительно сортируемых элементов.

Лучше не бывает

Не употребляя специальных свойств данных, нельзя отсортировать их быстрее, чем с помощью алгоритма сортировки слиянием. И хотя это обстоятельство может поначалу вызвать разочарование, оно же и обнадеживает, поскольку заверяет нас в том, что мы нашли самый быстрый алгоритм сортировки. Иными словами, сортировка слиянием является наилучшим из возможных решением задачи сортировки, а значит, и *оптимальным алгоритмом*. Исследователи, работающие в области информатики, могут считать задачу решенной и потратить свое время и энергию на решение других задач.

Оптимальность сортировки слиянием опирается на два взаимосвязанных, но отдельных факта. Время выполнения сортировки слиянием имеет линейно-логарифмический характер, но и любой алгоритм сортировки в общем случае должен иметь как минимум линейно-логарифмическое время выполнения. Именно этот второй факт подтверждает вывод об оптимальности сортировки слиянием, а заодно и любого другого алгоритма сортировки с линейно-логарифмическим временем выполнения в худшем случае. Конечная цель разработки алгоритмов и структур данных состоит в том, чтобы найти оптимальный алгоритм для решения задачи, т.е. такой алгоритм, временн*ая* сложность которого в худшем случае соответствует внутренней сложности задачи, которую он решает. Любой подобный алгоритм можно считать Священным Граалем для данной задачи. И подобно Индиане Джонсу в истории из фильма *Индиана Джонс и последний крестовый поход*, Джон фон Нейман нашел в сортировке слиянием Священный Грааль сортировки [6].

Очень важно различать время выполнения алгоритма и сложность решаемой задачи. Последняя означает, что для правильного решения необходимо предпринять *как минимум* указанное количество шагов. А время выполнения алгоритма означает, что для выполнения конкретного алгоритма требуется *не больше* чем указанное количество шагов. Утверждение о минимальной сложности задачи называется *нижней границей*. Нижняя граница позволяет оценить, каких минимальных затрат труда потребует решение задачи, а следовательно, характеризует присущую задаче сложность. Это можно сравнить с геометрическим расстоянием между двумя точками, которое служит минимальной границей для любого пути, соединяющего обе точки. Любой подобный путь может оказаться длиннее геометрического расстояния из-за возможных препятствий, но он в принципе не может быть короче этого расстояния. Разочарование, которое может возникнуть в связи с наличием нижней границы сортировки и связанным с ней ограничением

алгоритмов сортировки, должно быть возмещено глубоким пониманием, которое дает такое знание задачи сортировки. Это можно сравнить с аналогичными результатами из других дисциплин. Например, из физики известно, что нельзя передвигаться быстрее света или создать энергию из ничего.

Но как мы можем быть уверены в нижней границе сортировки? Вдруг имеется алгоритм, который еще никто не придумал и который действительно выполняется быстрее, чем в течение линейно-логарифмического времени? Доказать противное нелегко, поскольку для этого придется показать, что любой алгоритм, существует ли он или должен быть еще изобретен, должен быть выполнен за определенное минимальное количество шагов. Аргумент в пользу нижней границы сортировки учитывает количество возможных списков определенной длины [7] и показывает, что количество сравнений, требующихся для получения отсортированного списка, является линейно-логарифмическим [8]. А поскольку в каждом алгоритме должно быть выполнено это минимальное количество сравнений, то получается, что для выполнения любого алгоритма сортировки требуется как минимум линейно-логарифмическое количество шагов, что и доказывает наличие нижней границы.

Рассуждая о времени выполнения алгоритмов и нижних границах, следует принимать во внимание возможности компьютера, выполняющего алгоритмы. Например, как правило, принимается, что шаги алгоритма выполняются последовательно и что для выполнения одного вычислительного шага требуется одна единица времени. Подобные допущения положены также в основу анализа алгоритмов сортировки, обсуждаемых в этой главе. Но если допустить, что сравнения могут выполняться параллельно, то анализ несколько меняется, и мы получаем другие результаты для показателей времени выполнения и нижних границ.

Сохранение вычислений

По-видимому, Индиана Джонс довольно организованно готовится к своим приключениям: он всегда укладывает в дорожную сумку свою шляпу и бич. Но вместо того чтобы составлять подробный план всех своих действий, он мог бы подходить к своим приключениям иначе, определяя свой следующий шаг непосредственно перед тем, как его потребуется совершить. У обоих подходов (предварительного планирования и стремления жить одним мгновением) имеются свои достоинства и недостатки. Так, для путешествия на Северный полюс требуется иная одежда и экипировка, чем для путешествия на Амазонку. И здесь предварительное планирование кажется здравой мыслью. Но, с другой стороны, изменившиеся обстоятельства способны нарушить прежние планы, сведя на нет все усилия по предварительному планированию. В частности, во время приключения может произойти нечто столь неожиданное, что придется предпринять совсем иные действия, чем предусматривалось.

Если бы стратегия Индианы Джонса для поиска утерянного ковчега Завета состояла в том, чтобы определять следующее действие всякий раз, когда в этом возникнет потребность, он, по существу, выполнял бы сортировку выбором и всегда искал бы наименьший элемент среди действий, которые осталось выполнить, поскольку *наименьший* в данном случае означает "то, что должно предшествовать всем остальным действиям". Но, как обсуждалось ранее, это не очень эффективный способ, поскольку сортировка выбором является квадратичным алгоритмом. Индиана Джонс может добиться намного большего, если заранее составит план, использовав линейно-логарифмический алгоритм сортировки вроде сортировки слиянием. Такая стратегия заблаговременного вычисления информации называется *предварительным вычислением*, или попросту *предвычислением* (precomputation). Так, если речь идет о плане отыскания утерянного ковчега Завета, то предварительно вычисленной информацией является не ряд отдельных шагов, а их расположение в правильном порядке.

Упорядоченная информация хранится в отсортированном списке. Крайне важная особенность предварительного вычисления состоит в том, что вычислительные усилия затрачиваются в один момент времени, а вычисленный результат — в другой. Предварительно вычисленный результат сохраняется в некоторой структуре данных (в данном случае — в отсортированном списке). Этот отсортированный список действует как аккумуляторная вычислительная батарея, которую можно зарядить через сортировку. Время выполнения, затрачиваемое алгоритмом на составление отсортированного списка как структуры данных, приравнивается к энергии, используемой для заряда аккумуляторной батареи. Этой энергией можно воспользоваться для подпитки вычисления, например, для поиска следующего наименьшего элемента в списке. Здесь подпитка означает ускорение поиска, поскольку без аккумуляторной батареи отсортированного списка для поиска наименьшего элемента потребуется линейное время, а при наличии такой батареи — лишь константное время.

Одни способы зарядки аккумуляторных батарей структур данных оказываются более эффективными, чем другие, что отражают разные показатели времени выполнения различных алгоритмов сортировки. Например, сортировка вставками менее эффективна, чем сортировка слиянием, а то обстоятельство, что сортировка слиянием является оптимальным алгоритмом сортировки, означает, что для зарядки аккумуляторной батареи отсортированного списка существует наиболее эффективный метод. Но, в отличие от аккумуляторных батарей электропитания, энергия заряда которых может быть израсходована лишь один раз, аккумуляторные батареи структур данных обладают следующим замечательным свойством: зарядившись один раз, они могут разряжаться многократно, не требуя перезарядки. И это свойство служит очень важным аргументом в пользу предварительного

вычисления. В тех случаях, когда предполагается неоднократное использование структуры данных, возникает сильный соблазн расходовать усилия на предварительное вычисление, поскольку понесенные при этом затраты могут быть погашены за несколько использований сохраненных результатов. С другой стороны, чтобы работать должным образом, аккумуляторная батарея структур данных должна быть полностью заряжена. Не полностью отсортированного списка недостаточно, поскольку в этом случае не гарантируется, что наименьший элемент окажется в начале списка, а следовательно, могут быть получены неверные результаты. По существу, это означает, что аккумуляторная батарея структур данных находится в одном из двух следующих состояний: полностью заряженном и полезном или полностью разряженном и совершенно бесполезном.

Предварительное вычисление кажется отличной мыслью, подобно белке, заготавливающей орешки про запас на зиму. Но имеется немало случаев, когда неясно, окупятся ли усилия, затраченные на предварительное вычисление. Известны ситуации, когда заблаговременные действия дают преимущество, но это ничего не гарантирует и может фактически превратиться в недостаток. Если вы заранее приобретаете авиабилет или бронируете номер в гостинице по невозмещаемому тарифу, то можете совершить выгодную сделку. Но если при этом вы заболеете и будете не в состоянии отправиться в путешествие, то можете потерять свои деньги и сделать хуже, чем если бы вы подождали с приобретением билета или бронированием номера.

В подобных случаях ценность заблаговременных действий или предварительного вычисления оказывается под сомнением из-за неопределенности будущего. А поскольку преимущество предварительного вычисления зависит от конкретного исхода будущих событий, то оно отражает оптимистическое отношение к вычислению с уверенными видами на будущее. Но что, если у вас скептические виды на будущее? Вы можете считать мазохистами людей, заранее заполняющих свои налоговые декларации, и всегда откладываете это дело на крайний срок, поскольку налоговая служба может быть упразднена в любой момент, или вы можете умереть до срока подачи декларации. Но если вы, как и Бенджамин Франклин, убеждены, что в этой жизни нет ничего постоянного, кроме налогов и смерти, то предварительное вычисление может показаться вам вполне благоразумным делом.

Скептическое отношение к будущему требует совершенно иной стратегии планирования вычислений, т.е. такой стратегии, при которой предпринимается попытка отложить затратные операции до тех пор, когда избежать их уже никак нельзя. Такая стратегия основывается на надежде или ожидании, что может произойти нечто такое, что сделает затратную операцию неактуальной, а значит, позволит сэкономить на времени ее выполнения (а возможно, и на других ресурсах). В реальной жизни такое поведение называлось бы отсрочкой, а в информатике оно называется *отложенным* (или *ленивым*) *вычислением*. Отложенное вычисление

позволяет экономить затраты на вычисление всякий раз, когда информация, которую можно было бы получить из сэкономленного вычисления, оказывается ненужной. Например, в приключениях Индианы Джонса очень часто случается так, что первоначальный план приходится менять или вообще отвергать из-за непредвиденных обстоятельств или осложнений. В подобных случаях все усилия, потраченные на составление плана, оказываются напрасными, и их можно было бы сберечь, вообще не ничего не планируя.

Если девиз приверженцев предварительного вычисления — "Семь раз отмерь, один раз отрежь" или "Кто рано встает, тому Бог подает ", то сторонники отложенного вычисления отвечают на это "Сыр достается второй мышке" или "Быстрая вошка первой на гребешок попадает". Несмотря на то что отложенное вычисление привлекает обещанием не тратить усилия попусту, польза от него оказывается сомнительной, когда становящееся неизбежным действие отнимает больше времени, чем при предварительном вычислении, или того хуже — если оно отнимает больше времени, чем имеется. В частности, когда наступает срок выполнения одновременно нескольких отложенных действий, это может вызвать серьезные затруднения в отношении доступных ресурсов. Следовательно, в общем случае более разумной стратегией является равномерное распределение работы по времени. И хотя для этого придется затратить некоторые усилия на предварительное вычисление, в конечном счете такая стратегия позволит избежать кризиса, к которому могло бы привести отложенное вычисление.

На обед

Настала среда — день недели, когда вы договорились с некоторыми из коллег по работе вместе пообедать. Вы решили попробовать кухню в новом итальянском ресторанчике, но, придя в него, узнали, что устройство считывания кредитных карточек не работает, а плата в этот день принимается только наличными. Прежде чем сделать заказ, вы подсчитываете, сколько наличных вам придется заплатить за обед. А когда дело доходит до самого заказа, то вам приходится решать задачу выбора блюд (закусок, основных блюд, гарниров и напитков), чтобы накормить всех участников обеда и удовлетворить их вкусы как можно точнее, но в то же время не превысить бюджет, учитывая имеющуюся у вас сумму наличными. Безусловно, если у вас достаточно денег, выбор меню на обед не составит для вас особых трудностей. Но такое допущение нельзя считать надежным, поскольку сейчас люди редко носят с собой наличные, предполагая расплачиваться дебитной или кредитной карточкой либо с помощью смартфона.

Как же тогда выбрать блюда из меню, чтобы заказать обед? С этой целью вы могли бы начать с того, чтобы предложить каждому участнику обеда самому сделать заказ предпочитаемых блюд. Если такой заказ позволит вам уложиться в бюджет, то задачу можно считать решенной. Но что если общая сумма заказа превысит сумму наличных? В таком случае придется предложить участникам обеда заказать более дешевые блюда или же отказаться от некоторых закусок или напитков, чтобы общая сумма заказа оказалась в доступных пределах. Действенность такого подхода зависит от ваших способностей оценить общую ценность каждого заказа на обед, исходя из предпочтений всех его участников. Здесь под *ценностью* имеется в виду общая удовлетворенность обедом с учетом индивидуального выбора блюд.

Это непростая задача, но допустим, что мы в состоянии определить ценность заказов на обед. И тогда цель состоит в том, чтобы найти такой заказ, который имел бы максимальную ценность, и при этом его общая стоимость не превысила бы допустимый предел наличных денег. Для достижения этой цели, вероятно, подойдет стратегия постепенного обмена ценности на стоимость. Но такая стратегия не так уж проста, как кажется на первый взгляд. Даже если лучше вашему коллеге Бобу отказаться от заказа напитка, чем коллеге Алисе отказаться от

закуски, все равно не ясно, приведет ли такой выбор к большей ценности обеда в целом. Ведь дело в том, что выбранная Алисой закуска может оказаться более дорогой, чем выбранный Бобом напиток, а значит, отказ от нее поможет сэкономить столько, что ваша коллега Кэрол сможет заказать другое второе блюдо и, таким образом, удовлетворение общих предпочтений Боба и Кэрол может оказаться более ценным, чем индивидуальные предпочтения Алисы. Когда дело доходит до таких подробностей, очень трудно сказать, какие варианты выбора блюд следует изменить и что заказать.

Задача выбора приобретаемых предметов при ограниченном бюджете встает перед нами и во многих других ситуациях, включая планирование отпуска с более или менее дорогими мероприятиями, выбор маршрута путешествия, модели автомашины или электронных гаджетов. И как ни странно, это в общем случае весьма трудная задача. Все известные алгоритмы решения подобных задач имеют время выполнения, экспоненциально зависящее от количества выбираемых предметов. Рассмотрим в качестве примера составление всех возможных вариантов выбора блюд на обед, записываемых на небольшой салфетке размером 10×10 см, на которой может уместиться запись десяти выбранных блюд. Если допустить, что каждый участник обеда выберет из меню от одного до четырех блюд из десяти возможных (что дает 5 860 вариантов меню для одного человека), то салфеток, требующихся для записи всех возможных вариантов обеда на пять персон, окажется достаточно, чтобы покрыть ими поверхность Земли более 13 раз.

Для задачи выбора блюд на обед характерны следующие две особенности: 1) быстрый рост количества возможных вариантов, которые следует принять во внимание, и 2) известные алгоритмы работают лишь в том случае, когда в них проверяются все или почти все возможные варианты.

Подобные задачи называются *трудноразрешимыми* (intractable), потому что решающие их алгоритмы выполняются слишком долго, чтобы применять их на практике, кроме самых простых случаев.

Но это обстоятельство не мешает нам заказывать обеды, планировать отпуск и делать иной выбор, нередко приносящий нам удовлетворение. При этом мы употребляем *приближенные алгоритмы*, представляющие собой эффективные алгоритмы, которые вычисляют не точное, а достаточно близкое к оптимальному приближенное решение задачи. Например, самым простым приближением для решения задачи выбора блюд на обед является разделение общего бюджета на всех участников обеда, чтобы каждый из них смог выбрать себе блюда, уложившись в свою долю бюджета.

Трудностями поиска оптимальных вариантов выбора можно выгодно воспользоваться. Страховые компании могут предлагать выбор из чрезмерно большого количества вариантов страховок, что затрудняет клиентам выбор оптимального варианта, зато сам*ой* компании дает возможность заработать больше денег.

7 Трудноразрешимые задачи

Алгоритмы, обсуждавшиеся в предыдущих главах, демонстрировали разные показатели времени выполнения. Например, поиск наименьшего элемента в неотсортированном списке занимает линейное время, тогда как в отсортированном списке — константное время. Аналогично поиск конкретного элемента в неотсортированном списке отнимает линейное время, тогда как в отсортированном массиве или сбалансированном бинарном дереве поиска — логарифмическое время. В обоих случаях отличие заключается в предварительно вычисленной структуре данных. Но разные алгоритмы демонстрируют разное время выполнения при одних и тех же входных данных. Например, сортировка выбором является квадратичным алгоритмом, тогда как сортировка слиянием — линейно-логарифмическим алгоритмом.

Алгоритм с квадратичным временем выполнения может оказаться слишком медленным, чтобы оказаться практически полезным. Рассмотрим в качестве примера задачу сортировки всех 300 миллионов граждан США по их именам. Если попробовать решить эту задачу на компьютере, способном выполнять миллиард операций в секунду, то на сортировку выбором потребуется порядка 90 миллионов секунд (около двух лет и десяти месяцев), что совершенно непрактично. А для решения той же задачи с помощью линейно-логарифмической сортировки слиянием требуется меньше 10 секунд. Но если обрабатывать входные данные умеренного объема, то, возможно, и не стоит особенно беспокоиться о временн*о́й* сложности применяемого алгоритма, особенно учитывая то обстоятельство, что быстродействие компьютеров с каждым годом растет.

Рассмотрим в качестве аналогии возможные решения разных транспортных задач. Например, чтобы отправиться на работу, вы можете поехать на велосипеде, автобусе или на автомашине. Но для пересечения Атлантического океана ни одно из этих решений не годится, и поэтому вам придется сесть на круизное судно или в самолет. В принципе, вы можете пересечь Атлантический океан и на байдарке, но для этого вам придется потратить столько времени (и прочих ресурсов), что решить данную задачу таким способом практически невозможно.

Аналогично существуют вычислительные задачи, в принципе имеющие решения, но на практике их вычисления отнимут слишком много времени. В этой главе представлены примеры такого рода задач и, в частности, ряд задач, которые могут быть решены (по крайней мере, на сегодняшний день) только алгоритмами с *экспоненциальным* временем выполнения, т.е. такими алгоритмами, время выполнения которых растет по экспоненциальному закону по мере увеличения объема входных данных. Такие алгоритмы непригодны с практической точки зрения, за исключением обработки небольших объемов данных, что делает особенно актуальным вопрос нижних границ и существования более быстродействующих алгоритмов. Этот вопрос положен в основу проблемы P=NP, призванной решить вопрос о равенстве классов сложности P и NP, который хорошо известен в информатике и до сих пор остается открытым.

Как и при сортировке, ограничения, полученные информатикой, на первый взгляд, разочаровывают, но на самом деле они не должны вызывать такую реакцию. Даже если для решения конкретной задачи нельзя разработать эффективные алгоритмы, это совсем не означает, что мы должны вообще отступать перед такими задачами. В частности, можно придумать *приближенные алгоритмы*, вычисляющие не точные, но с практической точки зрения в достаточной степени пригодные решения подобных задач. А тем обстоятельством, что конкретную задачу нельзя решить на практике, можно иногда выгодно воспользоваться для решения других задач.

Несмотря на то что крупные входные данные обнаруживают отличие квадратичного алгоритма от линейно-логарифмического, квадратичный алгоритм может вполне подойти для обработки небольшого объема данных. Например, для сортировки списка из 10 000 элементов выбором потребуется около одной десятой секунды на компьютере, выполняющем миллиард операций в секунду. И хотя время выполнения сортировки в данном случае едва заметно, достаточно увеличить длину данного списка в десять раз, чтобы это время возросло в сто раз. Следовательно, время выполнения сортировки списка из 100 000 элементов возрастет до 10 секунд. Такой результат может оказаться неприемлемым, в особенности тогда, когда пользователи взаимодействуют с системой, ожидая ее немедленной реакции. Тем не менее следующее поколение компьютеров, вероятно, будет способно исправить это положение, и алгоритм снова может пригодиться.

Технические достижения способны расширить границы применения квадратичного алгоритма. Но не стоит надеяться, что это сделает пригодным алгоритм с экспоненциальным временем выполнения.

Перевешивание чаши весов

В начале истории из фильма *В поисках утраченного ковчега* Индиана Джонс исследует пещеру в поисках ценного золотого идола — цели его экспедиции. Этот идол покоится на чаше весов, предназначенной для отпирания целого ряда смертельных ловушек, если снять с нее идола. Чтобы обойти этот защитный механизм, Индиана Джонс сменяет идола на мешок песка, вес которого, как он надеется, приблизительно такой же, как и вес идола. Но, увы, мешок оказывает слишком тяжелым и отпирает ловушки, ведущие к впечатляющему выходу из смертельно опасной пещеры.

Если бы Индиана Джонс знал точный вес идола, он мог бы наполнить мешок нужным количеством песка, и тогда его выход из пещеры был бы не таким драматичным. Но поскольку у него под рукой не было весов, ему пришлось отмерить вес каким-то другим способом. Правда, построить рычажные весы не составляет большого труда. Для этого, по существу, достаточно взять палку, нацепить на один ее конец мешок, а на другой — точный вес, наполняя затем мешок песком до тех пор, пока этот самодельный рычаг не придет в равновесие. Допуская, что у Индианы Джонса нет под рукой объекта точно такого же веса, как и у идола, ему придется определить вес приблизительно, опробовав целый ряд объектов. На первый взгляд, это не такая уж и сложная задача. Так, если идол весит, скажем, 42 унции, что составляет около 2,5 фунтов или 1,2 кг, а у Индианы Джонса имеется шесть объектов весом 5, 8, 9, 11, 13 и 15 унций, то, опробовав их в разных сочетаниях, он может в итоге выяснить, что объекты весом 5, 9, 13 и 15 унций совместно будут весить 42 унции.

Но насколько точным оказывается такой метод опробования объектов в разных сочетаниях и сколько времени это отнимет? В данном примере вес самого тяжелого объекта оказывается меньше половины веса идола, а это означает, что потребуется по меньшей мере три объекта, чтобы достичь веса идола. Но ведь не сразу удается выяснить, какие именно три объекта следует выбрать, и потребуются ли для этого еще четвертый и пятый объекты. Более того, алгоритм должен действовать в любых возможных ситуациях, а следовательно, он должен быть в состоянии обрабатывать в качестве входных данных разное количество объектов с разным весом, а также достигать произвольного целевого веса.

Прямой метод решения задачи взвешивания состоит в том, чтобы систематически составлять объекты во всех возможных сочетаниях и проверять каждое их

сочетание на равенство их общего веса целевому. Такая стратегия иначе называется *генерацией с проверкой* (generate-and-test), поскольку состоит в повторяющемся выполнения следующих действий: 1) *сгенерировать* возможное решение и 2) *проверить*, является ли возможное решение фактическим.

В данном случае на стадии генерации создается сочетание объектов, а на стадии проверки веса объектов складываются, и суммарный их вес сравнивается с целевым. Следует особо подчеркнуть, что стадия генерации систематически повторяется и охватывает все возможные случаи, ведь иначе в таком алгоритме может быть упущено искомое решение.

Время выполнения алгоритма порождения и проверки зависит от количества сочетаний объектов, которые могут быть составлены. Чтобы выяснить, сколько существует таких сочетаний, рассмотрим, каким образом из всех имеющихся объектов создается любое отдельное сочетание. С этой целью возьмем произвольное сочетание чисел, например 5, 9 и 11. Для каждого имеющегося объекта мы можем выяснить, содержится ли он в этом конкретном сочетании. Так, число 5 в таком сочетании присутствует, а число 8 в нем отсутствует. Числа 9 и 11 находятся в данном сочетании, а числа 13 и 15 — нет. Иными словами, любое возможное сочетание задается решением о включении в него (или исключении из него) каждого элемента, причем все эти решения принимаются независимо одно от другого.

Процесс составления сочетания можно сравнить с заполнением опросного листа, состоящего из доступных объектов и расположенных рядом с ними клетками для отметки "галочкой". Одно сочетание соответствует опросному листу, в котором "галочкой" отмечены некоторые клетки рядом с выбранными объектами. Следовательно, число возможных сочетаний соответствует числу разных способов заполнения опросного листа. И здесь мы видим, что каждая клетка может быть либо отмечена "галочкой", либо нет — независимо от всех остальных клеток.

Таким образом, число возможных сочетаний задается как произведение имеющихся вариантов выбора, которых ровно 2: объект либо выбран, либо не выбран (это то же самое, что и отметить или не отметить клетку "галочкой"). У Индианы Джонса имеется на выбор шесть объектов, поэтому число возможных сочетаний равно двум в шестой степени: $2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^6 = 64$ [1]. Поскольку согласно алгоритму генерации и проверки каждое сочетание должно быть проверено, время его выполнения возрастает по меньшей мере с той же скоростью. В действительности выполнение данного алгоритма отнимает еще больше времени, поскольку для проверки одного сочетания следует сложить все веса и сравнить полученный суммарный вес с целевым.

Когда время выполнения взрывается

Несмотря на то что 64 сочетания — по-видимому, не так уж и плохо, важная особенность времени выполнения алгоритма состоит в том, что оно очень быстро растет по мере увеличения объема входных данных. Как пояснялось в главе 2, “От слов к делу: когда действительно происходит вычисление”, время выполнения алгоритмов измеряется как скорость роста, а не как абсолютная величина, поскольку это более общая его характеристика, не зависящая от конкретных задач и характеристик производительности компьютеров.

Последнее обстоятельство иногда служит оправданием для применения алгоритмов с плохим временем выполнения. При этом высказывается следующий аргумент: “Да, я знаю, что для завершения алгоритма требуется несколько минут, но давайте подождем до тех пор, пока мы не приобретем более быстродействующий компьютер. Это разрешит данный вопрос”. Ведь закон Мура гласит, что быстродействие компьютеров удваивается приблизительно через каждые 18 месяцев[1] [2].

Если быстродействие компьютера удваивается, то квадратичный алгоритм позволяет за то же время обработать входные данные, большие приблизительно в 1,4 раза [3]. А теперь рассмотрим, что в таком случае произойдет со временем выполнения экспоненциального алгоритма. Насколько можно увеличить входные данные, чтобы время выполнения данного алгоритма увеличилось не больше чем в два раза? Время выполнения данного алгоритма удваивается, если объем входных данных увеличивается на единицу. Это означает, что данный алгоритм при удвоении быстродействия компьютера способен справиться лишь с одним дополнительным элементом данных. Иными словами, чтобы обработать входные данные, увеличенные всего лишь на один элемент, быстродействие компьютера необходимо удвоить.

Время выполнения алгоритма удваивается настолько быстро, что никаких технологических усовершенствований, повышающих мощность компьютеров в некоторое фиксированное число раз, просто недостаточно для того, чтобы экспоненциальный алгоритм мог справиться со значительно более крупными данными. Не меняет дела повышение быстродействия даже в десяток раз. И если такое повышение позволит квадратичному алгоритму обработать в три раза больше входных данных, то экспоненциальному алгоритму удастся обработать входных данных лишь *на* три элемента больше, поскольку время его выполнения при этом возрастет в $2 \times 2 \times 2 = 2^3 = 8$ раз.

В табл. 7.1 наглядно показан огромный разрыв в возможностях алгоритмов с неэкспоненциальным и экспоненциальным временем выполнения в отношении обработки разных объемов входных данных. Время выполнения

---

[1] В настоящее время закон Мура постепенно перестает описывать реальную ситуацию, и производительность компьютеров в основном повышается за счет многоядерности, а это уже другая тема. — *Примеч. ред.*

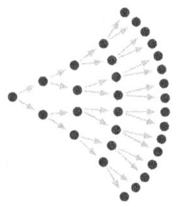
неэкспоненциальных алгоритмов начинает заметно расти только тогда, когда во входных данных оказывается больше 1000 элементов, тогда как экспоненциальные алгоритмы довольно сносно обрабатывают входные данные, состоящие не больше чем из 20 элементов. А если входные данные состоят из 100 элементов, то для выполнения экспоненциальных алгоритмов потребуется больше 400 миллиардов столетий, т.е. в 2900 больше возраста нашей Вселенной!

Таблица 7.1. Приблизительные показатели времени работы алгоритмов для разных объемов входных данных на компьютере с миллиардом операций в секунду

Объем входных данных	*Время выполнения*			
	Линейное	*Линарифмическое*	*Квадратичное*	*Экспоненциальное*
20				0,001 с
50				13 дней
100				
1000				
10000			0,1 с	
1 миллион	0,001 с	0,002 с	16 мин	
1 миллиард	1 с	30 с	32 г	

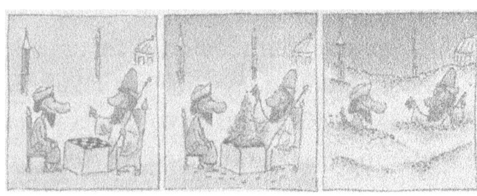
Примечание. Пустые ячейки таблицы обозначают время выполнения меньше 1 мс, которое слишком мало для восприятия человеком, а следовательно, не имеет никакого практического значения. А серые ячейки таблицы обозначают время выполнения, которое слишком велико, чтобы его постичь.

Ошеломительный результат роста экспоненциального времени выполнения алгоритмов можно сравнить со взрывом ядерной бомбы, возникающим в результате высвобождения ничтожно малого количества энергии при расщеплении атомов [4]. А опустошительные разрушения от такого взрыва объясняются тем, что бóльшая часть деления атомов происходит в течение короткого периода времени. При этом расщепление каждого атома влечет за собой два (и более) делений в течение короткого времени, что в конечном счете приводит к экспоненциальному росту делений и высвобождению соответствующего количества энергии.



Рассмотрим в качестве другого примера легенду о крестьянине, который изобрел игру в шахматы. Эта игра настолько понравилась царю, что он готов был исполнить любое желание крестьянина. А тот попросил положить одно зернышко риса на первую клетку шахматной доски, два зернышка — на вторую клетку, четыре — на третью клетку и так далее до тех пор, пока не будут покрыты все клетки шахматной доски. Не зная особенности роста по экспоненциальному закону, царь

посчитал, что исполнить желание крестьянина будет нетрудно, и пообещал это сделать. И разумеется, он не сумел сдержать свое обещание, поскольку для покрытия всей шахматной доски потребовалось бы 18 квинтиллионов зернышек риса, что в 500 раз[2] превышает объем производства риса во всем мире за 2014 год.



Огромный разрыв в объеме входных данных, которые способны обработать экспоненциальные и неэкспоненциальные алгоритмы, объясняет отличие практичных алгоритмов со временем выполнения, меньшим экспоненциального, от непрактичных алгоритмов с экспоненциальным (или еще худшим) временем выполнения. В частности, алгоритмы с экспоненциальным временем выполнения нельзя считать практичными для решения задач, поскольку для вычисления результатов по этим алгоритмам требуется слишком много времени, кроме тех случаев, когда объем входных данных очень невелик.

Общая судьба

Алгоритм генерации и проверки годится для решения задачи взвешивания только при относительно малом объеме входных данных (меньше 30 элементов или около того), и это как раз то, что нужно для решения конкретной задачи, стоящей перед Индианой Джонсом. Но вследствие экспоненциального роста времени выполнения этот алгоритм совершенно непригоден для обработки входных данных объемом 100 или больше элементов. Однако эта конкретная особенность роста времени выполнения алгоритма по экспоненциальному закону совсем не означает, что не может быть других, более эффективных алгоритмов с неэкспоненциальным временем выполнения, хотя на сегодняшний день такие алгоритмы, увы, неизвестны.

Задача, которую можно решить только с помощью алгоритма с экспоненциальным (или худшим) временем выполнения, называется *трудноразрешимой*. В частности, задача взвешивания может показаться трудноразрешимой, поскольку ее можно решить только с помощью алгоритма с экспоненциальным временем выполнения, хотя мы и не знаем этого абсолютно точно. Возможно, имеются неэкспоненциальные алгоритмы, которые еще не открыты. Если бы мы сумели доказать, что таких алгоритмов для решения данной задачи не существует, то ее можно было бы считать трудноразрешимой. И уверенность в этом могла бы дать нам нижняя граница.

[2] В Википедии приводится иное значение — в 1800 раз. — *Примеч. ред.*

Но почему это должно нас заботить? Может быть, лучше оставить решение этой задачи специалистам по информатике и перейти к исследованию других вопросов? Дело в том, что задача взвешивания схожа с целым рядом других задач с двумя следующими свойствами. Во-первых, для решения задачи взвешивания известны лишь алгоритмы с экспоненциальным временем выполнения. И во-вторых, если бы был найден неэкспоненциальный алгоритм для решения одной из этих задач, то немедленно были бы получены аналогичные алгоритмы для решения и всех остальных задач. Любая задача из этого ряда называется *NP-полной* [5]. Особое значение подобных задач объясняется тем, что многие практические задачи на самом деле являются NP-полными и разделяют общую судьбу потенциально трудноразрешимых задач.

К концу приключения Индианы Джонса в фильме *Индиана Джонс и королевство хрустального черепа* его спутник Мак собирает драгоценные предметы в храме хрустального черепа. Он пытается унести столько драгоценных предметов, сколько сможет, стараясь увеличить их количество до максимума. Эта задача, а также задачи взвешивания и коллективного обеда служат примерами задачи о наполнении рюкзака ограниченной вместимости как можно б*о*льшим количеством предметов с целью увеличить до предела ценность или полезность упакованных в рюкзак предметов. В задаче взвешивания в качестве рюкзака служат весы, ограничение на них накладывает измеряемый вес, количество пакуемых предметов определяется объектами, умещающимися на весах, а цель оптимизации состоит в том, чтобы измерить вес, как можно точнее соответствующий целевому весу. В задаче выбора, решаемой Маком, ограничением служит то, что он в состоянии унести, упаковка означает выбор драгоценных предметов, а оптимизация — общую их ценность. А в задаче коллективного обеда ограничением служит общая сумма наличными для заказа всех блюд, упаковка означает выбор блюд, тогда как оптимизация — увеличение до максимума ценности выбора блюд из меню на обед.

Между прочим, они оба — и Мак, и Индиана Джонс — терпят неудачу, решая свои задачи: Мак тратит слишком много времени на выбор драгоценных предметов и погибает в разрушающемся храме, а мешок с песком Индианы Джонса отпирает смертельные ловушки, хотя ему в конечном итоге удается их избежать. И хотя не ясно, объясняется ли их неудача NP-полнотой тех задач, которые они пытаются решить, она служит неплохим напоминанием об их трудноразрешимости.

Задача о рюкзаке вместе с ее приложениями является лишь одной из многих NP-полных задач. Другим известным примером служит задач коммивояжера, которому требуется объехать ряд городов, сократив до минимума длину пути. В качестве простого решения этой задачи можно сформировать все возможные замкнутые маршруты и выбрать кратчайший из них. Сложность такого алгоритма также заключается в его экспоненциальном характере, который к тому же

оказывается намного хуже, чем в задаче взвешивания. Так, если с помощью алгоритма взвешивания решение исходной задачи с 20 элементами можно завершить за 1 мс, то для вычисления кругового маршрута по 20 городам потребуется 77 лет. Отыскание замкнутых маршрутов находит полезное применение не только среди коммивояжеров, но и во многих других приложениях: от планирования маршрутов школьных автобусов до разработки маршрутов круизных судов. Многие задачи оптимизации также являются NP-полными.

Примечательная особенность NP-полных задач состоит в том, что все они либо одновременно оказываются трудноразрешимыми, либо все решаются с помощью алгоритмов с неэкспоненциальным временем выполнения. Чтобы показать, что задача является NP-полной, достаточно продемонстрировать, что ее решение допускает преобразование (в течение неэкспоненциального времени) в решение другой задачи, уже известной как NP-полная. Такое преобразование решения называется *приведением*, или *сведением*. Приведение является искусным способом решения одной задачи путем его преобразования в решение другой задачи. Приведение возникает очень часто и порой неявно, как, например, в том случае, когда Индиана Джонс сводит задачу поиска надежных плиток к написанию по буквам имени *Iehova* или же когда Шерлок Холмс сводит задачу хранения сведений о подозреваемых к операциям, выполняемым в словаре.

Приведение само по себе является вычислением и может служить удобным дополнением к арсеналу вычислительных методов, поскольку операции приведения могут быть объединены с другими видами вычислений. Например, задача поиска наименьшего элемента в списке может быть вначале сведена к сортировке списка, а затем — к извлечению первого элемента из отсортированного списка. В частности, приведение преобразует входные данные (в данном случае — неотсортированный список) в конкретную форму (в данном случае — отсортированный список) для обработки с помощью другого алгоритма (в данном случае — извлечения первого элемента). Объединение обоих видов вычислений, сортировки и последующего извлечения первого элемента, приводит в конечном счете к вычислению для решения исходной задачи поиска наименьшего элемента в произвольном массиве. Тем не менее очень важно принять во внимание время выполнения операций приведения. Например, для выполнения шага сортировки требуется линейно-логарифмическое время, и поэтому время выполнения алгоритма, получаемого путем такого приведения, не совпадает только с линейным временем простого просмотра списка на наличие наименьшего элемента. Следовательно, такое приведение не приносит особой пользы, поскольку приводит к менее эффективному алгоритму [6]. Таким образом, приведение NP-полных задач должно быть достигнуто посредством неэкспоненциальных алгоритмов, ведь иначе неэкспоненциальное решение одной задачи стало бы экспоненциальным из-за экспоненциального времени выполнения подобного приведения.

Неэкспоненциальное приведение всех NP-полных задач дает немалый выигрыш. Подобно тому, как колесо было изобретено лишь одним человеком, но может использоваться всеми людьми, решение одной NP-полной задачи решает все остальные задачи. Именно это обстоятельство и приводит к вопросу, который сводится к следующему известному уравнению:

P=NP ?

Этот вопрос был впервые поднят австрийско-американским логиком Куртом Гёделем (Kurt Gödel) и уточнен в 1971 году канадско-американским ученым в области вычислительных систем Стивеном Куком (Stephen Cook). Данный вопрос состоит в следующем: равнозначен ли класс задач P, которые могут быть *решены* в течение неэкспоненциального (полиноминального) времени, классу задач NP, которые могут быть *проверены* в течение полиноминального времени. Обнаружение экспоненциальной нижней границы для NP-полных задач даст отрицательный ответ на рассматриваемый здесь вопрос-уравнение. С другой стороны, обнаружение неэкспоненциального алгоритма для решения любой из этих задач даст утвердительный ответ на данный вопрос.

Разрешимость огромного числа практических задач зависит от ответа на вопрос-уравнение P=NP, что лишь подчеркивает его особую важность. Этот вопрос волновал ученых в течение четырех десятилетий и до сих пор остается одним из самых важных открытых вопросов в области информатики. Большинство современных ученых полагают, что NP-полные задачи действительно трудноразрешимы и что для их решения отсутствуют подходящие неэкспоненцильные алгоритмы, хотя никто из них не знает этого наверняка.

Обманчивость нирваны

Непомерно длинное время выполнения алгоритмов может, конечно, вызвать полное разочарование. Ведь найденное решение задачи может обойтись слишком дорого, чтобы достичь его на практике, подобно роскошным плодам, висящим на дереве перед Танталом из древнегреческого мифа, до которых ему никогда не дотянуться. Тем не менее разочаровывающие сведения о NP-полных задачах не должны служить причиной для отчаяния. Помимо методов борьбы с неэффективностью алгоритмов, имеется и удивительная возможность борьбы с ограничениями.

Если поиски подходящего решения задачи отнимают слишком много времени, то можно поднять руки вверх и сдаться, а можно попытаться извлечь наибольшую выгоду из сложившегося положения и попробовать найти приблизительное решение, которое может быть неточным, но вполне приемлемым. Так, если вы работаете, чтобы заработать себе на жизнь, то, вероятнее всего, откладываете

часть своих доходов, чтобы иметь достаточно средств для проживания на пенсии. На первый взгляд, это неплохой план. Бесспорно, еще лучше было бы выиграть в лотерею и тотчас подать в отставку. Но такой план редко воплощается в жизнь, а следовательно, он не может служить практической альтернативой уже имеющемуся хорошему плану откладывать часть средств на старость.

Экспоненциальный алгоритм, вычисляющий точное решение задачи, оказывается столь же непрактичным для обработки крупных объемов данных, как и план выиграть в лотерею для выхода на пенсию. Следовательно, в качестве альтернативы можно попытаться найти эффективный неэкспоненциальный алгоритм, хотя и не всегда способный вычислить идеальное решение, но дающий приблизительные результаты, вполне приемлемые для практических целей, т.е. приближенный алгоритм. Так, откладывание части заработанных средств на старость можно считать приближением выигрыша в лотерею, хотя это и не самое лучшее приближение.

Некоторые приближенные алгоритмы вычисляют результаты, которые гарантированно находятся в пределах некоторого множителя точности от идеального решения. Например, Индиана Джонс может найти приблизительное решение своей задачи взвешивания в течение линейно-логарифмического времени, добавляя объекты в порядке уменьшения их веса. Такой алгоритм дает решение, которое гарантированно хуже оптимального не более чем на 50%. Подобное решение кажется не слишком многообещающим из-за потенциально низкой точности, но во многих случаях оно оказывается намного ближе к оптимальному решению и при этом достигается небольшой ценой.

В задаче взвешивания простой алгоритм первоначально состоит в том, чтобы отсортировать объекты по весу, а затем найти первый объект, который весит меньше, чем целевой объект, и продолжать добавлять объекты до тех пор, пока не будет достигнут целевой вес. Так, если сложить сначала веса 15, 13 и 11, то в итоге получится суммарный вес 39. А если добавить затем вес 9, то целевой вес будет превышен, а значит, этот вес следует пропустить и опробовать другой вес. Но оба веса, 8 и 5, также чрезмерны, и поэтому приблизительный результат достигается сложением весов 5, 13 и 11 и лишь на 3 единицы веса отличается от оптимального решения. Иными словами, найденное приближенное решение соответствует оптимальному с точностью до 7%.

Такая стратегия неоднократного выбора наибольшего из возможных значений называется *жадным алгоритмом*, поскольку он хватается за первую же возможность. Жадные алгоритмы просты и вполне пригодны для решения многих задач, но для некоторых задач им недостает точных решений, как в данном случае. Именно жадное стремление выбрать вес 11 вместо того, чтобы дождаться очереди веса 9, сделало недоступным достижение оптимального решения в данном примере. Выполнение этого жадного алгоритма отнимает линейно-логарифмическое время, обусловленное первоначальным временем сортировки, а следовательно, он достаточно эффективен.

В связи с приближенностью алгоритма возникает следующий важный вопрос: насколько точного приближения удастся достичь по такому алгоритму в худшем случае? На примере жадного алгоритма взвешивания можно показать, что он всегда соответствует оптимальному решению с точностью до 50% [7].

Приближения являются достаточно хорошими решениями задач. Но они не настолько хороши, как оптимальные решения (хотя и лучше, чем отсутствие всякого решения вообще). В истории из фильма *Индиана Джонс и храм судьбы* главный герой, Индиана Джонс, и его спутники оказываются в затруднительном положении, когда их самолету грозит участь разбиться о гору. Оба пилота покинули самолет, слив оставшееся топливо и не оставив на борту никаких парашютов. Индиана Джонс выходит из столь трудного положения, воспользовавшись надувной лодкой как неким подобием парашюта, чтобы посадить пассажиров на землю. Зачастую лучше иметь хоть какой-нибудь приближенный алгоритм, каким бы неточным он ни был, чем вообще не иметь никакого практичного алгоритма.

Отходы — в доходы!

Приближенные алгоритмы способствуют повышению эффективности решений задач, которые оказываются неэффективными из-за неэкспоненциальных алгоритмов. Это хорошие новости, но еще не все. То, что решения некоторой конкретной задачи не могут быть вычислены эффективно, на самом деле может быть преимуществом. Например, алгоритм генерации и проверки для решения задачи взвешивания подобен тому, что пришлось бы сделать при попытке вскрыть кодовый замок, для которого неизвестна отпирающая его комбинация. Ведь для этого пришлось бы опробовать все комбинации, которых в трехразрядном кодовом замке имеется 10×10×10 = 1000. На испытание всех 1000 комбинаций потребуется немало времени, чем и объясняется относительная эффективность кодовых замков в защите доступа к сумкам, чемоданам и камерам хранения багажа. Безусловно, кодовые замки можно сломать, но это, скорее, решение задачи в обход, а не прямое.

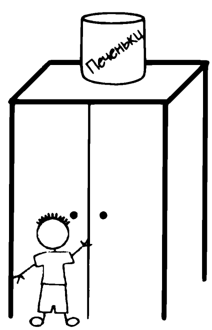
Перепробовать 1000 комбинаций на электронной вычислительной машине не составит особого труда, но медлительному человеку для этого потребуется немало времени, а это наглядно показывает, что эффективность — понятие относительное и зависит от технических возможностей компьютера. Но поскольку рост времени выполнения алгоритмов находится в прямой зависимости от увеличения объема входных данных, повышенное быстродействие компьютеров не способно возместить заметный рост времени выполнения экспоненциальных алгоритмов при незначительном увеличении объема входных данных. Это обстоятельство выгодно используется в криптографии для упрощения безопасного обмена сообщениями. Всякий раз, когда вы посещаете веб-сайт по адресу, начинающемуся с

префикса `https://`, в строке адреса вашего веб-браузера появляется пиктограмма закрытого замка, чтобы указать, что с этим веб-сайтом установлен безопасный канал обмена данными.

Один из способов отправки и приема зашифрованных сообщений действует по принципу шифрования и дешифрования сообщений по двум связанным вместе *ключам*: *открытому* и *секретному*. У каждого участника обмена данными имеется один из этой пары ключей. Открытый ключ известен всем, но секретный ключ каждого участника обмена данными известен только ему самому. Оба ключа совместно действуют таким образом, чтобы сообщение, зашифрованное открытым ключом, можно было расшифровать только соответствующим секретным ключом. Благодаря этому становится возможной отправка сообщения какому-нибудь адресату, зашифровав его общеизвестным открытым ключом данного лица. А поскольку получателю сообщения известен секретный ключ, то лишь он и сможет расшифровать данное сообщение. Так, если вам требуется проверить текущий остаток на своем банковском счете через Интернет, ваш веб-браузер передает ваш открытый ключ банковскому компьютеру, где он применяется для шифрования суммы на вашем счете, отправляемой обратно вашему веб-браузеру. Все, кто увидят это сообщение, не смогут его расшифровать, поскольку оно надежно зашифровано. Только вы сможете его расшифровать в своем веб-браузере, воспользовавшись собственным секретным ключом.

В отсутствие Интернета для такого же обмена нам пришлось бы, прибегнув к услугам обычной почты, отправить в банк ящик с незапертым висячим замком, к которому у вас имеется ключ. Получив ваш ящик, в банке напишут на листке бумаги сумму остатка на вашем счете, положат этот лист в ваш ящик, запрут его на приложенный висячий замок и отправят вам обратно. Получив ящик, вы откроете висячий замок своим ключом и увидите в нем сумму остатка на вашем счете. А поскольку никто другой не сумеет открыть замок, то хранящиеся в ящике сведения будут надежно защищены от несанкционированного доступа во время пересылки. В данном примере ящик с открытым замком соответствует открытому ключу, а помещение в него в банке листа бумаги с суммой остатка на счете и запирание замка — шифрованию сообщения.

Зашифрованное сообщение практически защищено от несанкционированного доступа, поскольку для его расшифровки не знающему секретный ключ придется вычислить простые множители очень большого числа. И хотя неизвестно, является ли вычисление простых множителей NP-полной задачей, в настоящее время неэкспоненциальные алгоритмы для решения этой задачи неизвестны, а значит, практическое ее решение отнимет слишком много времени. Таким образом, трудности решения отдельных задач можно выгодно использовать для защиты обмена данными от несанкционированного доступа.



Этот принцип не нов. Рвы, ограды, стены и все прочие защитные механизмы основаны именно на данном принципе. Но многие из этих средств защиты могут быть разрушены. В настоящее время отправка и прием зашифрованных сообщений может считаться безопасной из-за отсутствия неэкспоненциального алгоритма для разложения на простые множители. Но если только кто-нибудь обнаружит такой алгоритм, то вся безопасность обмена сообщениями мгновенно испарится. А вот если бы кому-нибудь удалось установить экспоненциальную нижнюю границу этой задачи, мы могли бы быть уверены в безопасности зашифрованного сообщения. Пока что рассмотренный здесь способ надежного обмена сообщениями продолжает исправно действовать из-за нашего невежества в данном вопросе.

Дальнейшее исследование

Приключения Индианы Джонса обычно связаны с поисками артефактов, местами и людьми. Иногда на поиски наводят предметы или сведения, собранные по ходу дела. В подобных случаях проходимый путь определяется динамически. В таких историях поиск приводит к списку посещаемых мест, а этот список служит путеводителем к заветной цели. Данному образцу следуют и такие фильмы, как *Сокровище нации* или *Код да Винчи.*

Порой основой для поиска служит карта спрятанных сокровищ. В таком случае путь ясен с самого начала, а поиск состоит в обнаружении вех, обозначенных на карте. Карты спрятанных сокровищ приобрели известность благодаря книге *Остров сокровищ* Роберта Льюиса Стивенсона (Robert Louis Stevenson). Такая карта служит алгоритмом для поиска конкретного места, но алгоритм может быть задан самыми разными способами. В некоторых историях карта спрятанных сокровищ содержит не прямые инструкции для поиска сокровищ, а скорее ключи к разгадке, коды или головоломки, которые необходимо расшифровать или разрешить. Такие карты на самом деле описывают не алгоритмы, а скорее задачи, которые приходится решать. Именно так происходит, например, в фильме *Сокровище нации,* в котором сама карта спрятанных сокровищ скрыта на обратной стороне Декларации независимости, где находится код для поиска очков, обнаруживающих другие ключи к разгадке на карте.

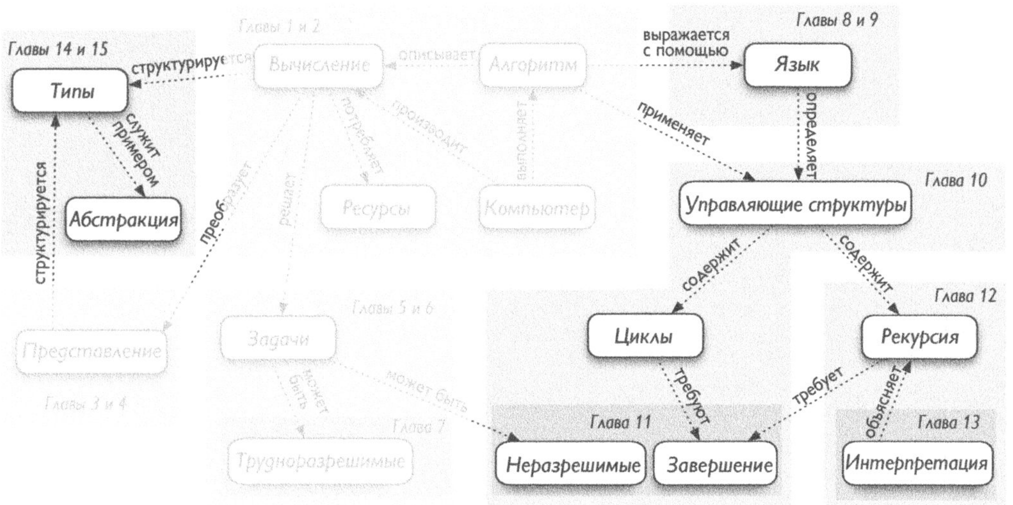
В фильме *Сокровище нации* содержится также головоломка, подобная трудной задаче пересечения плиточного пола, которую пришлось решать Индиане Джонсу. Одному из героев этого фильма, Бену Гейтсу (Ben Gates), приходится разгадывать набранный пароль по клавишам, нажатым на клавиатуре, для чего требуется найти правильный порядок следования букв. Слово или фраза, буквы которой составляют другую фразу, называется *анаграммой.* Решение анаграмм на самом деле имеет отношение не к сортировке, а к поиску конкретного порядка следования букв среди всех возможных перестановок. Например, анаграммы появляются в историях Гарри Поттера, в которых фраза “I am Lord Voldemort” (Я лорд Волдеморт) служит анаграммой для имени “Tom Marvolo Riddle”, а в фильме *Тихушники* кодовое название дешифратора “Setec Astronomy” служит анаграммой для фразы “Too many secrets” (Слишком много секретов).

В немецкой версии *Aschenputtel* сказки "Золушка", предложенной братьями Гримм, вредная мачеха дает Золушке трудное задание — отделить чечевицу от пепла, — что является простой формой блочной сортировки. Иногда события в истории также требуют сортировки, когда они повествуются в нехронологическом порядке. Одним из крайних тому примеров служит упоминавшийся ранее фильм *Помни*, в котором б*о*льшая часть сюжета раскрывается в обратном порядке. Иногда нечто подобное происходит и в фильме *Вечное сияние чистого разума*, в котором пара молодых людей подвергается процедуре стирания памяти после разрыва отношений. Воспоминания одного из них затем представлены в обратном порядке. А в фильме *Точка обстрела* с разных точек зрения показаны события, ведущие к попытке покушения на президента США. Каждое описание является неполным, но дополняет сюжет подробностями и фактами, а зрителю приходится собирать все это вместе, чтобы понять сюжет.

Книга *Облачный атлас* британского писателя Дэвида Митчелла (David Mitchell) состоит из шести вложенных одна в другую историй. Чтобы получить полную версию этих историй, читателю придется переупорядочить разные части книги. Любопытный вариант трудной задачи упорядочения представлен в романе *Игра в классики* аргентинского писателя Хулио Кортасара (Julio Cortázar), в котором содержатся два разных ряда явно указанных инструкций для чтения глав книги.

Часть II

ЯЗЫКИ



Языки и смысловое значение

Над радугой

Предписания врача

После обеда вы отправляетесь на прием к врачу. Обследовав вас, врач выписывает вам рецепт и заполняет направление на некоторые анализы крови. Вы берете это направление и идете в лабораторию, сдаете анализ крови, а затем отправляетесь с рецептом в аптеку. Пока что нет ничего удивительного в том, что лаборант, подвергающий сданную вами кровь ряду анализов, выполняет определенный алгоритм, как, впрочем, и аптекарь, подбирающий вам лекарства по рецепту врача. Но в этой ситуации примечательно то обстоятельство, что алгоритм определяется одним человеком, а выполняется другим.

Такое разделение труда оказывается возможным потому, что алгоритм написан на конкретном *языке*. И хотя врач мог бы позвонить в аптеку и дать инструкции непосредственно аптекарю, это усложнило бы процесс, поскольку потребовало бы от врача и аптекаря совместной работы. Представление алгоритма в письменной форме позволяет им работать независимо друг от друга и в разное время. Более того, однажды выписанный рецепт может быть использован неоднократно.

Разделение определения и выполнения алгоритма делает очевидной потребность в точном определении самого языка. Так, врач и аптекарь должны согласовать форму рецепта, его содержимое и назначение, ведь от этого зависит здоровье пациента. Например, дозы лекарств и соответствующие единицы измерения должны быть указаны так, чтобы не допустить никаких неоднозначностей. Если же врач и аптекарь интерпретируют дозы лекарств в разных единицах измерения, то неоднозначность интерпретации может сделать дозы лекарств неэффективно малыми или опасно большими. Аналогичные договоренности должны быть и между врачом и лаборантом.

Аптекарь и лаборант выступают в роли компьютеров, которые должны выполнять алгоритмы, данные им врачом. И только если они знают, как правильно читать и интерпретировать инструкции врача, они смогут успешно выполнить их, чтобы цель врача помочь пациенту была достигнута.

Первый шаг в выполнении написанного алгоритма состоит в том, чтобы провести его синтаксический анализ, т.е. извлечь его базовую структуру. Эта структура определяет главные составляющие алгоритма (в упомянутом выше примере рецепта — название лекарств, их дозы и частоту приема), а также их взаимосвязь

(например, соответствует ли указанная доза отдельному приему или общему количеству лекарства). Как только структура алгоритма будет выяснена, смысловое назначение языка, выражающее данный алгоритм, определяет, что именно должны делать аптекарь и лаборант. Сам процесс синтаксического анализа является алгоритмом для извлечения базовой структуры любой данной последовательности.

Языки, употребляемые для назначения анализов крови и выписывания рецептов, сильно разнятся. Помимо содержимого, они отличаются внешне. Так, если рецепт обычно состоит из последовательности слов, описывающих лекарства, их дозы, частоту приема и так далее, то назначение анализов крови принято давать в форме с несколькими клетками для пометки "галочками". Нередко конкретная форма языка обусловлена историческими причинами, но она может быть и результатом сознательного проектного решения для достижения ряда целей языка. Например, форма с клетками для пометки "галочками" отражает внутреннюю структуру языка и упрощает задачу написания и истолкования заказов, включая выписывание счетов, а также позволяет избежать потенциальной неоднозначности. В частности, форма назначения может быть разработана с целью предотвратить излишние анализы крови.

Язык играет центральную роль в информатике. Без языка нельзя было бы говорить о вычислении, алгоритмах и их свойствах. Во многих предметных областях были выработаны специальные языки со своей терминологией и обозначениями. Помимо применения языков, специалисты по информатике изучают сами языки, как это обычно делают лингвисты и филологи. В частности, специалисты по информатике решают вопросы, касающиеся точного определения языков, их обязательных свойств и порядка разработки новых языков. Они изучают формальные представления для обозначения, анализа и трансляции языков, а также разрабатывают алгоритмы для автоматического решения этих задач.

Философ Людвиг Виттгенштейн (Ludwig Wittgenstein) однажды написал: "Пределы моего языка означают пределы моего мира" [1]. В главе 8, "Сквозь призму языка", язык музыки используется для наглядного представления понятий языка, относящихся к информатике, а также для того, чтобы показать, что применения языка в этом мире имеют мало ограничений.

8 Сквозь призму языка

Мы свободно пользуемся языком каждый день, даже не осознавая сложности механизмов, задействованных в выполнении языком своих функций. Это сродни ходьбе. Однажды научившись ходить, мы можем пересекать водные преграды вброд, ходить по песку, подниматься по лестнице и преодолевать препятствия. Сложность задачи становится очевидной, когда мы пытаемся построить имитирующего подобное поведение робота. То же самое относится и к языкам. Пытаясь обучить машины эффективному использованию языков, мы осознаем, насколько это трудно на самом деле. Приходилось ли вам испытывать разочарование, пользуясь личным помощником Siri, установленным на смартфоне iPhone, или не сумев найти то, что нужно, в поисковом механизме Google? Тест Тьюринга для проверки машинного интеллекта идеально отражает это положение дел. Согласно этому тесту машина считается разумной, если пользователь не в состоянии отличить ее во время беседы от человека. Иными словами, владение языком служит тестом искусственного интеллекта.

Язык как явление изучался во многих дисциплинах, включая философию, лингвистику и социологию, и поэтому отсутствует единое согласованное определение того, что же такое язык. С точки зрения информатики язык — это *точные и эффективные средства передачи смыслового значения.* В главе 3, "Тайна знаков", было показано, каким образом знаки образуют основание для представления и способны придать смысловое значение вычислениям, связав символы с понятиями, которые их обозначают. Если знаки способны представлять лишь отдельные понятия, то язык определяет значащие комбинации знаков в предложения и

представляет взаимосвязи между этими понятиями. Как знаки, так и языки являются представлениями, но если знаков достаточно для представления объектов, интересных для конкретного языкового общения, то для представления вычислений через алгоритмы приходится обращаться к языкам.

В этой главе поясняется, что такое язык и каким образом языки определяются. Сначала в ней наглядно показано, как язык позволяет представить алгоритмы (а следовательно, и вычисления), принимая во внимание особую важность языков как отдельного предмета в информатике. Затем в главе показано, как языки могут быть определены через грамматику. Язык состоит из предложений, и если предложение нередко рассматривается просто как набор символов или слов, то такое представление оказывается слишком узким. Ведь это все равно что рассматривать рисунок сцены, распознавая объекты, но игнорируя их взаимосвязи. Представьте рисунок, изображающий человека, собаку и палку. Совсем разное значение имеет, несет ли собака палку, направляясь к человеку, или же человек держит палку, а собака пытается его укусить. Аналогично в языке каждое предложение обладает внутренней структурой, играющей важную роль в собирании его смыслового значения. Чтобы продемонстрировать эту особенность языка, здесь будет показано, что грамматика не только определяет внешний вид предложений как последовательностей слов (т.е. *конкретный синтаксис*), но и их внутреннюю структуру (т.е. *абстрактный синтаксис*).

❋ ❋ ❋

В этой главе речь пойдет не о самом вычислении, а об описании вычисления или, скорее, об описании самого описания вычисления. А что это означает? Несмотря на то что алгоритм является описанием вычисления, в этой главе речь пойдет не о конкретных алгоритмах, а о том, как описываются алгоритмы. Оказывается, что такое описание приравнивается к языку.

Когда приходится объяснять, что такое автомашина или цветок, мы говорим не о конкретных марках автомашин или видах цветов, а обо всех автомашинах и цветах вообще, так сказать, о сущности автомашин и цветов. Отдельные марки автомашин или виды цветов, вероятнее всего, будут упомянуты как примеры, но для того чтобы говорить обо всех возможных автомашинах и цветах, придется выработать *модель*, представляющую автомашину или цветок. Такая модель определяет *тип* всех автомашин или цветов (см. главу 14, “Волшебный тип”).

Шаблоны, состоящие из фиксированной структуры с некоторым количеством переменных частей, по существу, могут служить моделями для автомашин или цветов, поскольку б*о*льшая часть их собственной структуры фиксирована. Несмотря на то что шаблоны вполне пригодны для описания некоторых разновидностей алгоритмов, например, электронных таблиц или форм для назначения

анализов крови, модель, основанная на шаблоне, оказалась бы слишком жесткой и недостаточно выразительной для алгоритмов в целом. Необходимую гибкость для описания произвольных алгоритмов обеспечивают языки.

В истории, сопутствующей этой главе, сам язык играет заметную роль. В качестве предметной области для изучения понятий языка здесь выбрана музыка, поскольку музыкальные языки просты и достаточно структурированы, чтобы легко ощутить сущность поясняемых понятий. Область музыки достаточно узка, чтобы поддаваться специальному обозначению, тем не менее она общепонятна, а значит, может быть принята на веру, когда обсуждение сосредоточено непосредственно на музыкальных языках.

Идея считать музыку языком не нова. Например, воспользовавшись идеями, восходящими к Пифагору, астроном Иоганн Кеплер употребил музыкальные понятия для объяснения законов, определяющих частоту обращения планет. В своем романе *Человек на Луне* (1638) Фрэнсис Годвин (Francis Godwin) описывает, как обитатели Луны пользуются для общения музыкальным языком. А в фильме *Близкие контакты третьей степени* люди общаются с пришельцами, пользуясь пентатоникой, т.е. пятитональной музыкальной гаммой. Будучи высокоструктурированной, но при этом конкретной культурной средой, музыка отлично подходит для пояснения концепций языков.

Замечания по поводу мелодии

Мелодия *Over the Rainbow* (Над радугой) относится к самым популярным композициям XX века в США [1]. Она была написана Харольдом Арленом (Harold Arlen) [2] для фильма *Волшебник страны Оз*. В начале этого фильма героиня Дороти, роль которой играла Джуди Гарланд (Judy Garland), желает узнать, где находится место без тревог, о чем она и поет в своей песне.

Эту мелодию можно было бы прослушать непосредственно при наличии аудиокниги или в виде видеозаписи в YouTube, но как передать ее при отсутствии необходимых аудиосредств? Для этого необходимо такое представление музыки, интерпретация которого позволила бы ее воспроизвести. Одним из таких широко используемых сегодня представлений служит *стандартная нотная запись*, иначе называемая *нотным станом*. В качестве примера ниже приведена небольшая часть упоминаемой здесь мелодии, представленная в нотной записи.

Не особенно отчаивайтесь, если вы не знаете нотной записи. Все необходимые понятия будут пояснены по ходу изложения материала этой главы. А до тех пор достаточно знать, что овальные символы представляют отдельные ноты мелодии, их вертикальное расположение определяет высоту тона данной ноты, длительность ее звучания обозначается присоединяемым к ней "крючком" и тем, как она показана: как заполненный или пустой кружок. Высота тона и длительность звучания являются основными элементами, образующими мелодию. Напевая или слушая мелодию и следя за ней по нотам, или партитуре, можно вполне разобраться в смысловом значении отдельных нот в частности и в нотной записи вообще.

Эту партитуру можно считать предложением в языке нотной записи. Более того, партитура служит описанием алгоритма для порождения музыки. Всякий музыкант, понимающий этот язык, может исполнить такое предложение, по существу, превращаясь в компьютер. Аналогия с вычислением наглядно показана на рис. 8.1. Генерируемое вычисление производит представления звуков, которые могут принимать совершенно разные формы. Для извлечения звуков вокалист приводит в движение свои голосовые связки, а пианист, гитарист или виолончелист начинает и останавливает вибрирование струн, используя клавиши и молоточки, пальцы и смычок.

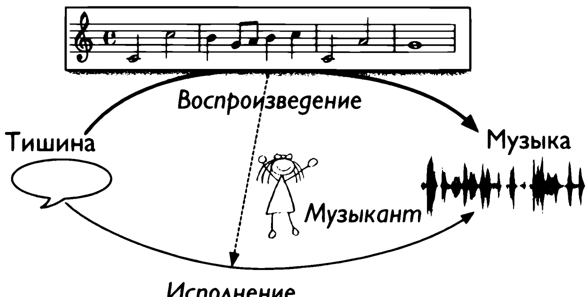
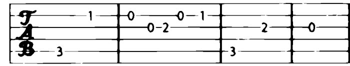


Рис. 8.1. В процессе воспроизведения (выполнения) музыкальной партитуры (алгоритма) генерируется исполнение (вычисление), преобразующее тишину в музыку. Исполнение производится музыкантом (компьютером), т.е. человеком или машиной, способной понимать нотную запись (язык) музыкального произведения. (См. также рис. 2.1.)

Нотная запись примечательна тем, что она упрощает верное воспроизведение мелодии — даже теми людьми, которые прежде ее не слышали. Композитор Харольд Арлен, придумавший упоминаемую здесь мелодию и закодировавший ее в нотной записи, мог бы просто послать партитуру Джуди Гарланд, и этого было бы достаточно, чтобы она исполнила песню надлежащим образом. В отсутствие такой записи единственный способ сделать музыку общим достоянием — записать ее с помощью аудиоаппаратуры или воспроизвести на слух, т.е. исполнить перед

другими людьми, которые должны запомнить мелодию и воспроизвести ее. Если вам приходилось когда-нибудь играть в испорченный телефон, то вам должно быть известно, насколько ненадежен такой способ [3][1].

Размышляя над конструкцией нотного стана, можно заметить, насколько она произвольна. Несмотря на то что нота является индексным знаком (см. главу 3, "Тайна знаков"), высота расположения которого отражает высоту тона, представленного данной нотой, по-видимому, нет никаких особых причин употреблять именно пять линий нотного стана, овальную форму знаков нот или конкретную длину и направление шеек нот для записи мелодии — все это можно было бы сделать и по-другому. И действительно, это иногда и делается по-другому.



Например, запись табулатуры для игры на гитаре основана на совершенно ином представлении. Она показывает непосредственно взаимодействие струн гитары, а следовательно, исключает абстрактное понятие нот. Число на линии табулатуры обозначает место, где следует прижать струну к грифу, перебирая ее. Одно из преимуществ такой записи заключается в ее непосредственности: она позволяет начинающим быстро научиться играть мелодии, не требуя усвоить сначала абстрактную нотную запись. А недостаток такой записи заключается в том, что она неточно отражает длительность нот. Кроме того, ее применение ограничивается лишь одним видом музыкальных инструментов.

Запись табулатуры отражает физическую структуру гитары, а следовательно, она менее произвольна. Например, у гитары обычно имеется шесть струн, и поэтому конструкция такой записи требует наличия шести горизонтальных линий и присутствия на них только чисел. И напротив, пять линий нотного стана выбраны произвольно, хотя их число может быть расширено по мере надобности. В приведенной выше нотной записи самая первая нота выполняет именно эту функцию, служа вспомогательной линией.

Табулатура и нотный стан — это два разных языка представления в области музыки. Каждый язык определяется набором правил, которые определяют, какие символы могут использоваться в записи и как их можно комбинировать. Эти правила определяют *синтаксис* языка; они используются для того, чтобы отличать правильные элементы языка, именуемые *предложениями*, от тарабарщины. Правильно составленный музыкальный фрагмент является предложением. Любые

[1] Здесь остается только вспомнить известный анекдот, в котором некто критикует известного оперного певца, а на вопрос, слышал ли он хоть одно его выступление, поясняет, что лично не слышал, но вот сосед вчера продемонстрировал его пение — ни слуха, ни голоса... — *Примеч. ред.*

обозначения, которые нарушают правила, не могут быть надлежащим образом выполнены в качестве алгоритма. Например, нет смысла использовать отрицательные числа в табулатуре. Гитарист не будет знать, как их интерпретировать и что с ними делать. Точно так же, если бы мы использовали на нотном стане несколько дополнительных линий, Джуди Гарланд не знала бы, какую ноту петь.

Исполнитель музыки может воспроизвести ее лишь в том случае, если ее нотная запись ясна и однозначна [4]. И это еще один способ сказать, что нотная запись, как и любая другая алгоритмическая запись, должна эффективно представлять исполняемые шаги, которые могут быть однозначно интерпретированы. Таким образом, чтобы гарантировать эффективность любого музыкального (и любого другого алгоритмического) языка, необходимо, прежде всего, точно определить, что в данном языке следует считать предложением.

Грамматические правила

Синтаксис языка можно определить с помощью *грамматики*, под которой понимается свод правил для построения предложений в данном языке. Употребление таких естественных языков, как испанский или английский, для описания синтаксиса — далеко не самая лучшая альтернатива, поскольку такие описания окажутся длинными и неточными. Именно поэтому математические уравнения или физические законы обычно выражаются не словесно, а с помощью некоторых специальных обозначений. Фактически во многих областях науки и техники выработаны собственные терминология и обозначения, чтобы сделать возможным эффективный обмен идеями в соответствующих предметных областях. Это справедливо и для изучения языков, будь то в области языковедения или информатики, и одну из специальных записей для определения языков дает грамматика.

Трудность описания языка состоит в конечном представлении потенциально бесконечного множества предложений. Аналогичную трудность испытывает и наука, которой приходится описывать бесконечное множество фактов с помощью небольшого ряда законов. Способ преодоления подобной трудности наукой помогает в объяснении некоторых понятий грамматики. Рассмотрим в качестве примера известное физическое уравнение $E = mc^2$, связывающее энергию (E), которой обладает объект, с его массой (m). Конкретное смысловое значение этого уравнения в данном случае не так важно, как наличие в нем *константы* c и двух переменных, E и m, которые могут обозначать произвольные положительные числа. В частности, обе переменные в этом уравнении допускают вычисление энергии, которой обладает объект, независимо от его величины и сложности. Благодаря использованию переменных в одном предложении удается представить бесконечное число физических явлений. Переменная служит в уравнении той же самой цели, что и параметр в алгоритме.

Как и в уравнениях, в грамматике содержатся константы и переменные; первые здесь называются *терминальными символами* или, более кратко, *терминалами*, а вторые — *нетерминальными символами* или *нетерминалами*. Причины таких названий станут вскоре ясны. Предложение в языке задается последовательностью терминальных символов и вообще не содержит нетерминальных символов, играющих лишь вспомогательную роль в построении предложений. Например, отдельные ноты на нотном стане появляются на нотоносцах и тактовых чертах, группирующих ноты в такты. Безусловно, нотная запись содержит и другие терминальные символы, хотя нот и тактовых черт оказывается достаточно, чтобы наглядно представить грамматические понятия, требующиеся для определения простых мелодий.

Например, терминалами, составляющими первый такт мелодии *Над радугой*, являются два нотных знака, и , после которых следует тактовая черта . Подобно константе в уравнении, терминальный символ представляет фиксированную, неизменяемую часть предложения, а само предложение соответствует одному конкретному научному факту, получающемуся из уравнения путем подстановки чисел вместо переменных.

А нетерминальный символ действует подобно переменной в уравнении, способной принимать разные значения. Вместо нетерминального символа может быть подставлена последовательность других (терминальных и нетерминальных) символов. Но такая подстановка не произвольна. Вместо этого все возможные подстановки определяются сводом правил. Нетерминальный символ подобен заполнителю и, как правило, служит для представления конкретной части языка. Правила подстановки определяют внешний вид этих частей в категориях последовательностей терминалов. Например, в определении упрощенной грамматики для нотного стана можно обнаружить нетерминальную *ноту*, которая может обозначать произвольный терминальный символ ноты [5].

Грамматика состоит из ряда *правил*. Каждое такое правило задается подставляемым нетерминалом, стрелкой, указывающей на подстановку, а также последовательностью символов, заменяемых нетерминалом. Эта заменяемая последовательность символов иначе называется правой частью правила (right-hand side — RHS). Простым тому примером служит правило *нота* →, правая часть которого состоит только из одного терминального символа. Соответствие составных частей грамматики, уравнений и алгоритмов сведено в табл. 8.1. Грамматика состоит из правил подобно тому, как алгоритм — из отдельных инструкций. Для уравнений здесь имеются соответствующие составные части.

Таблица 8.1. Соответствие составляющих частей грамматики, уравнений и алгоритмов

Грамматики	*Уравнения*	*Алгоритмы*
Нетерминал	Переменная	Параметр
Терминал	Константа, операция	Значение, инструкция
Предложение	Факт	Алгоритм только со значениями
Правило		Инструкция

Ноты различаются высотой тона и длительностью звучания, и для них имеется только один терминал, поэтому для каждой *ноты* требуется большое количество правил, как показано на рис. 8.2 [6]. Нетерминальная *нота* представляет категорию отдельных правил, которая определяется через все правила для *ноты*.

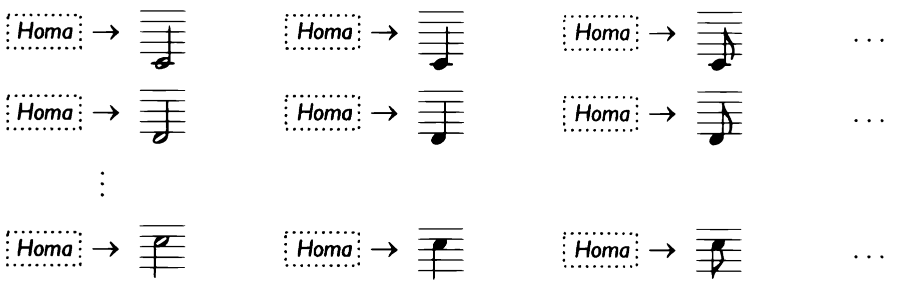


Рис. 8.2. Грамматические правила определяют возможные подстановки для нетерминала ноты. Поскольку произвольная нота представлена одним нетерминалом, для каждого сочетания высоты тона и длительности звучания требуется отдельное правило. В отдельных столбцах на этом рисунке показаны правила для нот с конкретной длительностью звучания. Слева: половинные ноты, звучащие в течение полутакта. Посредине: четвертные ноты. Справа: восьмые ноты

В общем случае правая часть правила может состоять из нескольких символов, которые могут быть как терминалами, так и нетерминалами. Последовательность, состоящая только из терминальных символов, не может быть изменена, поскольку правила допускают замену только нетерминальных символов. Этим также объясняются названия *терминалы* и *нетерминалы*. Завершенная последовательность терминалов является предложением языка, описываемым грамматическими правилами. С другой стороны, последовательность, содержащая нетерминальные символы, еще не завершена и не является предложением языка. Такая последовательность иначе называется *сентенциальной формой*, поскольку обычно описывает целый класс предложений (сентенций), которые могут быть получены из нее в результате подстановки нетерминалов. Сентенциальная форма подобна уравнению, в которое могут быть подставлены некоторые, хотя и не все, переменные, или алгоритм, в котором некоторые, хотя и не все, параметры могут быть заменены входными значениями.

Сентенциальные формы можно обнаружить и в правилах, определяющих мелодию. Нетерминальная *мелодия* обозначает все мелодии, которые могут быть сформированы грамматически. При первом подходе мелодии можно определить по следующим трем правилам (каждому правилу присваивается имя для последующей ссылки):

мелодия → *нота* *мелодия* (**Новая_нота**)

мелодия → *нота* ǂ *мелодия* (**Новый_такт**)

мелодия → *нота* (**Последняя_нота**)

Первое правило (Новая_нота) гласит: мелодия начинается с некоторой ноты, после которой следует другая мелодия. Такое правило может показаться, на первый взгляд, необычным, но если рассматривать мелодию как последовательность нот, то данное правило гласит, что последовательность нот начинается с ноты, за которой следует другая последовательность нот. Такая форма правила выглядит особенной потому, что заменяет символ правой частью данного правила, содержащей именно этот символ; поэтому данное правило считается *рекурсивным.* Такое рекурсивное правило само по себе не обеспечивает подстановку, которую предполагается совершить. И если бы для *мелодии* имелись только рекурсивные правила, грамматика в действительности оказалась бы проблематичной, поскольку нам, пожалуй, так и не удалось бы избавиться от нетерминала. Но третье правило в данном примере (Последняя_нота) не является рекурсивным и всегда может быть использовано для замены нетерминала мелодии нетерминалом ноты, который мог бы быть далее заменен терминальным символом ноты. Вместо третьего правила Последняя_нота можно было также применить следующую его альтернативу с пустой правой частью:

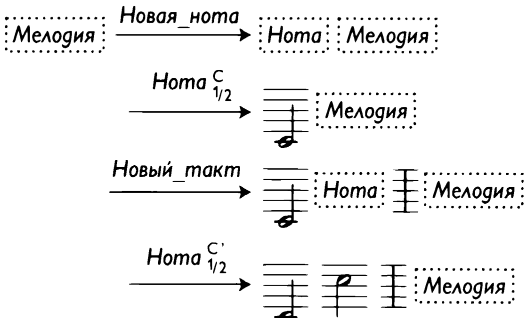
мелодия → (**Конец_мелодии**)

Данное правило гласит: заменить *мелодию* пустой последовательностью символов (т.е. ничем). Такое правило может быть использовано для удаления нетерминала мелодии из сентенциальной формы.

Рекурсивные правила обычно служат для формирования последовательности из некоторого количества символов путем повторяющихся применений.

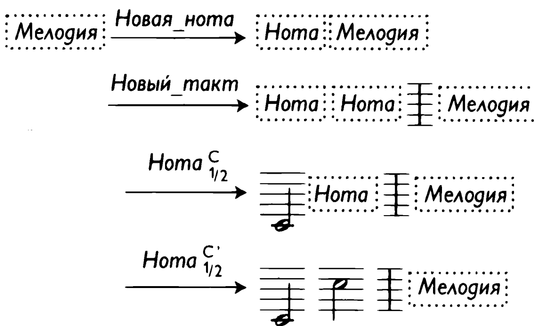
Второе правило (Новый_такт) подобно первому правилу и также допускает повторяющееся формирование нетерминалов нот. Но кроме того, оно формирует терминальный символ тактовой черты ǂ, который указывает на конец одного такта и начало нового.

Начиная, например, с нетерминального символа *мелодии*, можно сформировать последовательности символов, неоднократно заменяя нетерминалы через приложение грамматических правил. Например, первый такт мелодии *Над радугой* может быть произведен следующим образом:



Стрелка с меткой в каждой строке указывает на примененное правило. Например, сначала применяется правило Новая_нота для формирования нетерминала первой ноты. Затем этот нетерминал заменяется терминалом, представляющим первую ноту мелодии. Применяемое конкретное правило из рис. 8.2 обозначается аннотациями к высоте тона и длительности звучания как $Нота^{C}_{1/2}$, где C — указание ноты "до", а ½ — продолжительность звучания в течение половины целого такта. Этот процесс продолжается с помощью правила Новый_такт для формирования нетерминала другой ноты, после чего следует терминал — черта, завершающая текущий такт. На следующем шаге вместо нетерминала новой ноты также подставляется терминал этой ноты.

Решение о применении конкретных правил определяет формируемую мелодию. Следует, однако, иметь в виду, что порядок, в котором применяются правила, оказывается в определенной степени гибким. Например, применяемые правила $Нота^{C}_{1/2}$ и Новый_такт можно поменять местами, чтобы, как показано ниже, получить ту же самую сентенционную форму. Аналогичным образом можно было бы изменить порядок, в котором заменялись бы нетерминалы двух нот, чтобы добиться того же самого результата.



Завершить последовательный ряд применений правил можно, применив третье правило (`Последняя_нота` или `Конец_мелодии`), исключающее нетерминал оставшейся мелодии. (Так, если применяется правило `Последняя_нота`, то придется применить еще одно правило `Note`, чтобы исключить получающийся нетерминал ноты.) Полученная в итоге последовательность терминалов является предложением языка, а последовательность сентенциальных форм и приложений правил (от нетерминала первоначальной мелодии до конечного предложения) называется *выводом* (derivation). Последовательность терминалов становится предложением языка лишь в том случае, если такой вывод существует, а решение о принадлежности предложения к языку сводится к поиску соответствующего вывода. Сам же вывод доказывает, что результирующая последовательность терминалов является элементом языка.

Один из нетерминалов грамматики обозначается как *стартовый*, или *начальный символ*, представляющий основную категорию предложений, определяемых грамматикой. Такой нетерминал дает грамматике свое имя. Например, начальный символ грамматики должен обозначать *мелодию*, поскольку назначение такой грамматики — определять мелодии, и благодаря этому на грамматику можно ссылаться как на грамматику мелодий. Язык, определяемый грамматикой мелодий, состоит из всех предложений, которые могут быть выведены из начального символа *мелодии*.

Структура растет на деревьях

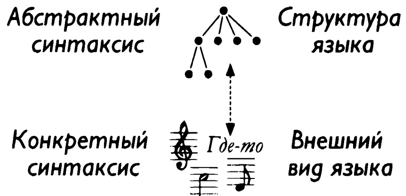
Наличие более одного языка в конкретной предметной области (например, нотного стана и табулатуры в музыке) может показаться странным, и в связи с этим может возникнуть следующий вопрос: имеются ли для этого какие-нибудь веские основания? Возможно, было бы лучше иметь только один язык, который мог бы служить стандартом? Мы ценим разнообразие пищи, одежды, мест отдыха и прочего, но необходимость обращаться с разными языками нередко оказывается неприятным и затратным делом. Ведь для этого требуются перевод и уточнение, что в конечном счете может привести к недоразумениям и ошибкам. В истории о вавилонской башне существование многих языков считается наказанием. Попытки создать универсальные языки наподобие эсперанто были направлены на преодоление трудностей, обусловленных наличием слишком большого числа языков. Кроме того, комитеты по стандартизации языков постоянно борются за сохранение контроля над многообразием технических языков.

Зачем же иметь так много языков, несмотря на все затраты? Зачастую новый язык принимается потому, что он служит конкретной цели. Например, языки вроде HTML или JavaScript могут оказать помощь в представлении информации в Интернете, что крайне полезно для многих коммерческих и общественных

организаций. В области музыки запись табулатуры оказывается вполне пригодной для играющих на гитаре и, в частности, для тех, кто не знает нот. С появлением программируемых музыкальных автоматов (например, синтезаторов и ритм-машин) был изобретен язык MIDI (Musical Instrument Digital Interface — цифровой интерфейс музыкальных инструментов) для кодирования управляющих сообщений, предписывающих синтезаторам производить звуки. В качестве примера ниже приведено начало мелодии *Над радугой* в формате MIDI.

```
4d54 6864 0000 0006 0001 0002 0180 4d54 : : :
```

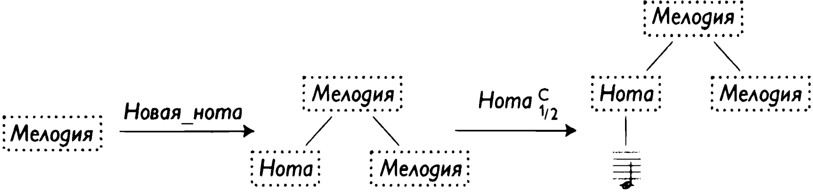
Однако несмотря на эффективность управления синтезатором, такое представление мелодии не очень удобно для пользователя. Ведь из него неясно, что именно означают цифры и буквы и как они связаны с музыкой, которую должны представлять. Для удобства пользователей лучше хранить ноты и табулатуру и переводить партитуры в формат MIDI, только если требуется снабдить синтезатор музыкой. Может также потребоваться взаимный перевод нот и табулатуры. Такой перевод, безусловно, хотелось бы выполнять автоматически, используя определенный алгоритм, и при этом сохранить смысловое значение того, что представлено, т.е. музыки.



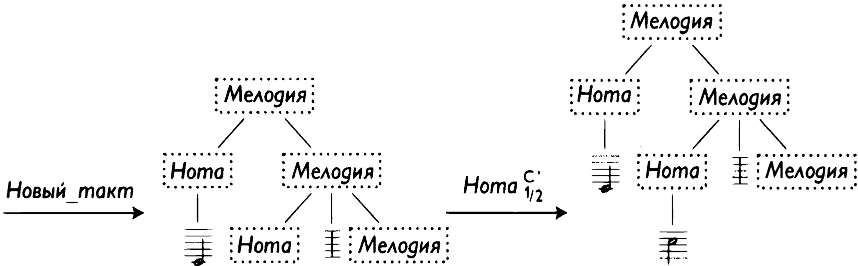
Взаимный перевод разных видов записи лучше всего выполнить через промежуточное представление, именуемое *абстрактным синтаксисом*. В отличие от *конкретного синтаксиса*, который определяет текстовый или визуальный *внешний вид* предложения, абстрактный синтаксис обнаруживает *структуру* предложения в иерархической форме. Конкретный синтаксис музыкального произведения на нотном стане задается последовательностью нот, тактовых черт и других символов, чего недостаточно для фиксации иерархической структуры музыкального произведения. Для этой цели можно использовать деревья, применявшиеся в главах 4, "Записная книжка сыщика", и 5, "Поиск идеальной структуры данных", для представления иерархии семейных отношений и словаря. Абстрактный синтаксис представлен *абстрактным синтаксическим деревом*.

Преобразование конкретного синтаксиса в абстрактное синтаксическое дерево может быть достигнуто за два шага. Во-первых, последовательность символов в конкретном синтаксисе превращается в *дерево синтаксического анализа*, в котором фиксируются иерархические отношения между символами. И во-вторых, дерево

синтаксического анализа упрощается до абстрактного синтаксического дерева. Дерево синтаксического анализа может быть построено наряду с выводом предложения. Напомним, что на каждом шаге вывода нетерминал заменяется правой частью правила, составленного для нетерминала. Но теперь вместо замены нетерминала правой частью правила для каждого символа из правой части просто вводится узел дерева, соединяемый ребром с нетерминалом. Таким образом, на каждом шаге вывод абстрактного синтаксического дерева расширяется добавлением новых узлов к нетерминалу на краю дерева. Если обратиться к предыдущему выводу, то сначала получается приведенная ниже последовательность деревьев, наглядно показывающая, каким образом применение правила расширяет дерево вниз.



Оба эти шага прямы и просты. Два следующих шага приводят к несколько неожиданному результату, поскольку дерево синтаксического анализа не раскрывает структуру мелодии. Дерево синтаксического анализа не показывает, как именно мелодия состоит из тактов, а те — из нот.



Недостаточность структуры является следствием того, каким образом была ранее определена грамматика: ее правила просто расширяют мелодию в последовательность нот. Поэтому не удивительно, что в дереве синтаксического анализа вообще не упоминается о тактах. Положение можно исправить, изменив определение грамматики, чтобы учесть такты. На рис. 8.3 показаны части деревьев синтаксического анализа и абстрактного синтаксического дерева. В информатике деревья строятся вверх дном, когда корень оказывается вверху, а деревья — внизу. При этом нетерминалы служат местами для ветвления дерева, а терминалы — его листьями.

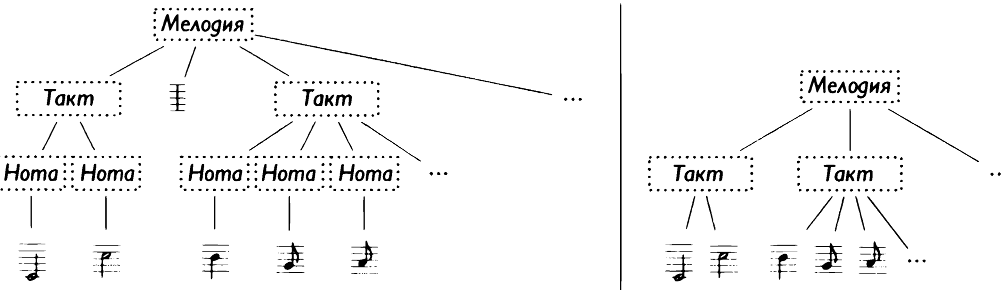


Рис. 8.3. Структура мелодии, представленная синтаксическими деревьями. В синтаксическом дереве фиксируется структурный результат вывода, но при этом игнорируются такие подробности, как порядок применения конкретных правил. Слева: дерево синтаксического анализа, в котором фиксируются все детали вывода. Справа: абстрактное синтаксическое дерево, в котором опущены ненужные подробности и сохраняется только структурная информация

Дерево синтаксического анализа, приведенное на рис. 8.3, является непосредственным результатом превращения вывода в дерево. В нем сохраняются все подробности вывода — даже те части, которые не требуются для перевода дерева в другие обозначения. А в абстрактном синтаксическом дереве игнорируются терминалы и нетерминалы, несущественные для структуры предложения. Например, терминалы тактовых черт избыточны, поскольку группирование нот в такты уже зафиксировано нетерминалами тактов. Нетерминалы нот вообще не нужны, поскольку каждый из них всегда расширяется ровно в один нотный терминал. При добавлении терминалов нот непосредственно как потомков к нетерминалам тактов фиксируется та же самая структурная информация, а следовательно, подтверждается удаление нетерминалов нот. Дерево синтаксического анализа и абстрактное синтаксическое дерево разделяют общую структуру. В частности, корни обоих видов деревьев задаются как нетерминал мелодии, а все их листья — как терминальные символы. Абстрактное синтаксическое дерево более явно отражает структуру предложения и служит основанием для анализа и перевода. Оно упрощает определение смыслового значения отдельного предложения.

Процесс построения дерева синтаксического анализа или абстрактного синтаксического дерева для заданного предложения называется *синтаксическим анализом*. Отношения между предложениями, синтаксическими деревьями и синтаксическим анализом наглядно показаны на рис. 8.4. Синтаксический анализ является вычислением, для которого существует несколько разных алгоритмов. Несмотря на то что в грамматике точно определяется, какие именно предложения относятся к языку, для синтаксического анализа также требуется определенная стратегия превращения заданного предложения в синтаксическое дерево. Трудность синтаксического анализа заключается в выборе подходящих грамматических правил во время анализа предложения. Для синтаксического анализа предложений имеются

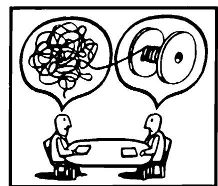
разные стратегии. Одна из них рассматривалась ранее и называется *нисходящим анализом*, который начинается с начального символа грамматики и состоит в повторном применении правила для постепенного построения синтаксического дерева путем расширения нетерминалов. Данный процесс продолжается до тех пор, пока предложение не примет вид последовательности терминалов в листьях дерева. При *восходящем анализе* предпринимается попытка сопоставить правые части правил в предложении и применить эти правила в обратном порядке. Цель такого анализа состоит в построении синтаксического дерева путем добавления к этому дереву нетерминалов из левой части совпавших правил в качестве его родителей. Данный процесс повторяется до тех пор, пока не будет получено дерево с единственным корневым узлом.

Рис. 8.4. Синтаксический анализ — это процесс выявления структуры предложения и ее представления в виде синтаксического дерева. Структурная распечатка вида преобразует синтаксическое дерево в предложение. Поскольку в абстрактном синтаксическом дереве могут быть опущены некоторые терминальные символы (например, тактовые черты), такая распечатка не может только собирать терминалы в листьях синтаксического дерева, но в общем случае должна применять грамматические правила для вставки дополнительных терминальных символов. Стрелка синтаксического анализа в записи табулатуры отсутствует, поскольку это неоднозначная запись, а следовательно, она не допускает построение однозначного абстрактного синтаксического дерева

Противоположный процесс преобразования синтаксического дерева в конкретный синтаксис называется *структурной распечаткой*. Это достаточно простое вычисление, поскольку структура предложения уже задана и служит руководством для создания конкретного представления.

Синтаксический анализ и структурная распечатка являются необходимыми инструментальными средствами для взаимного перевода между языками. Перевод между двумя языками с одинаковым абстрактным синтаксисом выполняется путем синтаксического анализа предложения на первом языке и применения

структурной распечатки на втором языке. Так, на рис. 8.4 показано, как перейти от нотного стана к табулатуре, сначала выполнив синтаксический анализ предложения в нотном стане, а затем применив структурную распечатку табулатуры к полученному абстрактному синтаксическому дереву. Для взаимного перевода на языки с разными абстрактными синтаксисами требуется дополнительное взаимное преобразование абстрактных синтаксических деревьев.



Синтаксический анализ является также необходимым первым шагом в определении смыслового значения отдельного предложения, поскольку он зависит от структуры самого предложения, представленной абстрактным синтаксическим деревом. Это очень важное наблюдение, заслуживающее повторения. Чтобы понять предложение, необходимо сначала установить его структуру в форме абстрактного синтаксического дерева [7]. То же самое справедливо и для прослушивания музыки. Ведь для того чтобы понять мелодию *Над радугой*, придется синтаксически проанализировать звуки и выявить разные ноты, порождающие данную мелодию. Кроме того, группирование нот в такты сопровождается акцентированием отдельных нот и осмыслением музыкальных фраз как частей мелодии. И, наконец, структурирование песни на более высоком уровне в основную ее тему и припев обеспечивают необходимую основу для распознавания повторов и лейтмотивов.

В связи с тем, что синтаксические деревья служат ключом к смысловому значению предложений, возникает следующий вопрос: всегда ли синтаксический анализ достигает успеха и что происходит при понимании предложения [8]? Недосказанному предложению недостает завершенного синтаксического дерева, а следовательно, его смысловое значение не до конца ясно. Но что если для одного предложения можно построить *несколько* разных синтаксических деревьев? Этот вопрос подробно рассматривается в главе 9, “Поиск нужного тона: смысл звука”.

Обращение из аптеки

Вы пришли в аптеку, чтобы подобрать себе лекарство, но возникла проблема: в рецепте оказалось недостаточно сведений о форме дозировки: в таблетках или в жидком виде? А поскольку рецепт выписан неоднозначно и страдает неточным смысловым значением, то он и не представляет алгоритм. В частности, такой рецепт не описывает ряд эффективных шагов, дающих аптекарю возможность подобрать вам лекарство.

Неоднозначность может возникнуть по нескольким причинам. В данном случае недостаточность информации отражается в абстрактном синтаксическом дереве с помощью нетерминала для формы лекарства, который не раскрывается. А поскольку аптекарь не может выполнить рецепт, предоставленный врачом, ему придется обратиться к врачу за разъяснениями. И как только неоднозначность будет устранена, аптекарь получит в свое распоряжение соответствующий алгоритм, позволяющий успешно приготовить для вас лекарство.

Разделение труда между врачом и аптекарем оказывается удачным потому, что оба они пользуются общим для рецептов языком. Но одного лишь этого недостаточно. Врач и аптекарь должны также одинаково понимать предложения на данном языке. А ведь это не простое требование, о чем свидетельствует тот факт, что сокращение названий лекарств и их доз фактически ведет к ошибкам отпуска лекарств по рецепту. Идея определить смысловое значение (т.е. *семантику*) языка путем присвоения явным образом смыслового значения каждому возможному предложению попросту наивна. Этот способ не годится не только для языков с бесконечным числом предложений; он весьма непрактичен даже для языков с конечным количеством предложений.

Вместо этого определение семантики языка следует делать в два взаимосвязанных этапа: сначала присвоить семантику отдельным словам, а затем определить правила для вывода семантики предложения из семантики его составных частей. Например, в форме для назначения анализов крови семантика отдельных клеток для отметки "галочкой" состоит в том, чтобы назначить один конкретный анализ крови. А семантика группы подобных клеток определяется как совокупность назначений, т.е. распоряжение привести в исполнение все назначения, полученные отдельно из семантики каждой клетки в группе. Для такого составного подхода

требуется определить синтаксис языка, что и делается в форме абстрактных синтаксических деревьев.

Языки для анализов крови и рецептов показывают, что семантика языка может принимать разные формы. Например, рецепт определяет алгоритм приготовления лекарства, тогда как форма для анализов крови предоставляет инструкции для взятия проб крови и проведения ее анализов. Компьютеры для выполнения алгоритмов на столь разных языках должны понимать разную семантику, а следовательно, должны иметь разную квалификацию для успешного исполнения своих обязанностей. Кроме того, язык для анализов крови наглядно показывает, что даже у одного языка может быть разная семантика, а именно: инструкции для взятия проб конкретного количества крови и проведения разных ее анализов. Таким образом, предложение на одном и том же языке может быть выполнено по-разному, зачастую разными компьютерами (например, флеботомистом и лаборантом) и дать не имеющие ничего общего результаты. Хотя это может быть даже полезно — для поддержания разделения труда.

Наличию разной семантики у одного языка можно найти важное применение в анализе алгоритмов с целью выявить и устранить потенциальные ошибки, что может помочь предотвратить вычисление неверных результатов. Сравните это с чтением документа разными людьми для выявления опечаток или грамматических ошибок, оценивания его содержимого или проверки его соответствия типографским нормам. Все эти задачи преследуют разные цели и могут быть решены до издания документа. Точно так же аптекарь должен перепроверить рецепт, прежде чем отпускать по нему лекарства, чтобы предупредить любые нежелательные последствия для пациента.

Языки распространены повсеместно. Помимо рецептов, лабораторных анализов, музыки и целого ряда других предметных областей, сама информатика изобилует языками — не считая даже языки программирования. Выше был представлен язык для описания грамматики. Имеются также языки для определения семантики других языков, синтаксических анализаторов и программ структурной распечатки и многого другого. Языки являются важнейшим инструментальным средством информатики для представления данных и вычислений. Эффективное использование языков зависит от наличия точной семантики, подобно тому как здоровье и безопасность пациентов зависят от ясной семантики рецептов. На базе мелодии *Над радугой* в главе 9, "Поиск нужного тона: смысл звука", будет наглядно показано, каким образом определяется семантика языка и какие трудности с этим связаны.

Поиск нужного тона: смысл звука

Как было показано в главе 8, “Сквозь призму языка”, партитура мелодии *Над радугой* является алгоритмом, который может быть выполнен музыкантами в виде воспроизведения музыки. Этот алгоритм был написан (т.е. придуман и закодирован) композитором Харольдом Арленом на языке нотного стана. Синтаксис этого языка может быть описан грамматикой, определяющей внешний вид предложения как партитуру, а его внутреннюю структуру — как абстрактное синтаксическое дерево.

В предыдущей главе было также показано, что один и тот же язык может быть описан разной грамматикой, что, на первый взгляд, может показаться странным. Но различия в грамматике приводят к различиям в абстрактном синтаксисе и представлениях структуры языка (например, музыки). Эти различия имеют значение, ведь когда Джуди Гарланд требуется исполнить песню *Над радугой*, она должна сначала проанализировать нотную запись этой песни синтаксически, чтобы обнаружить ее структуру. Иным словами, смысловое значение языка основывается на его абстрактном синтаксисе.

Именно поэтому неоднозначность представляет собой проблему для любого языка. Если у предложения может быть не одно абстрактное синтаксическое дерево, то не ясно, какой именно структуры следует придерживаться, применяя определение семантики. Следовательно, неоднозначное предложение может иметь не одно смысловое значение. Именно так обычно и определяется термин ***неоднозначность***.

В этой главе вопрос неоднозначности рассматривается более подробно, наряду с тем, каким образом предложения обретают смысловое значение. Ключевой для понимания данного вопроса момент состоит в том, что систематическое определение смыслового значения языка основывается на понятии *композиционности*. Это понятие означает, что семантика предложения получается из сочетания семантик его составных частей систематическим путем, определенным структурой предложения. Это показывает, что структура языка играет очень важную роль в определении его смыслового значения.

Звучит неубедительно

Язык нотной записи может быть описан довольно простой грамматикой. Но даже для столь простого языка совсем не ясно, какими именно грамматическими правилами следует пользоваться. Одним из недостатков, которыми может страдать язык, является его *неоднозначность*. Это означает, что одно предложение может иметь не одно, а несколько смысловых значений. Неоднозначность может проникнуть в предложение двумя разными путями. Во-первых, основные слова или знаки языка могут быть неоднозначны, и это явление называется *лексической неоднозначностью* или *омонимией* (об омонимах читайте в главе 3, "Тайна знаков"). Во-вторых, конкретное сочетание слов в предложении может быть неоднозначным, несмотря на то что отдельные слова вполне однозначны. Это явление называется *грамматической неоднозначностью*. В качестве примера рассмотрим предложение "Боб знает девушек больше, чем Алиса". С одной стороны, это предложение может означать, что Боб знаком не с одной девушкой, а с другой — у него знакомых девушек больше, чем у Алисы[1].

Грамматическая неоднозначность возникает в том случае, когда грамматика способна сформировать не одно, а несколько синтаксических деревьев для данного предложения. Обратившись снова к музыкальному примеру, рассмотрим следующую часть мелодии *Над радугой*. Любопытно, что ее партитура вообще *не* содержит тактовых черт, что обусловливает ее неоднозначность как музыкального предложения, поскольку не ясно, на какой именно ноте, первой или второй, следует делать акцент при ее воспроизведении.

[1] В русском переводе с английского появляется еще одна трактовка этой фразы: "Боб знает девушек лучше, чем Алиса". — *Примеч. ред.*

Зная эту мелодию и попытавшись спеть ее по партитуре, можно заметить, что акцент следует делать на второй ноте. Вероятно, петь ее, делая акцент на первой ноте, будет трудно. Но, как правило, акцент делается именно на первой ноте в каждом такте. Это означает, что если бы Джуди Гарланд дали приведенную выше партитуру вместо первоначальной, она бы неверно предположила, что акцент следует делать на первой ноте. Обе эти интерпретации данного музыкального предложения отражены в разных деревьях абстрактного анализа, как показано на рис. 9.1. В первой интерпретации первые восемь нот сгруппированы в первом такте, а последняя нота находится во втором такте. Во второй же интерпретации первая нота находится в первом такте, а остальные восемь нот сгруппированы во втором такте.

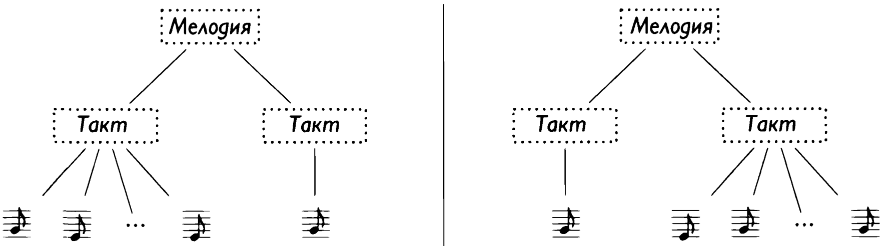


Рис. 9.1. Неоднозначность в грамматике может стать причиной наличия разных абстрактных синтаксических деревьев у одного предложения. Такие деревья представляют разную иерархическую структуру одного и того же предложения. В общем случае определить смысловое значение такого предложения невозможно, поскольку на него оказывает влияние его структура

Оба рассматриваемых здесь синтаксических дерева можно вывести, если видоизменить грамматику, исключив из нее символы тактовых черт. Отличие этих абстрактных синтаксических деревьев возникает из-за выбора разных правил для расширения нетерминала первого такта. Дерево, приведенное на рис. 9.1, *слева*, получается в результате его расширения до восьми нетерминалов нот, тогда как дерево, приведенное на рис. 9.1, *справа*, — в результате расширения до одного нетерминала ноты. Оба дерева представляют одну и ту же последовательность нот, но их внутренние структуры различаются. В дереве слева акцент делается на первой ноте, тогда как в дереве справа — на второй ноте. Произвести правильно синтаксический анализ такой последовательность фактически не представляется возможным, поскольку общее время звучания девяти нот составит $1\frac{1}{8}$, что не помещается в один такт.

Чтобы правильно выразить музыкальное предложение, придется в каком-нибудь его месте расположить тактовую черту с целью разделить ноты на два такта. И правильнее расположить эту тактовую черту после первой ноты. Благодаря этому акцент будет сделан на второй ноте.

Еще одним примером неоднозначности служит эпизод из фильма *Волшебник страны Оз*, в котором злая ведьма пишет своей метлой на небе сообщение на английском языке *Surrender Dorothy*, когда Дороти находится со своими друзьями в Изумрудном городе. Это сообщение может означать призыв к обитателям Изумрудного города сдать Дороти или же сдаться самой Дороти. В последнем случае между словами "Surrender" и "Dorothy" следует поставить запятую. Подобно тому как тактовые черты в нотной записи помогают прояснить правильную структуру и акцент мелодий, запятые и точки служат той же самой цели в письменных естественных языках, а знаки препинания и ключевые слова — в языках программирования.

Иногда в языках имеются конструкции, оставляющие некоторый выбор интерпретатору алгоритма в отношении тех шагов, которые требуется предпринять.

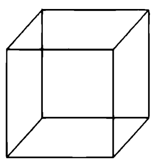
Например, приведенная здесь запись музыкальной импровизации дает музыканту возможность выбрать высоту тона для второй, третьей и четвертой нот. Алгоритм, в котором используется подобная конструкция, называется ***недетерминированным.*** В нотной записи недетерминированность иногда применяется для того, чтобы предоставить свободное место для музыкальной импровизации, т.е. больше свободы музыкантам в их интерпретации и исполнении музыкального произведения. Но недетерминированность не следует путать с неоднозначностью. С алгоритмической точки зрения музыкальная партитура снабжает музыканта инструкциями для воспроизведения последовательности звуков, каждый из которых имеет свою особую высоту тона и длительность. Неоднозначная же запись оставит музыканта в раздумье, что же следует делать. Таким образом, если недетерминированность является ***свойством*** языка, то неоднозначность — ***логической ошибкой*** в определении синтаксиса языка. Как ни странно, алгоритмы, оставляющие некоторый выбор открытым, довольно распространены. Например, в простом алгоритме поиска Гензелем и Гретель обратного пути домой камень, который должен быть выбран следующим, однозначно не указывается.

Неоднозначность — распространенное явление в языках. Она напоминает нам, что язык не только служит собранием правильно построенных предложений, но и используется для преобразования этих предложений в их структуру, представленную в форме абстрактных синтаксических деревьев. Неоднозначность может быть забавной; иногда она применяется в музыке для создания искусных эффектов. Например, мелодию можно начать так, чтобы вызвать у слушателя

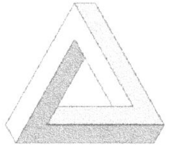
какое-нибудь одно восприятие, а затем добавить другой ритм, вызывающий удивление и вынуждающий слушателя воспринимать ее по-другому.

Такой эффект в рассматриваемом здесь примере можно сымитировать следующим образом. (Это лучше всего сделать двум людям. Если имеется клавиатура, то же самое может сделать и один человек, хотя эффект получится не таким сильным.) Один человек начинает играть или напевать, но не петь во весь голос, поскольку это слишком трудно, чередуя восьмые ноты "ми" и "соль" с (неверным) акцентом на нотах "соль". А через некоторое время вступает второй человек, начиная напевать четвертые ноты (например, "до") вместе с каждой нотой "ми", напеваемой первым человеком. После нескольких повторов получится смещение ритма, и мелодия неожиданно зазвучит как хорошо известный отрывок из песни *Над радугой.*

Неоднозначность возникает и в других обозначениях. Вам, вероятно, приходилось видеть куб Неккера. Если смотреть на его изображение достаточно долго, то его восприятие сменяется между кубом, передняя грань которого находится левее и ниже задней, и кубом, передняя грань которого находится правее и выше задней. И в этом случае у одного визуального представления имеются две разные интерпретации. Такая неоднозначность имеет значение. Допустим, что требуется коснуться правого верхнего угла на передней стороне куба. В таком случае достигаемое вами положение зависит от вашей интерпретации рисунка куба.



В связи с этим стоит упомянуть и о таких фигурах, как треугольник Пенроуза, навеянный рисунками нидерландского художника-графика М.К. Эшера (M.C. Escher). Но особенность этих фигур состоит не в неоднозначности их разных возможных интерпретаций, а скорее в *отсутствии* какой-нибудь интерпретации, согласующейся с физической реальностью, воспринимаемой людьми (см. главу 12, "Своевременный стежок вычисляется впрок").



Неоднозначность — занимательное понятие. Она служит неисчерпаемым источником для юмора и игры слов в естественных языках, как, например, "Казнить нельзя помиловать". Но в то же время неоднозначность вызывает серьезные трудности в алгоритмических языках, поскольку она способна воспрепятствовать ясному представлению предполагаемого смыслового значения отдельного предложения. Существует тонкая взаимозависимость между синтаксисом языка и его смысловым значением. С одной стороны, приходится тщательно анализировать синтаксис языка, чтобы определить его смысловое значение. С другой стороны, необходимо понять смысловое значение, прежде чем определить синтаксис надлежащим образом.

Обретение смыслового значения

Пример неоднозначности демонстрирует, что, не зная структуру предложения, невозможно понять его правильное смысловое значение. Но что же такое смысловое значение предложения?

Такие естественные языки, как английский, служат для общения на любую тему, и поэтому очень трудно сузить область их смыслового значения, помимо высказывания обо всем, о чем вообще можно говорить. Для музыкальных языков это формулируется намного проще: смысловым значением предложения, т.е. музыкального произведения, является звук, который можно услышать при воспроизведении данного музыкального произведения. Прежде чем углубляться в вопросы, касающиеся смыслового значения языка и порядка его определения, следует также подчеркнуть два разных значения термина *смысловое значение*. С одной стороны, имеется смысловое значение отдельного предложения. А с другой стороны, имеется смысловое значение языка. Чтобы различить оба эти понятия, здесь главным образом употребляется термин *семантика*, когда речь идет о языках, а термин *смысловое значение* — для ссылки на отдельные предложения.

Общее схематическое представление взаимосвязи предложений, языков и смыслового значения приведено на рис. 9.2. По существу, семантика языка задается смысловым значением всех его предложений, а смысловое значение отдельного предложения задается его связыванием со значением предметной области, обсуждаемой на данном языке.

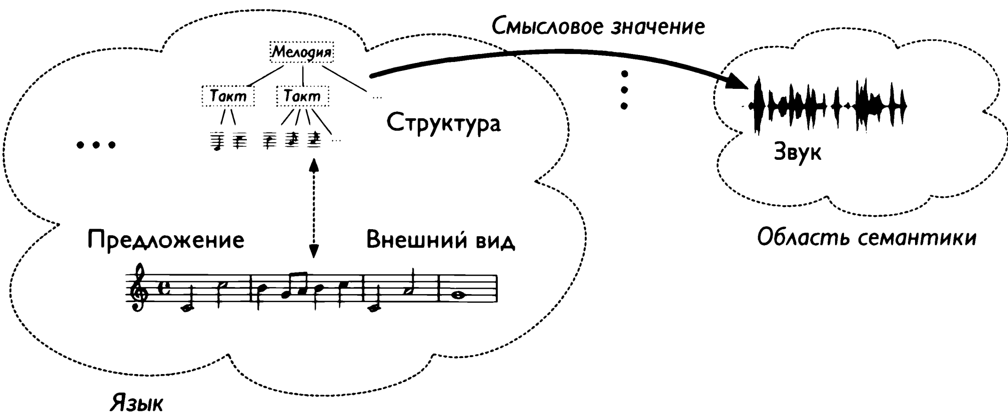


Рис. 9.2. Семантика языка задается отображением, связывающим структуру каждого предложения, представленного его абстрактным синтаксическим деревом, с элементом из области семантики. Такое представление о смысловом значении называется денотационной семантикой, поскольку основывается на присваивании денотаций (т.е. означиваний) предложениям языка

Смысловым значением одного конкретного предложения в музыкальном языке является сама музыка, которую можно услышать, когда кто-нибудь ее исполняет. Но поскольку разные музыканты по-разному интерпретируют исполняемую мелодию (одни ее поют, другие играют на музыкальных инструментах, а третьи варьируют гармонизацию или скорость воспроизведения), то мелодия, по-видимому, не будет звучать одинаково, как это определяет смысловое значение данной мелодии. Это затруднение можно преодолеть, поставив условие, чтобы музыка исполнялась одним музыкантом на конкретном музыкальном инструменте или синтезаторе по стандарту MIDI. С другой стороны, можно сказать, что смысловое значение мелодии *Над радугой* задается множеством всевозможных звучаний при ее надлежащем исполнении. В связи с этим, конечно, возникает следующий вопрос: что же считать надлежащим исполнением? Чтобы не вдаваться во всестороннее обсуждение этого противоречивого вопроса, обратимся к словам Поттера Стюарта (Potter Stewart), бывшего члена Верховного суда США, который на вопрос о его критерии непристойности, как известно, ответил так: "Я узна*ю* ее, когда вижу". Так, если вам приходилось слышать первоначальную версию песни *Над радугой* из фильма *Волшебник страны Оз*, вы можете судить о надлежащем ее исполнении, когда слышите ее. Ведь музыка — это искусство, и нас не должно удивлять или волновать, что ее нельзя охватить полностью формальными определениями. В этой связи важно отметить, что смысловым значением партитуры *Над радугой* является звук, который распознает как ее исполнение всякий, кому эта мелодия хорошо известна.

Если теперь взять все звуки, которые могут возникнуть в результате исполнения любой воображаемой музыкальной партитуры, то в конечном счете получится так называемая *область семантики*. Для такого языка, как нотный стан или табулатура, областью семантики является множество всех смысловых значений, которые может иметь любое предложение языка. Область семантики, безусловно, является крупным множеством, но в него *не* входит немало таких элементов, как, например, автомашины, животные, мысли, фильмы, правила дорожного движения и т.д. Следовательно, это множество по-прежнему остается полезным для определения характеристик языка — оно описывает то, что пользователь языка ожидает найти в обозначении предложения данного языка.

Собрание всех отдельных предложений языка и связанных с ними смысловых значений составляет семантику языка. Если язык однозначен, то каждое из его предложений имеет лишь одно смысловое значение [1] и поэтому выбор предложения из языка всегда будет приводить к его смысловому значению. В неоднозначном языке то обстоятельство, что одно предложение может иметь не одно, а несколько смысловых значений, отражается в наличии у него нескольких синтаксических деревьев, каждое из которых может иметь разное смысловое значение. Описанное представление смыслового значения называется *денотационной*

семантикой и основывается на принципе присваивания смысловых значений предложениям. В информатике имеется ряд других подходов к определению семантики языков, но денотационная семантика, вероятно, ближе всего к нашей интуиции, и поэтому ее легче понять, чем некоторые ее альтернативы.

Основное назначение языка состоит в передаче смыслового значения, но изобретенные до сих пор языки не обладают изначально очевидной семантикой. Следовательно, для того чтобы язык приносил пользу, ему должна быть назначена семантика. Но поскольку большинство языков состоят из бесконечного числа предложений, перечислить все предложения и их смысловые значения просто невозможно. Поэтому должен быть какой-то систематический способ определения семантики языка. Этот вопрос в конечном счете сводится к поиску алгоритма для определения смыслового значения отдельных предложений. Например, алгоритмическое определение денотационной семантики для языка состоит из двух частей. В первой части выполняется преобразование терминальных символов в основные элементы области семантики. В рассматриваемом здесь примере это означает преобразование отдельных нот в звуки конкретной частоты и длительности. А во второй части задаются правила, которые определяют для каждого нетерминала порядок построения его смыслового значения из смысловых значений его потомков в синтаксическом дереве. В данном примере имеются три нетерминала, а правило для *ноты* оказывается простым, поскольку у нетерминала каждой ноты имеется лишь один потомок. Это означает, что нота должна звучать так же, как и ее нота-потомок. Смысловое значение *такта* получается в результате соединения звуков его потомков в том порядке, в каком они появляются. И наконец, смысловое значение *мелодии* получается таким же образом, как и *такта*, т.е. в результате соединения звуков, получаемых как смысловое значение отдельных тактов мелодии.

В данном примере наглядно показано, каким образом смысловое значение предложения систематически составляется через его абстрактное синтаксическое дерево путем объединения сначала смыслового значения листьев в смысловые значения их родителей, затем — их смысловых значений и так далее до тех пор, пока не будет получено смысловое значение корня дерева. Определение семантики такого вида называется ***композиционным.*** Композиционное определение привлекательно тем, что отражает порядок грамматического определения синтаксиса бесконечного языка через конечный ряд правил. Язык, семантика которого может быть определена композиционным способом, также называется ***композиционным.*** Принцип композиционности был установлен немецким математиком и философом Готтлобом Фреге (Gottlob Frege) в ходе исследования языков и способов формального определения их семантики. Чтобы получить конечное определение смыслового значения для бесконечного числа грамматически задаваемых предложений, требуется некоторая степень композиционности. Если же языку недостает

композиционности, то, как правило, определить его семантику трудно, поскольку смысловое значение произвольного предложения нельзя получить, составив смысловые значения его частей, но оно должно быть описано отдельно. Такие описания являются исключениями, отменяющими общие правила для определения смыслового значения.

Описываемый здесь упрощенный музыкальный язык является композиционным, тогда как многие другие языки таковыми не являются или же являются композиционными частично. Примером тому служит английский язык, как, впрочем, и многие другие естественные языки. Чтобы найти самые простые примеры некомпозиционности, не следует искать дальше составных слов. Если огнетушитель — это приспособление для тушения огня, то хот-дог означает не горячую собаку, а горячую сосиску; красный петух — это пожар, а не домашняя птица красного цвета. Другими примерами могут служить идиоматические выражения вроде "отбросить коньки" или "врезать дуба", смысловое значение которых получается отнюдь не из составляющих эти выражения слов.

Нотный стан также состоит из некомпозиционных элементов. Например, лига — это символ, связывающий две последовательные ноты с одинаковой высотой тона (как правило, последнюю ноту одного такта с первого нотой следующего такта). Именно так происходит в самом конце мелодии *Над радугой*, где находится фраза "Почему я не могу?", где "могу" поется на одной ноте, которая длится в течение двух тактов. По правилу поиска смыслового значения мелодии сначала следует определить смысловые значения отдельных тактов, а затем соединить их. Это приведет к двум звукам продолжительностью один такт вместо двух тактов. Следовательно, чтобы получить надлежащим образом смысловое значение нот,

связанных в два или больше тактов, правило поиска смыслового значения мелодии должно быть замещено другим правилом, трактующим группы тактов, ноты которых связаны вместе.

Правила определения смыслового значения отдельного предложения необычайно похожи на правила, которые соблюдаются музыкантами, когда они интерпретируют нотную запись. И это не случайно, поскольку денотационная семантика языка служит алгоритмом для вычисления смыслового значения отдельного предложения из данного языка. Компьютер, понимающий язык, на котором задана денотационная семантика, способен выполнить эту семантику, а значит, вычислить смысловое значение предложений. Компьютер или алгоритм, способный выполнить семантику, называется *интерпретатором*. Если представить Гензеля и Гретель интерпретаторами инструкций для следования по камешкам, это может показаться не совсем обычным. А если назвать Джуди Гарланд интерпретатором музыки Харольда Арлена, то никаких возражений не возникнет. Так же как алгоритм обеспечивает повторяемость вычислений, определение языка может быть использовано для создания компьютеров, предназначенных для выполнения предложений языка, делая, таким образом, повторно выполняемым сам язык. Если обратиться к примеру нотной записи, то всякий, понимающий ее семантику, может научиться интерпретировать эту нотную запись. Иными словами, семантика может быть использована для обучения музыкантов без помощи учителя. Это дает людям возможность учиться самостоятельно.

В связи с интерпретаторами возникают следующие вопросы: что произойдет, если мы создадим запись исполнения музыки. Воплощают ли записи каким-то образом смысловое значение музыкального произведения? Не совсем, поскольку музыкальная запись воспроизводит лишь другое представление музыки. Первые записи мелодии *Над радугой* были сделаны на аналоговых носителях в виде пластинок с бороздами, по которым ходила игла проигрывателя, а ее движения преобразовывались в звуковые волны. Позднее звуковые волны стали преобразовывать в цифровой код, записываемый на компакт-диск в виде последовательности битов (единиц и нулей), которые затем считываются лазером и интерпретируются цифро-аналоговым преобразователем для воспроизведения звуковых волн. А ныне б*о*льшая часть музыки представлена в программных форматах вроде MP3, специально предназначенных для поддержки потоковой передачи музыкальных записей через Интернет. Но, каким бы ни было конкретное представление музыки, оно по-прежнему представляет ее в некоторой форме или на определенном языке, требующем исполнителя или компьютер для исполнения инструкций на этом языке с целью добиться предполагаемого звукового эффекта. Таким образом, запись и воспроизведение музыкального произведения фактически являются формой перевода с одного языка на другой, например из нотного стана в цифровое представление звуковых волн.

Дальнейшее исследование

Мелодия *Над радугой* была использована ранее с целью продемонстрировать, что каждое предложение языка обладает структурой, которая очень важна для понимания его смыслового значения. Для исследования структуры мелодий и потенциальной неоднозначности из записи можно было бы воспользоваться любым другим музыкальным произведением, при условии, что оно может быть описано в некоторой достаточно выразительной нотной записи. Чтобы понять значение нот, тактовых черт и прочих структурных элементов нотной записи, целесообразно исследовать альтернативные системы записи музыки и уяснить их ограничения. В частности, систематическое ограничение существующих видов записи приводит к появлению новых языков. Так, если наложить ограничение на высоту тона нот, то в конечном итоге получится *запись для ударных инструментов*, а если вообще пренебречь высотой тона — язык ритмов. Сократив далее длительности нот лишь до двух (короткой и длинной), можно получить *азбуку Морзе*. Пренебрегая мелодией, *аккордные символы* демонстрируют только гармонический ряд музыкального произведения, тогда как *схемы аккордов* объединяют аккордовые символы и ритмическую запись, чтобы продемонстрировать гармонический и ритмический ряды музыкального произведения.

Нотный стан более нагляден, чем обычный текстовый язык, хотя предложение (т.е. музыкальное произведение) по-прежнему является линейной последовательностью нотных символов, которые можно приспосабливать, изменяя их положение и внешний вид. По этому принципу формируется основание для многих видов записи, которые не являются традиционными текстовыми языками. Например, на универсальном графическом языке *Bliss* предложение состоит из последовательностей слов, которые являются символами, могущими быть, в свою очередь, составленными из более простых символов. Это похоже на часть *египетских иероглифов*, символы которых можно по-разному объединять в зависимости от их размеров и выражаемого намерения. Bliss и египетские иероглифы относятся к категории общих языков, предназначенных для выражения произвольных мыслей. Нотный же стан имеет узкую область применения, и поэтому он намного более простой. Характерным примером записи специального назначения служат *химические формулы*, описывающие строение молекул из атомов (например,

формула воды H_2O означает, что каждая молекула воды состоит из одного атома кислорода и двух атомов водорода).

Тем не менее приведенные выше примеры демонстрируют главным образом линейные виды записи, т.е. такие языки, в которых предложение состоит из последовательности символов. Визуальные языки могут быть выразительными, если в них используется двумерная топология. Например, химические формулы представляют лишь пропорции атомов, составляющих молекулу, игнорируя пространственное расположение атомов. А *структурные формулы* обеспечивают описание геометрического расположения атомов. *Диаграммы Фейнмана* являются двумерным языком, применяемым в физике для описания поведения субатомных частиц.

Во всех этих языках употребляется статическое одно- и двухмерное представление. Это означает, что можно взять изображение предложения и отправить его кому-нибудь, способному его затем интерпретировать. Другие языки преодолевают это ограничение на мгновенные снимки, используя также временн*ое* измерение. Например, язык образуют *жесты*. Движения тела нельзя в достаточной степени охватить одним жестом. Для определения языка движений тела требуются, как правило, видеозаписи или последовательности изображений. В пользовательских интерфейсах настольных и планшетных компьютеров, а также сотовых телефонов такие движения рук, как проведение пальцем и щипок, являются выполняемыми действиями. Помимо жестов, имеются виды записи для описания танцев, например *лабанотация*. Предложение в таком языке является алгоритмом, выполняемым танцовщиком, в результате чего получается танец как вычисление. Подобного рода языком служат *инструкции по оригами*, описывающие алгоритмы для складывания объектов из бумаги. Язык не является прерогативой людей. Например, в серии детских книг о приключениях доктора Дулиттла животные общаются движениями носа, ушей и хвоста. Но даже за пределами вымысла можно обнаружить, например, *танец пчел*, сообщающий о местоположении запасов пищи.

Смысловое значение языка определяется правилами, переводящими абстрактный синтаксис предложения в значения из области семантики. Общение может оказаться успешным только в том случае, если его участники согласуют эти правила. В книгах *Алиса в Стране чудес* и *Алиса в Зазеркалье* Льюиса Кэрролла наглядно показано, что происходит, если правила нарушаются или по-разному интерпретируются теми, кто принимает участие в общении.

Управляющие структуры и циклы

День Сурка

Сила привычки

Вернувшись в свой офис, вы первым делом отправляете несколько писем. С этой целью вы, не задумываясь, складываете дважды каждое письмо по горизонтали, чтобы сократить его размер по высоте приблизительно в три раза, а затем вкладываете его в конверт. Таким приемом вы пользуетесь постоянно, поскольку бумага и конверт, которые вы выбрали, имеют вполне определенный формат, а вы уже давно убедились, что именно такая схема складывания письма позволяет уместить его в конверт.

Процесс складывания писем является вычислением, описываемым алгоритмом, элементарным примером которого служит оригами. Но несмотря на всю свою простоту, алгоритм складывания бумажных писем наглядно показывает целый ряд моментов, касающихся языков.

Прежде всего, алгоритм может быть описан двумя несколько различающимися способами. Если *складывание* означает сложить лист бумаги, отступив на конкретное расстояние от его края (например, на треть всей длины листа бумаги), то можно сказать, что его требуется "сложить и сложить" или "сложить дважды". На первый взгляд, в этих инструкциях нет особых различий, но на самом деле они совсем по-разному описывают повторяющееся действие. В первом случае повторяющееся действие явно перечисляется столько раз, сколько требуется, тогда как во втором случае лишь говорится, как часто это действие следует повторить. Различие становится более ясным, когда приходится отправлять лист бумаги крупного формата и для его складывания требуется предпринять дополнительные шаги. Например, чтобы сложить лист бумаги три раза, соответствующую инструкцию можно составить двумя способами: "сложить, сложить и сложить" или "сложить три раза". Если расширить данный пример до 500 повторений одного и того же действия, то станет сразу ясно, какой из этих способов окажется более практичным.

Первый способ описания действия означает *последовательную композицию* из отдельных шагов, тогда как второй способ означает *цикл*. И то, и другое служит примером *управляющих структур*, являющихся компонентами алгоритма, которые не выполняют фактически никакого алгоритмического действия, а только организуют действие других шагов алгоритма. Это можно сравнить с функциями работников и руководителей на фабрике. Руководители ничего не производят непосредственно, но координируют действия работников, производящих нечто конкретное. Управляющие структуры вообще являются важнейшими

составляющими любого языка для описания алгоритмов, а циклы (или рекурсия), в частности, используются в большинстве нетривиальных алгоритмов.

На самом деле имеется третий алгоритм складывания листа бумаги со следующим описанием: "сложить бумагу по размеру". В нем также используется циклическая управляющая структура, но данный цикл отличается от предыдущего одной важной особенностью. В нем не указано явно, сколько раз следует повторить описываемое действие, а лишь определяется условие, которое должно быть выполнено, чтобы завершить повторение данного действия. Фактически этот алгоритм оказывается более общим, чем два предыдущих алгоритма, поскольку он подходит для бумаги любого формата, а следовательно, позволяет решить более общую задачу. Без цикла (или рекурсии) выразить этот алгоритм было бы невозможно. А цикл с фиксированным числом повторений можно всегда переписать как последовательность такого же количества шагов.

Допустим, что вам требуется отправить не одно письмо, а целую пачку документов. Всякий раз, когда вам приходится иметь дело с документами на более чем пяти листах бумаги, вы выбираете более крупный конверт, в который документы помещаются стопкой без складывания, вместо того чтобы пытаться сложить и впихнуть все эти листы в небольшой конверт. Принятие решения складывать или складывать документы служит примером еще одной управляющей структуры, называемой *условной конструкцией* и выполняющей одно из двух возможных действий в зависимости от заданного *условия*. В данном примере условие следующее: количество листов бумаги равно или меньше пяти. В первом случае листы бумаги *не* складываются и вкладываются в большой конверт, а во втором случае они складываются и вкладываются в маленький конверт.

Условие требуется для того, чтобы определить момент, когда следует завершить повторение цикла. Так, условие для завершения цикла "сложить три раза" выражается с помощью счетчика, достигающего конкретного значения, т.е. 3. А в условии цикла "сложить по размеру" проверяется свойство бумаги (ее формат), которое преобразуется действиями в данном цикле. Этот последний вид условия оказывается более эффективным, о чем свидетельствует тот факт, что оно выражает более общий алгоритм. Но эта повышенная *выразительность* не дается даром. Если завершение цикла на основе счетчика нетрудно обнаружить, поскольку счетчик не зависит от действий, выполняемых в цикле, то завершение более общего цикла не столь очевидно. И в этом случае нельзя с уверенностью сказать, остановится ли вообще алгоритм, в котором применяется такая разновидность цикла (см. главу 11, "Счастливый конец не гарантируется").

Б*о*льшую часть того, что мы делаем, приходится повторять неоднократно, и зачастую такие действия могут быть описаны с помощью циклов. Иногда даже кажется, что повторяются целые дни. Такое представление доведено до крайности в фильме *День Сурка*, сюжет которого послужит нам руководящим примером для исследования циклов и других управляющих структур в главах 10, "Намылить, смыть, повторить", и 11, "Счастливый конец не гарантируется".

10 Намылить, смыть, повторить

Любой алгоритм задается на каком-то языке, семантика которого определяет вычисление, представленное этим алгоритмом. Ранее было показано, что разные языки имеют разную семантику и предназначены для разных компьютеров. Так, алгоритм, выражаемый на языке программирования, выполняется на электронной вычислительной машине и, как правило, обозначает вычисление данных. С другой стороны, алгоритм, выражаемый на языке нотной записи, исполняется музыкантом и обозначает звуки. Но несмотря на эти различия большинство нетривиальных языков разделяют следующее общее для них занимательное свойство — состоят из двух видов инструкций: *операций*, имеющих прямое действие, а также *управляющих структур*, организующих порядок, применение и повторение операций.

Управляющие структуры не только играют решающую роль в формулировании алгоритмов, но и определяют выразительность языков. Это означает, что от определения управляющих структур и решения включить конкретные их разновидности в язык зависит, какие именно алгоритмы могут быть выражены на языке, а следовательно, какие задачи они позволяют решать.

Одной из таких управляющих структур является так называемый *цикл*, позволяющий описывать повторяющиеся действия. Мы уже не раз приводили примеры циклов, включая алгоритм поиска Гензелем и Гретель обратного пути домой, когда им приходилось неоднократно искать очередной камешек, который они еще не посещали, или сортировку выбором, действующую по принципу многократного поиска наименьшего элемента в списке. И даже нотная запись содержит

управляющие структуры для выражения циклов. Но, несмотря на то что циклы широко использовались в представленных ранее примерах, мы еще не обсуждали их подробно. Следуя примеру синоптика Фила Коннорса, снова и снова переживающего День Сурка, мы поясним, каким образом действуют циклы и другие управляющие структуры. В этой главе будут рассмотрены способы описания циклов и те различия в вычислениях, которые они способны выразить.

Вечность и один день

Вам, вероятно, приходилось слышать о Дне Сурка — обычае, согласно которому по поведению сурка можно предсказать раннюю весну или еще целых шесть недель зимы. Конкретнее, ежегодно во второй день февраля сурок просыпается от зимней спячки и выходит из своей норки. Если в солнечный день он замечает свою тень, то возвращается в свою норку, и это указывает на то, что зима продлится еще шесть недель. А если в пасмурный день он не может найти свою тень, это означает ранний приход весны.

Я вырос за пределами Соединенных Штатов и поэтому узнал об этом обычае из вышедшего на экраны в 1993 году фильма *День Сурка*, главный герой которого, Фил Коннорс (Phil Connors) — поначалу высокомерный и циничный синоптик из города Питтсбурга — сообщает о торжествах по случаю Дня Сурка в небольшом городке Панксатони. Повествуемая в этом фильме история занимательна тем, что Филу Коннорсу приходится снова и снова проживать один и тот же день. Каждое утро он встает в 6 часов утра под одну и ту же мелодию по радио и вынужден переживать один и тот же последовательный ряд ситуаций. Сюжет этого фильма разворачивается по мере того, как главный его герой по-разному реагирует на эти ситуации.

Повторение играет важную роль в жизни каждого из нас. Учиться чему-нибудь имеет смысл лишь тогда, когда можно надеяться, что это пригодится в будущем. В более широком смысле любой прошлый опыт можно применить лишь в обстоятельствах, сходных с теми, при которых он был приобретен. Ежедневно мы повторяем немало действий: встаем, одеваемся, завтракаем, отправляемся на работу и т.д. Действие или ряд действий, которые непосредственно повторяются несколько раз, называются *циклом*. Повторяемое действие называется телом *цикла*, а каждое выполнение тела цикла — его *итерацией*, т.е. повторением. Разговаривая с приятелями в баре, Фил Коннорс размышляет над своим положением:

Фил: *Что бы ты делал, если бы торчал на одном месте, и каждый день был бы точно таким же, как и предыдущий, и ты ничего не мог бы с этим поделать?*

Ральф: *Ты рассказываешь историю моей жизни…*

Этот диалог можно подытожить в следующем описании чьего-то образа жизни в виде цикла:

repeat *повседневный распорядок дня*
until *придет смерть*[1]

В повседневном распорядке дня Фила Коннорса бесит, а зрителя фильма потешает, что каждый день в его жизни все происходит точно так же, как и в предыдущий, — все, что не относится к непосредственной реакции на его действия в течение дня. Ведь иначе смотреть этот фильм сразу же стало бы скучно, как и слушать монотонное повторение припева в конце песни.

Дни нашей жизни, как правило, приносят меньше разочарования, чем вечно повторяющийся День Сурка, поскольку наши вчерашние действия влияют на то, что происходит сегодня. Следовательно, каждый день, каким бы повторяющимся он ни казался, происходит в другом контексте, а понимание того, что мы и другие делаем, имеет большое значение, поскольку дает нам ощущение непрерывного изменения и продвижения. Подобные наблюдения предполагают различение двух разновидностей циклов: тех, которые производят на каждой итерации один и тот же результат, и тех, которые производят разные результаты. Примером первой разновидности служит цикл, в котором выводится название города, в котором вы родились (например, Нью-Йорк). Если не переименовать город, то в этом цикле будет произведен постоянный поток одних и тех же выводимых результатов: "Нью-Йорк", "Нью-Йорк", "Нью-Йорк"... А в цикле для вывода сводки погоды, в которой сообщается об ожидаемых осадках в виде дождя, будет произведен поток, состоящий из утвердительных и отрицательных прогнозов, если только вы не живете в Черапунджи [1], где, вероятнее всего, будет получен постоянный поток из одних лишь утвердительных прогнозов по поводу дождя.

Следует, однако, иметь в виду, что даже в цикле, производящем разные результаты, его тело остается тем же самым на каждой итерации, а отличие в результатах достигается благодаря переменным. *Переменная* — это имя, указывающее на часть окружающего мира. Через имя переменной алгоритм может наблюдать за окружающим миром и манипулировать им. Например, переменная `weather` указывает на текущие погодные условия, скажем, солнечный день, тогда как алгоритм, проверяющий текущие погодные условия, получает соответствующее значение через доступ к данной переменной. Изменить значение переменной `weather` в алгоритме нельзя, но переменная `pebble`, обозначающая следующий камешек, к которому должны направиться Гензель и Гретель, изменяется с каждым посещаемым камешком. Аналогично в инструкции "Найти наименьший элемент в списке как

[1] Дословно: повторять повседневный распорядок дня, пока не придет смерть. Цикл `repeat until` выполняет действия, указанные после ключевого слова `repeat` до тех пор, пока не станет истинным условие после ключевого слова `until`. — *Примеч. ред.*

составной части сортировки выбором" переменная `smallest_element` изменяется на каждой итерации цикла (если только в списке не содержатся дубликаты).

Термин *цикл* иногда употребляется для описания как самого цикла (т.е. алгоритма), так и формируемого в нем вычисления, и поэтому самое время напомнить об отличии описания вычисления от его выполнения. Так, цикл ежедневной сводки погоды выглядит следующим образом:

repeat *сводка погоды* **until** *вечно*

Результатом выполнения этого цикла является последовательность значений, а это уже нечто совсем иное, чем само описание цикла. Например, выполнение данного алгоритма могло бы привести к такой последовательности:

дождь пасмурно солнечно солнечно гроза солнечно ...

Такой цикл сообщения о погоде описывает разновидность цикла, в котором оказывается сам Фил Коннорс: каждое утро он должен готовить сводку погоды по прогнозу своего коллеги по работе сурка Фила из Панксатони. (Любопытно, что, несмотря на все драматические отклонения от привычного распорядка дня — в какой-то момент Фил Коннорс даже пытается прикончить сурка Фила из Панксатони и покончить с самим Филом Коннорсом, — он, похоже, никогда не упускает случай составить сводку погоды по прогнозу сурка.) Безусловно, Фил Коннорс надеется, что цикл, в котором он находится, не будет повторяться бесконечно. В действительности он предполагает следующую форму этого цикла:

repeat *сводка погоды* **until** *'некоторое скрытое условие'*

Его первоначальные попытки выскользнуть из цикла, как из замкнутого круга, по существу, являются стремлением обнаружить скрытое условие, являющееся весьма важной составляющей любого цикла и называемое *условием завершения.* Ближе к концу истории Фила Коннорса мы узнаем условие завершения его цикла — когда он становится добрым человеком, заботящимся о людях и помогающим им. Его ежедневное перевоплощение дает ему возможность улучшить свою карму, которая служит ключом, чтобы покинуть, наконец, День Сурка как чистилище.

Цикл проживания Дня Сурка служит примером следующей общей схемы цикла [2]:

repeat *шаг* **until** *условие*

Условие завершения оказывается в самом конце данной схемы и вычисляется после выполнения тела цикла. Если это условие истинно, цикл завершается. В противном случае тело цикла будет выполнено снова, после чего условие его завершения снова будет проверено с целью выяснить, следует ли продолжить или завершить цикл, и т.д.

Очевидно, что условие завершения, оказывающееся ложным в один момент, может в дальнейшем стать истинным, если содержит переменную, которая может быть изменена действиями, выполняемыми в теле цикла. Например, попытки привести в движение автомашину, в которой закончилось топливо, не будут успешными до тех пор, пока она не будет заправлена. Так, если повторяющиеся действия включают в себя только поворот ключа зажигания, пинание шин или произнесение волшебного заклинания, то все это не поможет сдвинуть автомашину с места. Таким образом, условие завершения позволяет отличать *конечные циклы*, условие завершения которых в конечном итоге становится истинным, от *бесконечных циклов*, условие завершения которых всегда остается ложным.

Недаром говорится, что делать постоянно одно и то же, ожидая разных результатов, было бы безумием [3][2]. Следовательно, может возникнуть искушение заклеймить бесконечные циклы порозом безумия, но имеются случаи, для которых идеально подходят никогда не завершающиеся циклы. Характерным примером бесконечного цикла является веб-служба, принимающая запрос, обрабатывающая его, после чего переходящая к следующему запросу, и так до беспонечности. Но если цикл является частью алгоритма и, в частности, если после него следуют другие шаги алгоритма, то можно надеяться, что он в конечном итоге завершится. Ведь иначе последующие шаги алгоритма так и не будут выполнены, а сам алгоритм вообще не завершится.

Завершение является одним из самых важных свойств цикла. Условие завершения определяет, завершается ли цикл, но для завершения цикла крайне важно влияние его тела на окружающий мир и, в частности, на те его части, от которых зависит условие завершения. Сообщение прогноза погоды не особенно изменяет окружающий мир и вряд ли оказывает влияние на неизвестное условие завершения цикла Дня Сурка для Фила Коннорса. От безысходности он пытается предпринимать все более и более крайние действия, включая разные формы самоубийства и убийство сурка Фила из Панксатони, отчаянно надеясь воздействовать на окружающий мир таким образом, чтобы условие завершения, наконец-то, стало истинным.

Все под контролем

Когда День Сурка начинает повторяться, Фил Коннорс теряет контроль над своей жизнью. Напротив, он сам оказывается под контролем цикла Дня Сурка, болезненно переживая это обстоятельство. То, что он находится под контролем цикла, означает, что он живет в теле цикла, который управляет тем, когда завершится повторение и когда Фил Коннорс сможет вырваться из этого цикла.

---

[2] Здесь следовало бы уточнить понятие “одного и того же”, так как иначе оказывается “безумием” поглядывать на часы, дожидаясь некоторого момента времени. — *Примеч. ред.*

Цикл управляет частотой выполнения своего тела, но результат действия цикла получается только при выполнении шагов его тела. Иными словами, цикл оказывает не прямое, а косвенное действие через повторение шагов его тела. Поскольку в общем случае частота выполнения алгоритмических шагов имеет значение, цикл оказывает свое влияние через количество повторений выполнения его тела. А поскольку цикл управляет выполнениями своего тела (через условие завершения), то он называется *управляющей структурой.* Цикл — это управляющая структура для повторного выполнения ряда алгоритмических шагов. Двумя другими управляющими структурами являются последовательная композиция и условная конструкция.

Последовательная композиция соединяет два или больше шагов в упорядоченную последовательность. Для обозначения того обстоятельства, что два шага выполняются последовательно, а не параллельно, ранее употреблялся союз *и*, хотя лучше было употребить такие ключевые слова, как *а затем* или *вслед за.* Но ради простоты и краткости далее будет употребляться обозначение, принятое в большинстве языков программирования: точка с запятой, обозначающая соединение двух алгоритмических шагов. Такое обозначение подобно написанию списка элементов. Кроме того, оно краткое и не отвлекает от самих шагов алгоритма. Например, последовательная композиция *встать*; *позавтракать* означает следующее: сначала встать, а затем позавтракать. Порядок следования шагов, безусловно, имеет значение, хотя для некоторых людей последовательная композиция *позавтракать*; *встать* будет приятным отклонением от обычного распорядка дня по воскресеньям. Общая форма последовательной композиции выглядит следующим образом:

шаг ; *шаг*

Здесь *шаг* — это нетерминал, вместо которого подставляется любой простой или составной шаг. В частности, *шагом* может быть еще одна последовательная композиция других шагов. Так, если требуется вставить принятие душа между подъемом и завтраком, с этой целью можно воспользоваться последовательной композицией дважды, расширив первый *шаг* до последовательной композиции *встать*; *принять душ*, а второй *шаг* — до *позавтракать*, что вместе составит последовательную композицию *встать*; *принять душ*; *позавтракать* [4].

В упоминавшемся ранее примере складывания листа бумаги вдвое, инструкцию "сложить и сложить" можно теперь переписать как *сложить*; *сложить*, соединив знаком ; оба шага, которые совсем не обязательно должны быть разными. А выполнение цикла **repeat** *складывание* **until** *бумага подходит по размеру* (или **repeat** *складывание три раза*) приводит к такому же самому вычислению, как и последовательность *сложить*; *сложить*; *сложить*. И это наглядно показывает, что циклы служат удобным средством для описания последовательности действий.

Важный вклад цикла в организацию систематических вычислений состоит в том, что он производит произвольно длинные последовательности шагов, при том что повторяемые действия упоминаются в нем лишь один раз.

В *условной конструкции* один из двух или более шагов выбирается для выполнения в зависимости от справедливости заданного условия. Как и в цикле, условие используется в этой конструкции для принятия решения. Общая форма условной конструкции выглядит следующим образом:

if *условие* **then** *шаг* **else** *шаг* [3]

Всякий раз, когда сурка Фила из Панксатони просят предсказать погоду, он, по существу, выполняет следующий синоптический алгоритм:

if *солнечно* **then** *объявить продление зимы еще на шесть недель*
else *объявить приход ранней весны*

Условная конструкция — это управляющая структура, позволяющая делать в алгоритмах выбор между альтернативными шагами. Приведенная выше условная конструкция является частью ежегодного цикла для сурка Фила из Панксатони и ежедневного цикла для Фила Коннорса. Она наглядно показывает, что управляющие структуры можно объединять произвольным образом. Это означает, что условные конструкции можно включать как составные части в циклы или последовательные композиции, а циклы — как альтернативные варианты действий в условных конструкциях или же как составные части в последовательности шагов, и т.д.

Основные шаги алгоритма подобны ходам в игре (например, передаче мяча и забиванию гола в футболе или нападению фигурой и рокировке в шахматах). А управляющие структуры определяют стратегии в этих играх, например **repeat** *передать пас* **until** *мяч оказывается перед воротами* (а если вы умеете играть в футбол, как Лионель Месси, то **repeat** *дриблинг* **until** *мяч оказывается перед воротами*). Это означает, что управляющие структуры составляют крупные игры из простых движений и ходов.

Как было показано в главе 8, "Сквозь призму языка", для описания музыки имеются самые разные виды записи. Аналогично разные виды записи имеются и для алгоритмов. Каждый язык программирования служит примером определенного способа записи алгоритмов, и хотя языки могут существенно различаться набором предоставляемых управляющих структур, в большинстве из них в той или иной форме предоставляются цикл, условная конструкция и последовательная композиция [5]. Записью, наглядно показывающей различие управляющих структур, является *блок-схема.* На блок-схеме алгоритм обозначается

---

[3] **Если** *условие* **то** *шаг* **иначе** *шаг.* В случае истинности *условия* выполняется *шаг,* указанный после ключевого слова `then`; если *условие* ложно, выполняется *шаг,* указанный после ключевого слова `else`. — *Примеч. ред.*

прямоугольниками, соединяемыми стрелками. Основные действия в алгоритме указаны внутри прямоугольников, принимаемые решения заключены в ромбы, а стрелками обозначен ход вычисления. Для последовательной композиции это означает, что следовать нужно по стрелке от одного прямоугольника к другому. А в условных конструкциях и циклах, в которых условия обозначены двумя исходящими стрелками, выбор стрелки, по которой нужно следовать, зависит от заданного условия. Некоторые примеры записи управляющих структур в виде блок-схемы приведены на рис. 10.1.

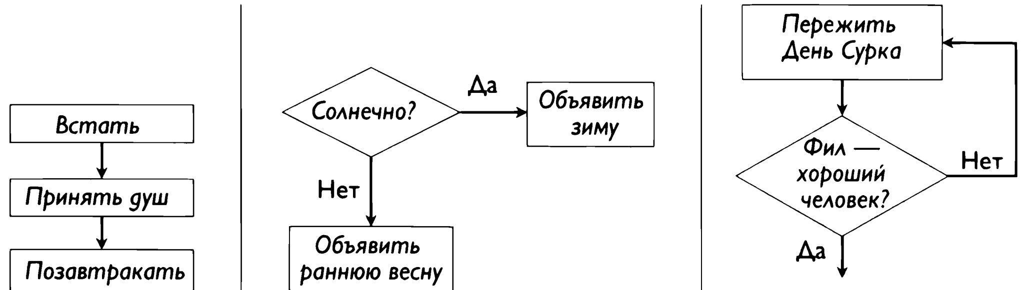


Рис. 10.1. Запись управляющих структур в виде блок-схем. Слева: последовательность шагов, предпринимаемых Филом Коннорсом каждое утро. Посредине: условная конструкция, показывающая решение, которое приходится принимать сурку Филу из Панксатони каждый раз, когда наступает День Сурка. Справа: цикл, выражающий течение жизни Фила Коннорса по ходу фильма День Сурка. *Стрелка с обозначением "Нет", ведущая к условию, замыкают два узла блок-схемы в цикл*

Поражает, насколько похожи записи условных конструкций и циклов. Обе они состоят из условия с двумя возможными продолжениями. Единственное существенное различие состоит в том, что путь с обозначением "Нет" ведет от условия, заданного в цикле, к шагу цикла и продолжается обратно к условию. Составленный таким образом цикл наглядно поясняет само название *цикл*, присвоенное данной управляющей структуре.

Блок-схемы являются *визуальным языком.* В отличие от текстового языка, представляющего алгоритм в виде линейной последовательности слов и символов, визуальный язык представляет символы в двух- или трехмерном пространстве, связанном пространственными отношениями. В блок-схемах применяются стрелки для выражения отношений типа "выполнить следующим" между действиями, представленными прямоугольниками. Блок-схема внешне похожа на транспортную сеть, в которой пересечения служат местами выполнения действий, а соединения ведут от одного действия к другому. С одной стороны, это можно сравнить с передвижением по парку развлечений, который служит алгоритмом для развлечения. Разные люди посещают отдельные аттракционы в различном порядке и разное число раз в зависимости от их склонностей и впечатлений от

аттракционов. С другой стороны, блок-схему можно сравнить с проходами в супермаркете, соединяющими разные секции и полки с товарами. В этом случае супермаркет можно рассматривать как алгоритм приобретения опыта покупок.

Блок-схемы были широко распространены в 1970-е годы, а в настоящее время они лишь иногда используются для составления документации на программное обеспечение, но крайне редко — в качестве записей для программирования. Это, в частности, объясняется тем, что такой вид записи не поддается масштабированию. Ведь даже блок-схемы умеренных размеров трудно читать, а из-за обилия стрелок их называют “макаронным кодом”. Кроме того, сходство в обозначении условной конструкции и цикла затрудняет их выявление и различение на блок-схемах, хотя такое обозначение очень удобно для того, чтобы наглядно показать взаимосвязь между этими управляющими структурами.

Представленные здесь управляющие структуры применяются в алгоритмах для отдельных компьютеров. Современные многоядерные микропроцессоры способны выполнять операции параллельно. Люди также могут выполнять операции параллельно, особенно если находятся в коллективе. Чтобы воспользоваться преимуществами параллелизма, алгоритму требуются управляющие структуры, специально предназначенные для этой цели. Например, можно написать инструкцию *гулять* || *жевать жвачку*, дать кому-нибудь приказ прогуляться, жуя одновременно жвачку. Это совсем иная инструкция, чем инструкция *гулять*; *жевать жвачку*, которая означает, что сначала следует погулять, а затем пожевать жвачку.

Параллельная композиция приносит пользу в том случае, если два результата, которые не зависят один от другого, требуются для иного вычисления. Например, Шерлок Холмс и доктор Ватсон нередко разделяют свою сыскную работу при расследовании преступления. Но с другой стороны, нельзя одновременно встать и принять душ. Оба эти действия должны быть выполнены в строгой последовательности, поскольку в данном случае имеет значение порядок их выполнения.

С параллельным вычислением тесно связано распределенное вычисление, которое происходит через обмен данными между взаимодействующими посредниками. Например, когда Фил Коннорс и его коллектив составляют сводку погоды по предсказаниям сурка, им требуется синхронизировать работу телекамеры с речью Фила Коннорса. Алгоритму для описания такой координации действий требуются свои управляющие структуры, в частности для отправки и приема сообщений.

В общем, в любом предметно-ориентированном языке, т.е. в языке для специальной области применения, могут быть свои управляющие структуры. Например, нотная запись содержит управляющие структуры для повторов и переходов, а языки составления рецептов — конструкции для выбора, позволяющие составлять разнообразные рецепты. Управляющие структуры служат связующим звеном для объединения примитивных операций в крупные алгоритмы, способные описывать важные вычисления.

Цикл за циклом

Пытаясь вырваться из замкнутого круга Дня Сурка, Фил Коннорс, по существу, пробует обнаружить свое условие завершения цикла. Это довольно необычный способ обращения с алгоритмами вообще и с циклами в частности. Обычно мы выражаем алгоритм и выполняем его, чтобы получить требуемое вычисление. Фил Коннорс же, напротив, являясь частью вычисления, не знает описывающий его алгоритм. В поисках действия, которое приведет к истинному условию завершения, он пытается реконструировать алгоритм.

Циклы и их завершение играют важную роль в вычислении. Безусловно, циклы (и рекурсия) относятся к числу самых важных управляющих структур, поскольку запускают вычисление. Без циклов вычисления можно описать лишь с помощью нескольких фиксированных шагов, что ограничивает вычисления и упускает из виду самое интересное в них.

Принимая во внимание важность циклов, нет ничего удивительного в том, что они могут быть описаны разными способами. Упоминавшаяся до сих пор схема цикла **repeat** *шаг* **until** *условие* называется ***циклом с повторением***. Такому циклу свойственно выполнять свое тело хотя бы один раз, каким бы ни было условие его завершения. С другой стороны, ***цикл с проверкой условия*** выполняет свое тело лишь в том случае, если его условие истинно, а следовательно, тело такого цикла может быть вообще не выполнено. Форма цикла с проверкой условия выглядит следующим образом:

while *условие* **do** *шаг*[4]

Несмотря на то что управление обеими упомянутыми выше разновидностями циклов осуществляется по заданному условию, оно играет разную роль в каждом из них. Если в цикле с повторением условие управляет выходом из цикла, то в цикле с проверкой условия оно управляет (первым или повторным) входом в цикл. Иными словами, если условие истинно, то цикл с повторением завершается, тогда как цикл с проверкой условия продолжается. А если условие ложно, то цикл с повторением продолжается, тогда как цикл с проверкой условия завершается [6]. Это отличие наглядно демонстрирует запись обоих циклов в виде блок-схемы, приведенная на рис. 10.2.

[4] **Пока** *условие* **выполнить** *шаг*. Цикл `while do` проверяет условие, указанное после ключевого слова `while`, и, если оно истинно, выполняет шаг, указанный после ключевого слова `do`, после чего вновь возвращается к проверке условия. В случае ложности условия цикл завершает работу. — *Примеч. ред.*

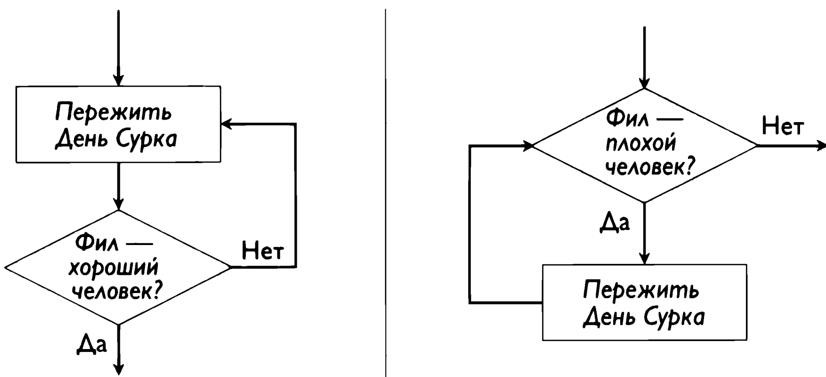


Рис. 10.2. Блок-схемы, демонстрирующие разное поведение циклов с повторением и с проверкой условия. Слева: блок-схема цикла с повторением. Справа: блок-схема цикла с проверкой условия

Несмотря на кажущееся разным поведение обеих разновидностей рассматриваемых здесь циклов, цикл с повторением можно записать, используя цикл с проверкой условия, и наоборот. Для этого достаточно обратить условие завершения цикла, т.е. преобразовать его таким образом, чтобы оно было истинным в тех случаях, когда исходное условие ложно. Например, условие завершения цикла с повторением Дня Сурка, которое состоит в том, чтобы "стать хорошим человеком", становится предусловием "не стать хорошим человеком" или "оставаться плохим человеком" для соответствующего цикла с проверкой условия. Но помимо этого, необходимо проявить особое внимание, чтобы обеспечить одинаковое число итераций в обоих циклах. Например, цикл с повторением дает Филу Коннорсу возможность пережить День Сурка хотя бы один раз, каким бы он ни был, тогда как цикл с проверкой условия позволяет сделать это лишь в том случае, если он плохой человек. А поскольку в данной истории это именно так, оба цикла ведут себя одинаково.

Языковое представление дает возможность более формально выразить равнозначность циклов с помощью следующего простого уравнения:

repeat *шаг* **until** *условие* = *шаг*; **while** *не* *условие* **do** *шаг*

В этом случае первоначальный *шаг* требуется сделать прежде цикла с проверкой условия, поскольку его тело может быть вообще не выполнено, в отличие от тела цикла с повторением, которое выполняется хотя бы один раз. Иногда отличие обоих циклов действительно имеет значение. Рассмотрим в качестве примера упоминавшийся ранее алгоритм Гензеля и Гретель. Его можно выразить в виде цикла с повторением следующим образом:

repeat *найти камешек* **until** *оказаться дома*

Недостаток этого алгоритма заключается в следующем: если бы Гензель и Гретель выполнили его, когда уже были дома, то цикл никогда бы не завершился, поскольку им не удалось бы найти камешек. Люди обычно не делают подобных глупостей, а вместо этого прерывают цикл. Но если строго придерживаться такого алгоритма, то вычисление так и не завершится.

Циклы можно описывать и с помощью рекурсии. Подробнее о рекурсии речь пойдет в главе 12, “Своевременный стежок вычисляется впрок”, но уяснить основную идею нетрудно (в действительности рекурсия уже рассматривалась в главе 6, “Сортировка алгоритмов сортировки”, когда речь шла об алгоритмах разбиения). Для рекурсивного описания алгоритма необходимо сначала присвоить ему имя, а затем воспользоваться этим именем в его собственном определении. Так, цикл Дня Сурка может быть рекурсивно описан следующим образом:

ДеньСурка = пережить день;
if *Фил — хороший человек* **then** *завершение* **else** *ДеньСурка*

Такое определение, по сути, имитирует цикл с повторением: как только день прожит, в условной конструкции проверяется условие завершения. Если оно ложно, вычисление просто завершается. В противном случае алгоритм выполняется снова. Рекурсивное выполнение алгоритма подобно непосредственному переходу в начало последовательности и запуску цикла на повторное выполнение.

Общим для всех рассмотренных до сих пор различных описаний циклов (с повторением, проверкой условия и рекурсией) является то обстоятельство, что их завершением управляет условие, которое повторно вычисляется до или после каждого выполнения тела цикла. Окончание цикла зависит от влияния его тела на условие завершения, чтобы оно в конечном счете стало истинным (в цикле с повторением) или ложным (в цикле с проверкой условия). Это означает, что заранее неизвестно, через сколько итераций пройдет цикл. И даже не ясно, завершится ли вообще любой подобный цикл. Такая неопределенность на самом деле является важной частью цикла Дня Сурка, переживаемого Филом Коннорсом.

Но из циклического описания некоторых вычислений заранее понятно, сколько раз должен быть выполнен цикл. Например, если задача состоит в том, чтобы вычислить квадрат первых десяти натуральных чисел, то очевидно, что такое вычисление может быть достигнуто в цикле, повторяющем операцию возведения в квадрат ровно десять раз. А если вспомнить алгоритм складывания листа бумаги по размеру конверта, то его можно описать с помощью цикла, выполняющегося ровно два раза. В таких случаях употребляются *циклы с фиксированным числом итераций*, или просто циклы `for`, имеющие следующую общую форму [7]:

for *число* **times do** *шаг*[5]

[5] Дословно — **для** *число* **раз выполнить** *шаг*. — *Примеч. ред.*

По этой схеме цикл складывания листа бумаги можно выразить следующим образом:

for *2* **times do** *сложить лист*

Преимущество цикла `for` заключается в том, что еще до его выполнения совершенно ясно, сколько итераций будет в нем выполнено. Совсем иначе дело обстоит с другими циклами, поскольку число их итераций выясняется только при выполнении цикла. И это крайне важное отличие, поскольку цикл `for` гарантированно завершается, тогда как другие циклы могут выполняться бесконечно (см. главу 11, "Счастливый конец не гарантируется").

В связи с этим возникает вопрос времени выполнения циклов. Очевидно, что для выполнения цикла, например, 100 раз потребуется выполнение 100 шагов. Иными словами, цикл проявляет линейный характер в отношении своих итераций, и это справедливо для всех разновидностей циклов. Помимо количества итераций, необходимо принимать во внимание время выполнения тела цикла. Время выполнения цикла равно числу итераций, умноженному на время выполнения тела цикла. Например, сортировка выбором является циклом, тело которого состоит из операций поиска минимального элемента в списке. Этот цикл проявляет линейный характер в отношении длины списка, а время выполнения его тела пропорционально половине длины списка. Следовательно, время выполнения сортировки выбором пропорционально квадрату длины списка.

Если принять во внимание, что поведение цикла `for` намного более предсказуемо, чем остальных разновидностей циклов, то почему бы не пользоваться исключительно циклами `for`? Дело в том, что циклы `for` менее выразительны, чем циклы с повторением, проверкой условия и рекурсией. В частности, некоторые задачи, которые могут быть решены с помощью этих разновидностей циклов, нельзя решить с помощью цикла `for`.[6]

Примером тому служит цикл Дня Сурка, в начале которого (по крайней мере, Филу Коннорсу) неизвестно, сколько итераций потребует его выполнение. Нетрудно заметить, что любой цикл `for` может быть представлен с помощью цикла `while` или `repeat`, если явно организовать в последнем ведение счетчика. А вот обратное не справедливо, поскольку нельзя заранее предугадать, сколько итераций потребует выполнение цикла `while` или `repeat`, прежде чем он завершится.

Но за предсказуемость цикла приходится расплачиваться. Если неопределенность продолжительности или конечного результата приключений можно лишь приветствовать, то при вычислениях лучше заранее знать, сколько времени отнимет то или иное вычисление, прежде чем пользоваться им или полагаться на него, особенно если оно может длиться бесконечно.

---

[6] Это относится только к приведенной здесь разновидности цикла `for`. Так, реальный цикл `for` в языках программирования наподобие C/C++ не менее функционален, чем прочие циклы, и при этом даже более выразителен. — *Примеч. ред.*

И не останавливаться

Итак, сложив письма и вложив их в конверты, вы хотите поближе познакомиться с коллегой по работе, недавно поступившей на работу, офис которой находится этажом выше вашего. Ваша прогулка в ее офис является результатом выполнения алгоритма, повторяющего ряд шагов до тех пор, пока вы не достигнете нужного офиса. Но что если вы не знаете точно, где он находится? В таком случае самый простой алгоритм, который состоит в том, чтобы выполнять шаги до тех пор, пока нужный офис не будет найден, может так и не завершиться и длиться бесконечно.

Конечно, вы не будете вечно скитаться по зданию и перейдете к алгоритму Б, который заключается в том, чтобы прервать безуспешные (а возможно, и, ради большей уверенности, неоднократные) поиски по всему этажу и вернуться на свое рабочее место. Условие завершения такого алгоритма оказывается более сложным — оно требует не только поиска конторы, но и возможности его завершения по истечении некоторого времени.

Циклы, которые не завершаются, кажутся, на первый взгляд, нелепой идеей. Несмотря на то что некоторые вычисления являются результатом выполнения цикла без условия завершения (например, веб-службы или даже простой таймер или счетчик), для вычисления, предназначенного для получения одного фиксированного результата, могут быть использованы лишь те циклы, которые должны быть непременно завершены, поскольку незавершенность цикла препятствует достижению конечного результата.

Обычно предполагается, что алгоритмы должны завершаться, ведь иначе они оказываются неэффективными для решения задач. Поэтому было бы полезно, если бы можно было каким-то образом выяснить заранее, завершится ли выполнение некоторого алгоритма. Единственная причина незавершенности алгоритма состоит в том, что один из его циклов не завершится [1], так что суждение о завершении алгоритма сводится к определению, завершатся ли все циклы, выполняемые в данном алгоритме. Различные алгоритмы складывания бумаги действительно завершаются. Это ясно следует из применения цикла *сложить три раза*, поскольку в нем количество итераций упоминается явно. Цикл **repeat** *складывание* **until** *бумага подойдет по размеру* также завершается, хотя это может и не быть

очевидным сразу. Даже если известно, что на каждом шаге складывания размер бумаги уменьшается, необходимо также учитывать, что складывание может происходить, при необходимости, по разным осям. В таком случае в какой-то момент окончательный размер сложенной бумаги станет меньше, чем размер конверта, поскольку на каждом шаге складывания он уменьшается вдвое. И хотя совсем не очевидно, сколько итераций потребуется для выполнения цикла, а следовательно, как долго будет выполняться алгоритм, все же ясно, что данный алгоритм в конечном счете завершится.

Однако в общем случае для циклов это нехарактерно. Цикл прогулки в поисках конторы вашей коллеги по работе не завершится, если в качестве условия его завершения требуется найти несуществующий офис. Причина, по которой вы не будете искать несуществующий офис до бесконечности, заключается в том, что задача его поиска является лишь небольшой частью более крупного ряда задач на целый рабочий день. И если алгоритм не в состоянии решить подзадачу, он будет отвергнут, а вместо него будет принят другой алгоритм или же данная задача будет заменена другой. Если обратиться к примеру программирования робота, просто выполняющего цикл поиска офиса и не воспринимающего его как часть более крупной задачи, то робот будет блуждать до бесконечности или до тех пор, пока не исчерпает всю свою энергию или же не будет остановлен человеком.

Так как же отличить конечный цикл от бесконечного? Как было показано ранее, ответ на этот вопрос нетрудно найти, если применяются циклы `for`, имеющие фиксированное количество итераций, а следовательно, всегда завершающиеся. Если же применяются циклы `repeat` или циклы `while`, то необходимо уяснить взаимосвязь между условием завершения и шагами в теле цикла. В главе 11, “Счастливый конец не гарантируется”, будет обсуждаться следующий интересный вопрос: имеется ли алгоритм, с помощью которого можно определить, завершится ли цикл? Ответ на него может вас удивить.

Время выполнения алгоритмов имеет значение. В частности, линейный алгоритм предпочтительнее квадратичного, а экспоненциальный алгоритм непрактичен (практичен только при обработке очень небольших объемов данных). Тем не менее экспоненциальный алгоритм лучше бесконечного алгоритма — пусть даже для обработки только небольших объемов данных. Вопрос завершения алгоритма, по существу, сводится к вопросу завершения циклов и является одним из самых главных. Именно ответ на этот вопрос отчаянно пытается найти Фил Коннорс, чтобы удовлетворить условие, завершающее цикл Дня Сурка.

11 Счастливый конец не гарантируется

Циклы и рекурсия придают силу алгоритмам. В частности, циклы дают алгоритмам возможность обрабатывать входные данные произвольного объема и сложности. Без циклов алгоритмы были бы способны обрабатывать лишь небольшие объемы простых данных. Циклы отрывают алгоритмы от грешной земли и дают им воспарить в небеса, служа для алгоритмов тем же, чем крылья служат для самолетов: без крыльев самолеты могут двигаться, но их потенциальные возможности по перевозке не будут реализованы в полной мере. Аналогично некоторые виды вычислений могут быть описаны с помощью алгоритмов и без циклов, но весь потенциал вычислений может быть реализован только с помощью циклов. Но этим потенциалом нужно каким-то образом управлять. Как наглядно показывает история из фильма *День Сурка*, управляющая структура, которой некому управлять, из блага становится злом. Ключом к управлению циклом является понимание условия, принимающего решение о завершении цикла. Филу Коннорсу в конечном счете удается завершить цикл, в котором он находится, что приводит к счастливому концу. Но это происходит, скорее, по счастливому стечению обстоятельств, а не по заранее составленному плану действий.

Как правило, предполагается, что алгоритмы завершаются, и поэтому хотелось бы еще до выполнения алгоритма знать, завершаются ли применяемые в нем циклы. Попытка выяснить данное обстоятельство может оказаться непростым делом, а следовательно, в идеальном случае эту задачу было бы желательно поручить некоторому алгоритму. Поиски алгоритма, способного определить, завершится ли другой алгоритм, известны в информатике под названием *проблема останова* (или *остановки*). Данная проблема поясняется и обсуждается в этой

главе; здесь также показана ее неразрешимость. Это, по существу, означает, что такого алгоритма не существует. Этот удивительный факт многое говорит об ограничениях алгоритмов и вычислений в общем случае.

Несмотря на то что события, развертывающиеся в истории из фильма *День Сурка*, похожи на цикл, сам сценарий фильма *не* содержит цикл. Напротив, все повторы действий разъяснены подробно. Следовательно, по сценарию нетрудно определить, что данная история завершится. Ведь режиссеру фильма должна быть известна его продолжительность, прежде чем начинать данный проект. Впрочем, нетрудно представить версию данной истории в виде пьесы, действие в которой не точно предписано, а импровизируется. В такое описание *мог бы* быть включен цикл с условием завершения, и тогда было бы не ясно, сколько времени потребовалось бы для такого импровизированного исполнения пьесы *День Сурка*. И если бы актеры (да и зрители) имели неисчерпаемый источник энергии для игры и наблюдения за ней, то заранее нельзя было бы сказать, завершится ли эта пьеса когда-нибудь.

Неуправляемый полет

На второй день своего пребывания в городке Панксатони Фил Коннорс начинает подозревать, что ему, возможно, предстоит пережить предыдущий день, когда он встает под ту же самую мелодию по радио и встречает тех же людей в таких же ситуациях. Когда Филу Коннорсу по телефону отвечают "Завтра!" он отвечает:

> *А что если завтра не настанет? Сегодня же его не было.*
> *[Гудки в трубке]*

В течение многих повторов Дня Сурка большинство событий и встреч оказываются такими же, как и день назад. Но кое-что все же меняется, поскольку Фил Коннорс всякий раз реагирует по-разному. Поначалу, как только Фил Коннорс начинает свыкаться с новой для него ситуацией, он пытается извлечь из нее выгоду и манипулировать людьми, накапливая информацию в повторяющихся Днях Сурка. Отчетливее всего это проявляется в его попытке узнать всю подноготную продюсера Риты, ставшей позже его любовницей.

Стратегия Фила Коннорса, по крайней мере в принципе, оказывается вполне работоспособной, поскольку он и другие люди совершенно по-разному переживают цикл Дня Сурка. Но самое главное, что если он ощущает повторение этого дня, то все остальные люди этого не осознают. Более того, когда он пытается поделиться своим положением с другими людьми (сначала — с Ритой, а позднее — с психиатром), они считают его сумасшедшим. Но это обстоятельство дает ему важное преимущество, поскольку он, в отличие от других, способен запоминать то, что происходило в предыдущих итерациях дня.

Нечто подобное наблюдается и в циклах алгоритмов. Одни элементы остаются постоянными в течение разных итераций цикла, тогда как другие изменяются. Например, в цикле поиска элемента в списке (см. главу 5, "Поиск идеальной структуры данных") сам список элементов не меняется, как, впрочем, и искомый элемент. Но текущее положение в списке, а следовательно, и проверяемый в настоящий момент элемент изменяется на каждой итерации до тех пор, пока не будет найден искомый элемент или же достигнут конец списка.

При более тщательном анализе обнаруживается, что тело цикла и условие его завершения имеет доступ к так называемому *состоянию* окружающего мира. Как пояснялось в главе 10, "Намылить, смыть, повторить", такой доступ происходит через *переменные.* В инструкциях тела цикла можно не только читать, но и изменять значения этих переменных. А в условии завершения можно только прочитать значения переменных, чтобы получить истинное или ложное значение и на основании этого завершить или продолжить цикл.

В алгоритме поиска элементов состояние состоит из списка, в котором осуществляется поиск, искомого элемента и текущего положения в списке во время поиска. Чтобы сделать обсуждение рассматриваемого здесь вопроса более конкретным, обратимся к следующей аналогии. Допустим, что вы сделали снимок цветка во время одной из своих недавних вылазок на природу, и вам требуется узнать, что это за вид цветка. С этой целью вам нужно обратиться к справочнику растений, на каждой странице которого находится описание конкретного цветка, включая и тот, который запечатлен на фотографии. Если предположить, что изображения растений в этом справочнике не приведены в каком-то определенном порядке, то вам придется пролистать все его страницы, чтобы найти страницу с описанием интересующего вас цветка или, просмотрев весь справочник до конца, так ничего и не найти. Поиск цветка проще всего начать с первой страницы справочника растений, затем перейти ко второй странице и т.д.

Состояние такого поиска состоит из трех элементов: изображения искомого цветка, справочника растений и текущей страницы этого справочника, снабженной, например, закладкой. Данное состояние читается в условии завершения, где проверяется, достигла ли закладка текущей страницы конца справочника или же на текущей его странице изображен такой же цветок, как и на сделанной вами фотографии. В любом из этих случаев цикл завершается, но лишь в последнем случае — с удачным исходом поиска. Цикл должен также содержать инструкции для листания страниц, если изображение цветка еще не найдено и в справочнике имеются непроверенные страницы. На данном шаге состояние цикла изменяется, поскольку изменяется текущая страница. И это единственное изменение состояния, которое происходит в цикле. Очевидно, что изменение состояния имеет решающее значение для алгоритма поиска, поскольку без листания страниц справочника изображение искомого цветка вряд ли удастся найти, если только оно не окажется на первой странице.

Данное обстоятельство очевидно и для Фила Коннорса. Поэтому он пытается изменить положение дел в городке Панксатони, чтобы условие завершения цикла Дня Сурка стало истинным и он смог вырваться из этого цикла. Но мало того что состояние цикла Дня Сурка оказывается намного более крупным, включающим всех людей (и даже самого сурка Фила из Панксатони), их мысли, отношения и так далее — не ясно даже, из чего оно вообще состоит. И только по случайному стечению обстоятельств Филу Коннорсу удается, наконец, вырваться из данного цикла.

Можно, конечно, возразить, что аналогия цикла Дня Сурка с вычислением хромает, поскольку выполняющий его компьютер и правила, по которым он действует, полностью вымышлены, напоминая в какой-то степени механизм метафизического чистилища для создания хороших людей. Но, несмотря на то что история о Дне Сурка, несомненно, вымышлена, ее аналогия с циклом вполне уместна. Кроме того, она наглядно показывает, что вычисление может происходить при самых разных обстоятельствах и на компьютерах, действующих по обширному разнообразию правил. Если правила вычисления логичны и согласованы, то нет предела вычислительным сценариям, которые способно создать наше воображение.

Мы все еще здесь?

Как упоминалось в главе 1, “Путь к пониманию вычислений”, описание алгоритма и его выполнение — это две разные вещи. Циклы и их завершаемость снова выдвигают этот вопрос на передний план, поскольку не ясно, каким образом конечное описание цикла может привести к бесконечно долгому выполнению. Это явление лучше всего объяснить с помощью метода, прослеживающего выполнение цикла и называемого *развертыванием цикла* (рис. 11.1). В упомянутом выше примере поиска изображения цветка в справочнике растений тело цикла состоит из действия по листанию страниц. Развертывание цикла означает порождение последовательности инструкций по листанию страниц.

Если речь идет о Дне Сурка, то развертывание цикла в данном случае менее очевидно, хотя принцип остается тем же. Фил Коннорс проживает каждый день, действуя в соответствии со своими целями, определяемыми его характером, и реагируя на встречи с людьми. У такого поведения нет точного описания, поэтому его нельзя назвать алгоритмом. Тем не менее выполнение цикла Дня Сурка развертывает действия из каждого дня в длинную последовательность, которая в конечном счете приводит к удовлетворению условия завершения. Мы не можем точно установить конкретное действие, изменившее состояние таким образом, что условие завершения оказалось выполненным. Это совсем иная ситуация, чем в примере с поиском изображения цветка в справочнике растений, где совершенно ясно, что листание страниц является необходимым действием для изменения состояния, а следовательно, и возможности завершить алгоритм поиска.

Рис. 11.1. Развертывание цикла означает создание последовательности копий его тела. Для каждой итерации цикла создается отдельная копия его тела. На этом рисунке показано, каким образом видоизменяется состояние, если это вообще происходит. Если состояние вообще не меняется, то условие завершения не выполняется и последовательность становится бесконечной

Допустим, что листание страниц вперед и назад чередуется. Несмотря на то что такое листание все равно изменяет состояние, оно, очевидно, воспрепятствует завершению алгоритма, если только книга не состоит всего из двух страниц. Разумеется, лишь немногие люди поступят именно так, но данный пример наглядно показывает, что одних только изменений состояния недостаточно, чтобы гарантировать завершение алгоритма. Скорее, алгоритм должен внести правильные изменения в состояние, чтобы достичь своего завершения.

Еще один аспект этой проблемы заключается в том, что нужно не только завершить алгоритм, но и получить правильный результат. Правильность в данном примере означает, что поиск останавливается на странице с изображением искомого цветка, если таковая имеется в справочнике растений, или же на последней его странице, из чего становится известно, что такого цветка в нем нет. Но рассмотрим теперь другую разновидность данного алгоритма, предполагающую одновременное листание не одной, а двух или больше страниц. В таком случае можно пропустить изображение цветка. И хотя алгоритм все равно завершится, но, дойдя до последней страницы, нельзя с уверенностью сказать, что искомый цветок отсутствует в справочнике растений.

Условие завершения цикла Дня Сурка состоит в том, чтобы Фил Коннорс стал хорошим человеком. А поскольку он этого не знает, поначалу он пытается предпринять всевозможные действия, чтобы изменить состояние и достичь завершения цикла. К числу его действий относятся различные формы самоубийства и даже убийство сурка Фила из Панксатони, но все напрасно. Осознав, что цикл не поддается его контролю, Фил Коннорс меняет свое отношение с циничного извлечения наибольшей выгоды из прожитого дня на помощь людям. В конечном счете он успешно вырывается из цикла как из нравственного замкнутого круга,

как только завершается его превращение в хорошего человека. А в качестве идеально счастливого конца Рита отвечает на его любовь взаимностью.

Задача, которую приходится решать Филу Коннорсу, особенно сложна, поскольку он действует вслепую. Ведь он должен удовлетворить условие завершения цикла Дня Сурка, даже не зная, в чем оно состоит. Более того, он не знает состояние, лежащее в основе данного цикла, и как его изменить. Конечно, нам обнаружить завершение алгоритма должно быть намного легче, поскольку мы можем видеть условие завершения, знать лежащее в его основе состояние и выяснить, каким образом действия в теле цикла способны изменить это состояние.

Конца и краю не видно

Допустим, что вам представили алгоритм, и вы должны решить, стоит ли его выполнять, т.е. даст ли он результат в течение конечного периода времени. Как вы поступите? Зная, что единственной причиной бесконечного поведения являются циклы, вы сначала должны выявить все циклы в данном алгоритме, а затем попытаться установить взаимосвязь между условием завершения и инструкциями в теле каждого цикла. Ведь окончание цикла зависит от условия завершения, а также от того, каким образом в теле цикла преобразуются значения, от которых данное условие зависит. Такой анализ позволит вам решить, завершится ли конкретный цикл и есть ли шансы на завершение всего алгоритма в целом. А для полной уверенности, что алгоритм завершится, вам придется проанализировать подобным образом каждый цикл в алгоритме.

Завершение оказывается настолько важным свойством алгоритмов, отделяя алгоритмы, способные решить задачу, от не способных ее решить, что было бы неплохо знать заранее свойство завершения любого алгоритма, которым, возможно, придется воспользоваться. Но провести анализ завершенности алгоритма не так-то просто, и для решения этой задачи может потребоваться немало времени. В связи с этим возникает искушение автоматизировать эту задачу, т.е. создать алгоритм (например, под названием `Halts` (*Останов*)), автоматически проводящий анализ завершенности проверяемого алгоритма. Такая мысль не кажется нелепой, поскольку имеется немало одних алгоритмов для анализа других алгоритмов (например, для синтаксического анализа, как пояснялось в главе 8, “Сквозь призму языка”).

Но, к сожалению, составить алгоритм `Halts` невозможно. И дело не в том, что это слишком трудно в настоящее время, а специалисты в области информатики не занимались этой задачей долго и достаточно упорно. Нет, известно, что составить алгоритм `Halts` невозможно ***в принципе***: ни теперь, ни вообще. И это обстоятельство нередко называется ***неразрешимостью проблемы останова***. Проблема останова является одной из основных в информатике. Она была предложена Аланом Тьюрингом (Alan Turing) в 1936 году в качестве примера неразрешимой задачи.

Но почему нельзя составить алгоритм `Halts`? Любой алгоритм, анализируемый алгоритмом `Halts`, сам имеет конечное описание, и поэтому нам, по-видимому, придется проверить конечное число инструкций, а также их влияние на состояние, определяющее условие завершения.

Чтобы стало понятнее, почему действительно нельзя определить алгоритм `Halts`, составим сначала алгоритм *Loop* (*Цикл*), завершаемость которого очевидна (рис. 11.2). В качестве параметра алгоритм *Loop* принимает число, присваивая его переменной x. Если применить алгоритм *Loop* к числу 1, что записывается как *Loop*(1), это приведет к вычислению, в ходе которого значение 1 присваивается переменной x. И на этом данный алгоритм останавливается, поскольку становится истинным условие завершения цикла с повторением (см. среднюю блок-схему на рис. 11.2). В противном случае, т.е. для любого другого числа, происходит циклический возврат к присваиванию значения параметра данного алгоритма переменной x. Например, применение алгоритма *Loop* к числу 2, что записывается как *Loop*(2), приводит к вычислению, в ходе которого значение 2 присваивается переменной x, а следовательно, к бесконечному циклу, поскольку условие завершения цикла с повторением остается ложным (см. правую блок-схему на рис. 11.2).

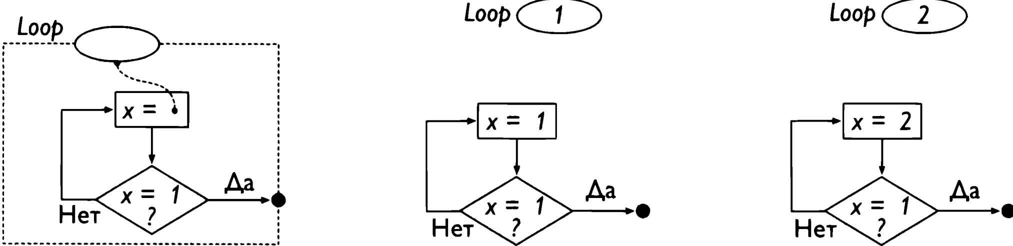


Рис. 11.2. Алгоритм **Loop** *останавливается, если вызывается с числом* **1** *в качестве своего аргумента, и зацикливается до бесконечности, если вызывается с другим аргументом. Слева: определение алгоритма* **Loop**. *Посредине: применение алгоритма* **Loop** *к числу* **1**. *Справа: применение алгоритма* **Loop** *к числу* **2**

Стратегия, призванная продемонстрировать, что алгоритма `Halts` не существует, основывается на том, чтобы первоначально допустить, что он все же существует, а затем показать, что такое допущение приводит к противоречию. Такая стратегия называется *доказательством от противного* и широко применяется в математике.

Так как же будет выглядеть алгоритм `Halts`? Как показано на примере алгоритма *Loop*, режим завершения обычно зависит от входных данных алгоритма. Это означает, что у алгоритма `Halts` должно быть два параметра: один, скажем, *Alg*, — для задания проверяемого алгоритма, а другой, скажем, *Inp*, — для задания входных данных, по которым должен быть проверен данный алгоритм. Таким образом, алгоритм `Halts` обладает структурой, приведенной на рис. 11.3 [1].

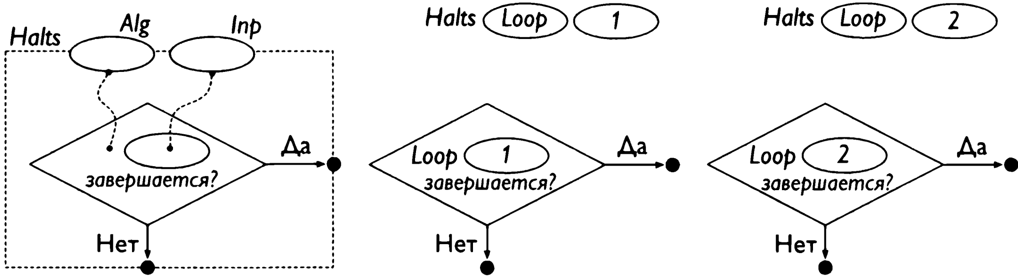


Рис. 11.3. Слева: структура алгоритма **Halts**. *Этот алгоритм принимает два параметра: проверяемый алгоритм (***Alg***) и его входные данные (***Inp***), а затем проверяет, завершается ли алгоритм* **Alg(Inp)**, *применяемый к входным данным. Посредине: применение алгоритма* **Halts** *к алгоритму* **Loop** *и числу* **1**, *что дает положительный результат. Справа: применение алгоритма* **Halts** *к алгоритму* **Loop** *и числу* **2**, *что дает отрицательный результат*

Итак, вооружившись алгоритмом `Halts`, мы можем определить другой алгоритм, `Selfie` (*Селфи*), что приведет к противоречию с предположением, что алгоритм `Halts` может существовать. Алгоритм `Halts` применяется в алгоритме `Selfie` совершенно особым образом: в нем предпринимается попытка определить, завершается ли алгоритм при условии, что в качестве входных данных задается его собственное описание. В этом случае алгоритм входит в бесконечный цикл, а в противном случае — останавливается. Определение алгоритма `Selfie` приведено на рис. 11.4.

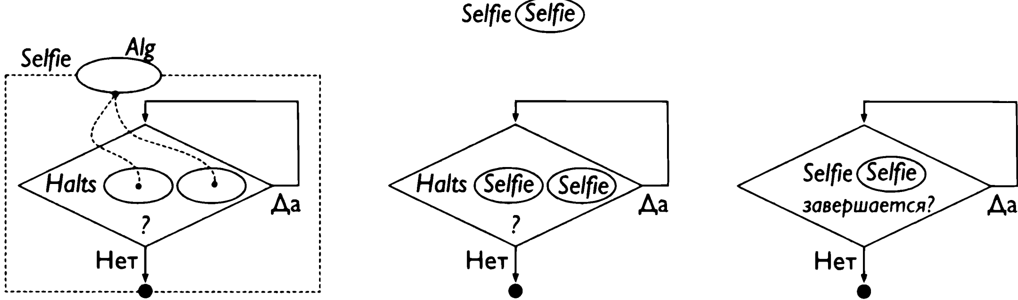


Рис. 11.4. Слева: определение алгоритма **Selfie**. *Он принимает алгоритм (***Alg***) в качестве параметра и проверяет, завершается ли данный алгоритм, будучи применен к самому себе. В этом случае алгоритм* **Selfie** *входит в бесконечный цикл, а иначе он остановится. Посредине: применение алгоритма* **Selfie** *к самому себе приводит к противоречию. Так, если применяемый к самому себе алгоритм* **Selfie** *завершается, то он входит в бесконечный цикл, т.е. не завершается. А если данный алгоритм не завершается, то он останавливается, т.е. завершается. Справа: расширение определения алгоритма* **Halts**, *дающее несколько иное представление о возникающем парадоксе*

Задание в качестве входных данных алгоритма описания его самого может показаться, на первый взгляд, странным, но на самом деле это не такая уж и странная мысль. Например, вызов `Loop(Loop)`, т.е. применение алгоритма `Loop` к самому себе, так и не завершится, поскольку алгоритм `Loop` завершается только в том случае, если в качестве его входных данных задано число `1`. А что произойдет, если применить алгоритм `Halts` к самому себе? Завершится ли вызов `Halts(Halts)`? Завершится, поскольку алгоритм `Halts`, как допускается, определяет, завершится ли любой проверяемый им алгоритм, а значит, он должен завершить данную проверку. Таким образом, он должен завершиться, когда применяется к самому себе.

Логическое обоснование для определения алгоритма `Selfie` становится ясным, если рассмотреть, что происходит, когда алгоритм `Selfie` выполняется, обрабатывая самое себя в качестве входных данных. В действительности это приводит к парадоксальной ситуации, ставящей под вопрос возможность существования алгоритма `Halts`. Чтобы уяснить, что же при этом происходит, можно развернуть определение алгоритма `Selfie`, когда он применяется к самому себе. С этой целью подставим `Selfie` вместо параметра `Alg` в определение алгоритма `Selfie`. Как показано на рис. 11.4 посредине, это приведет к тому, что цикл, который завершается, когда алгоритм `Selfie` применяется к собственному определению, *не* останавливается. Ведь если условие `Halts(Selfie, Selfie)` истинно, то алгоритм снова возвращается циклически к проверке данного условия, а иначе он останавливается.

В итоге получается блок-схема, описывающая вычисление, которое, по-видимому, ведет себя следующим образом: если алгоритм `Selfie`, применяемый к самому себе, останавливается, то он выполняется бесконечно, а если он не останавливается, то прекращается. Данное противоречие может стать еще более очевидным, если заменить вызов алгоритма `Halts` его определением, как показано на рис. 11.4, *справа*. Так завершается или нет вызов `Selfie(Selfie)`?

Допустим, что он завершается. В таком случае алгоритм `Halts`, который, как допускается, способен правильно определить, завершится ли проверяемый алгоритм при конкретных входных данных, определит, что вызов `Selfie(Selfie)` завершается. Но ведь это приведет к выбору при вызове `Selfie(Selfie)` утвердительной ветви в условной конструкции и входу в бесконечный цикл. Это означает, что если вызов `Selfie(Selfie)` завершается, то он не завершается. Следовательно, данное допущение, очевидно, неверно. А теперь допустим, что вызов `Selfie(Selfie)` не завершается. В таком случае алгоритм `Halts` даст ложный результат, а это приведет к выбору при вызове `Selfie(Selfie)` отрицательной ветви в условной конструкции и остановке. Это означает, что если вызов `Selfie(Selfie)` не завершается, то он завершается. Но и это допущение неверное.

Таким образом, если вызов `Selfie(Selfie)` останавливается, то он выполняется бесконечно, а если он не останавливается, то все равно останавливается, т.е. противоречие возникает так или иначе. Но это нельзя считать верным. Что же тогда пошло не так? За исключением стандартных управляющих структур (цикла и условной конструкции), при составлении алгоритма `Selfie` было сделано допущение, что алгоритм `Halts` способен правильно определить режим завершения проверяемых алгоритмов. А поскольку это допущение привело к противоречию, то оно должно быть неверным. Иными словами, алгоритм `Halts` не может существовать, поскольку допущение о его существовании приводит к логическому противоречию.

Такой порядок рассуждений напоминает логические парадоксы. Рассмотрим в качестве примера следующую разновидность известного парадокса парикмахера [2]. Допустим, что сурок Фил из Панксатони может видеть тени только тех сурков, которые не могут видеть свою тень. Вопрос: сможет ли сурок Фил из Панксатони увидеть свою тень? Допустим, что он сможет это сделать. В таком случае он не относится к числу тех сурков, которые не могут видеть свою тень. Но ведь это те сурки, тень которых он может видеть. А поскольку он не относится к их числу, то он не может видеть свою тень, что противоречит нашему допущению. Поэтому допустим, что он не может видеть свою тень. Но тогда он относится к числу тех сурков, тень которых он может видеть, что противоречит и этому допущению. Таким образом, что бы мы ни предполагали, все равно получаем противоречие, а это означает, что сурка, соответствующего данному описанию, просто не существует. Аналогично не существует и алгоритма остановки, соответствующего описанию алгоритма `Halts`.

Как и в примере поиска изображения в справочнике растений, мы можем вполне нормально судить о режиме завершения конкретных алгоритмов. Но не противоречит ли это тому факту, что алгоритм `Halts` не существует? Нет, это лишь показывает, что мы можем решить задачу останова в частных случаях. А это не подразумевает наличие единого способа на все случаи жизни.

Отсутствие алгоритма `Halts` не только вызывает удивление, но и наводит на глубокие размышления. Это говорит нам о том, что существуют вычислительные задачи, которые компьютеры вообще не способны решить. Такие задачи, для решения которых не существует алгоритмов, называются ***невычислимыми*** или ***неразрешимыми*** [3].

Узнав о существовании неразрешимых или невычислимых задач, вы можете разочароваться, если считали, что всякая математическая или вычислительная задача может быть решена автоматически. Но, может быть, таких случаев лишь несколько, а для решения большинства задач алгоритмы все же существуют? К сожалению, и это не так. В действительности подавляющее большинство задач неразрешимо. Исследовать этот вопрос непросто, поскольку он подразумевает два

разных понятия бесконечности [4]. Чтобы наглядно представить это различие, рассмотрим двумерную сетку, бесконечно простирающуюся во всех направлениях. Каждую разрешимую задачу можно разместить в одной узловой точке сетки. А поскольку на сетке имеется бесконечно много узловых точек, то количество разрешимых задач действительно велико. Но неразрешимых задач еще больше; их настолько много, что они просто не умещаются в узловых точках сетки. Так, если попытаться разместить их вместе с разрешимыми задачами на двумерной сетке, то они займут все пространство между узловыми точками сетки. А если рассмотреть небольшую часть сетки, скажем, мелкий квадрат, ограниченный четырьмя узловыми точками, то четырем разрешимым задачам в этих точках будет соответствовать бесконечно много неразрешимых задач, заполняющих пространство между ними.

Отсутствие алгоритмического решения для большинства задач, несомненно, является прискорбным фактом, но в то же время оно дает нам возможность глубоко проникнуть в основополагающий характер информатики как научной дисциплины. Физика раскрывает нам важные ограничения, присущие пространству, времени и энергии. Например, первый закон термодинамики о сохранении энергии гласит, что энергию нельзя создать или уничтожить; ее можно только преобразовать. А согласно специальной теории относительности Эйнштейна информацию или материю нельзя переносить быстрее скорости света. Аналогично знание о разрешимых и неразрешимых задачах показывает нам рамки и пределы вычисления. Ведь знать пределы своих возможностей, вероятно, так же важно, как и свои сильные стороны. Аналогичная мысль приписывается Блезу Паскалю: “Мы должны знать пределы своих возможностей”.

Дальнейшее исследование

История из фильма *День Сурка* наглядно показывает разные особенности циклов. Решающее значение в этой истории приобрел вопрос, закончится ли вообще цикл. Главный герой этого фильма, Фил Коннорс, пытается найти такое сочетание действий, которое позволило бы завершить цикл. Аналогично в фильме *Грань будущего* офицер из армейской службы по связям с общественностью Билл Кэйдж (Bill Cage), погибающий в тот момент, когда он сообщает о нашествии пришельцев на Землю, неоднократно отсылается обратно во времени к предыдущему дню, чтобы снова пережить прошедшие события. В отличие от фильма *День Сурка*, в котором итерация цикла длится ровно один день, на каждой последующей итерации герой фильма *Грань будущего*, Билл Кэйдж, попадает в предыдущий день все дальше и дальше, что отсрочивает его смерть по мере того, как он пробует предпринять альтернативные действия на каждой итерации. (Этот фильм, известный также под названием *Live. Die. Repeat* (Жить, умереть, повторить), образует вполне очевидный циклический алгоритм.) Это можно сравнить с неоднократным воспроизведением видеоигры, когда, зная свои прошлые ошибки, игрок пытается всякий раз перейти на более высокий уровень. Данное явление используется в видеоигре *Dark Souls* (Темные души), в которой оно становится особенностью самой игры. Герои романа *Replay* (Повторное воспроизведение) Кена Гримвуда (Ken Grimwood) также переживают отдельные части своей жизни, но переживаемые ими эпизоды становятся все более короткими, всегда завершаясь в одно и то же время и начинаясь ближе к конечному моменту. Более того, люди не контролируют повторные воспроизведения своей жизни. То же самое происходит и с главным героем фильма *Исходный код*, в котором пассажир пригородного поезда в Чикаго попадает в восьмиминутную петлю времени, пытаясь предотвратить подрыв поезда бомбой. В этой истории сам главный герой не контролирует цикл, поскольку управление его компьютерной имитацией осуществляется извне.

Цикл действует по принципу изменения отдельных частей состояния на каждой итерации до тех пор, пока не будет удовлетворено условие его завершения. Пригодными для воздействия на цикл и его завершение оказываются лишь те части состояния, которые переходят из итерации в итерацию. Так, состоянием, которое может измениться в фильме *День Сурка*, является только личность Фила

Коннорса, а все остальное (физическое окружение, воспоминания о людях и прочее) восстанавливается на каждой итерации и не может подвергаться воздействию со стороны Фила Коннорса. Анализ истории с такой точки зрения помогает раскрыть неявные предположения о влиянии действий, воспоминаний о людях и прочем, способствуя тем самым пониманию данной истории. Он также помогает обнаружить несоответствия и пробелы в канве истории.

Если действия, выполняемые в цикле, не оказывают на отдельные части состояния такого воздействия, которое способно сделать истинным условие завершения, то цикл не завершится. Так, фильм *Треугольник* кажется, по сути, состоящим из одного большого бесконечного цикла. Но поскольку некоторые герои фильма имеют доппельгенгеров[1], то структура данного цикла на самом деле оказывается более изощренной и указывает на существование нескольких перемежающихся циклов. Наиболее отчетливо бесконечный цикл проявляется в древнегреческом мифе о Сизифе, который наказан Зевсом и должен бесконечно толкать камень в гору. Как только Сизиф добирается до вершины горы, камень тотчас скатывается вниз, и ему приходится начинать все с самого начала. История Сизифа была философски интерпретирована Альбером Камю в эссе *Миф о Сизифе*, где он размышляет о том, как жить в мире, который совершенно лишен смысла и в котором никакое действие не имеет долгосрочных последствий.

[1] Доппельгенгер (doppelgänger): в парапсихологии, научной фантастике — призрачный двойник.

Рекурсия

Назад в будущее

Прочтите это еще раз

Вернувшись в свой офис, вы должны сделать несколько звонков по телефону. Чтобы позвонить кому-нибудь, вам достаточно совершить несколько касаний пальцем сенсорного экрана своего смартфона или же дать задание своему электронному помощнику инициировать звонок вместо вас. Но без помощи компьютера вам пришлось бы искать нужный номер телефона в телефонном справочнике. Как же это сделать? Ведь вы вряд ли начнете поиск с первой страницы, просматривая фамилии всех абонентов по очереди, т.е. трактуя телефонный справочник как список. Вместо этого вы, вероятно, откроете телефонный справочник где-нибудь посредине и сравните искомую фамилию с увиденным. И вам, возможно, повезет сразу же найти нужного абонента на текущей странице телефонного справочника. В таком случае вы достигли цели; в противном случае вам придется продолжить поиск до или после текущей страницы, снова открыв телефонный справочник где-нибудь посредине еще не просмотренного ряда страниц. Иными словами, вы применяете *бинарный поиск*, чтобы найти нужный номер телефона. Тот же алгоритм применяется и для поиска слова в печатном словаре или книги на полках в библиотеке.

Как пояснялось в главе 5, "Поиск идеальной структуры данных", такой алгоритм называется *рекурсивным*, поскольку выполняется как часть самого себя. Это становится очевидным при попытке пояснить данный алгоритм кому-либо. После шагов первой итерации (открытия телефонного справочника, сравнения фамилий, решения продолжить поиск до или после текущей страницы) необходимо пояснить, что придется повторить еще раз те же самые шаги. Например, можно сказать "Затем повторите те же действия", где слова "те же действия" обозначают то, что требуется повторить. Если попробовать дать название всему алгоритму в целом, например `BinarySearch`, то это название можно затем употребить в инструкции "Затем повторите алгоритм `BinarySearch`", чтобы выразить необходимость повторения действий данного алгоритма. Такое употребление названия `BinarySearch` приводит к определению алгоритма, в котором используется имя самого этого алгоритма, т.е. мы получаем описательную рекурсию, в которой имя, символ или какой-нибудь другой механизм применяется для выражения рекурсивной ссылки на себя в своем определении.

Выполнение такого алгоритма потенциально приводит к многократному повторению нескольких шагов (например, к открытию телефонного справочника на конкретной странице или сравнению фамилий). В этом отношении рекурсия подобна

циклу, который также приводит к повторению шагов алгоритма. Имеет ли в таком случае значение, что применять в алгоритме: цикл или рекурсию? Что касается самого вычисления, то это не имеет никакого значения, поскольку *любое вычисление, описываемое с помощью рекурсии, может быть также представлено с помощью циклов, и наоборот.* Но для понимания алгоритма важен способ, которым он выражается. В частности, такие задачи разбиения, как бинарный поиск или быстрая сортировка, нередко проще всего выразить с помощью рекурсии, поскольку рекурсия в данном случае оказывается нелинейной, т.е. рекурсивная ссылка на алгоритм делается неоднократно. А вычисления, в ходе которых фиксированный ряд действий повторяется до тех пор, пока не будет удовлетворено заданное условие (что соответствует линейной рекурсии), чаще всего описываются с помощью циклов.

Но рекурсия присутствует в алгоритмах не только как описательное явление. Последовательности значений сами могут быть рекурсивными. Если вы напеваете песенку “У попа была собака”, совершая поиск в словаре, то тем самым подчеркиваете рекурсивный характер данного процесса.

В отличие от таких физических воплощений рекурсии, как матрешки, которые всегда заканчиваются в определенный момент, лингвистическая рекурсия может длиться бесконечно. Бинарный поиск завершается, поскольку рано или поздно нужная фамилия будет найдена в телефонном справочнике или доступные для поиска страницы будут исчерпаны. А бесконечные песенки, как подразумевает их название, никогда не кончаются, по крайней мере если понимать их строго алгоритмически.

Чтобы продемонстрировать данное положение, проведем небольшой эксперимент. Попробуйте решить простую задачу, описанную в следующем предложении:

Прочитайте это предложение снова.

Если вы читаете теперь это предложение, значит, вы не следовали инструкциям к данной задаче, а иначе вы не дошли бы до этого места[2]. Подобно тому, как вы прекратили бесплодные поиски офиса коллеги, в данном случае вы приняли решение прервать выполнение алгоритма (т.е. чтение приведенного выше предложения) вне самого алгоритма. Рекурсивные описания, как и циклы, подвержены незавершенности, и обнаружить бесконечность рекурсии алгоритмически так же невозможно, как и бесконечность циклов.

В главе 12, “Своевременный стежок вычисляется впрок”, различные формы рекурсии поясняются с помощью метафоры путешествий во времени. Тщательно анализируя парадоксы, возникающие в путешествии во времени и в рекурсии, можно лучше понять смысловое значение рекурсивного определения.

---

[2] Как программист из анекдота, который мыл голову очень долго, — ведь инструкция на флаконе шампуня гласила: “Намылить голову, смыть пену, повторить”. — *Примеч. ред.*

12 Своевременный стежок вычисляется впрок

Слово *рекурсия* имеет два смысловых значения, что может стать причиной недоразумений, возникающих в связи с этим понятием. Рекурсия играет важную роль в описании вычисления, и поэтому ее необходимо как следует уяснить. Несмотря на то что в алгоритмах рекурсию можно заменить циклами, это на самом деле более основополагающее понятие, чем циклы, поскольку рекурсия используется для определения не только вычислений, но и данных. В частности, рекурсия требуется для определения списков, деревьев и грамматики, и альтернативы на основе циклов для этих определений не существует. Это означает, что если из двух понятий — цикла и рекурсии — приходится выбирать что-нибудь одно, то выбор должен быть остановлен именно на рекурсии, поскольку от нее зависят многие структуры данных.

Слово *рекурсия* происходит от латинского глагола *recurrere*, приблизительно означающего "бежать назад, возвращаться", и служит для обозначения некоторой формы самоподобия или ссылки на самое себя. Эти два смысловых значения приводят к разным понятиям рекурсии.

Самоподобие можно обнаружить в изображениях, содержащих самое себя в меньшей форме, например в изображении комнаты с телевизором, на экране которого отображается тот же самый вид комнаты, включая уменьшенную версию телевизора, отображающего комнату, и т.д. А ссылка на самое себя (или рекурсивная ссылка) возникает в том случае, если определение понятия содержит ссылку на себя, которая обычно задается с помощью имени или символа. В качестве примера рассмотрим определение *потомка*, которое послужило в главе 4, "Записная

книжка сыщика", основанием для составления алгоритма, вычисляющего потомков. Вашими потомками являются все ваши дети и их потомки. Здесь в определение ваших потомков входит ссылка на слово *потомок*.

В этой главе представлены различные формы рекурсии и поясняется взаимосвязь между самоподобием и рекурсивной ссылкой как формами рекурсии. Для наглядного представления некоторых особенностей рекурсии в этой главе служит история из кинотрилогии *Назад в будущее* и концепция путешествия во времени. Путешествие во времени можно рассматривать как механизм, действующий при описании последовательности событий подобно рекурсии. Мы начнем с наблюдения, что рекурсивные определения можно понимать как инструкции для путешествия во времени. Далее в главе поясняется проблема временных парадоксов и ее взаимосвязь с вопросом осмысления рекурсивных определений через понятие *фиксированных точек*.

В этой главе рассматривается также ряд свойств рекурсии. В частности, рекурсия в изображении комнаты с телевизором является прямой в том смысле, что изображение комнаты содержит самое себя как свою часть. Кроме того, рекурсия не ограничена, т.е. допускает вложенность до бесконечности. Так, если бы можно было неоднократно увеличивать изображение и расширять экран телевизора до размера самого изображения, то достичь конца вложенности никогда бы не удалось. В этой главе рассматриваются примеры косвенной и ограниченной рекурсии, а также исследуется их влияние на само понятие рекурсии.

И наконец, в этой главе поясняется тесная взаимосвязь между циклами и рекурсией, а также демонстрируется, каким образом алгоритмы, включающие циклы (например, алгоритм отслеживания Гензелем и Гретель по камешкам обратного пути домой), могут быть описаны с помощью рекурсии. При этом демонстрируется, что циклы и рекурсия равнозначны в том смысле, что любой алгоритм, в котором применяется цикл, может быть всегда преобразован в алгоритм, вычисляющий тот же самый результат с помощью рекурсии, и наоборот.

Все о времени

Подобно многим другим историям о путешествии во времени, в кинотрилогии *Назад в будущее* путешествие во времени используется как средство для решения проблем. Общий принцип состоит в том, чтобы выявить в прошлом событие, вызвавшее затруднение в настоящем и вернуться во времени, чтобы изменить событие в надежде, что тогда последующие события развернутся иначе и исключат данное затруднение. В истории из кинотрилогии *Назад в будущее* ученый доктор Браун (Doc Brown) изобрел в 1985 году машину времени, позволившую его товарищу Марти Макфлаю (Marty McFly), ученику средней школы, случайно вернуться в 1955 год и помешать своим родителям влюбиться, что оказалось

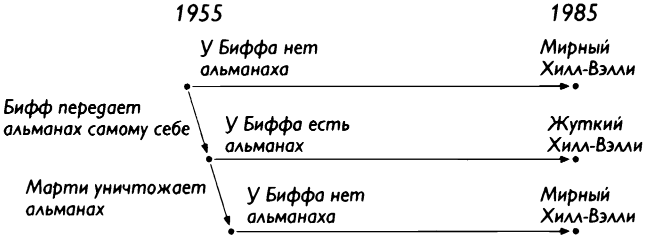
угрозой существованию его родных и самого Марти. В конечном счете ему удалось восстановить в основном ход истории и благополучно вернуться в 1985 год.

Во втором фильме Марти, его подруга Дженнифер и доктор Браун возвращаются из путешествия в 2015 год, в котором они пытались избежать беды с детьми Марти и Дженнифер. Вернувшись, они находят мрачный и жестокий 1985 год, в котором Бифф Таннен (Biff Tannen), соперник отца Марти из первого фильма, стал богатым и влиятельным человеком. Бифф убил отца Марти и женился на его матери, нажив свое состояние на спортивном альманахе за 2015 год, по которому можно было предсказывать результаты спортивных событий. Он украл спортивный альманах у Марти в 2015 году и отдал его самому себе в более молодом возрасте, отправившись в 1955 год с помощью машины времени. Чтобы восстановить ту реальность 1985 года, которую они покинули, отправляясь в 2015 год, Марти и доктор Браун отправляются обратно в 1955 год, чтобы отнять спортивный альманах у Биффа.

Доктор Браун: *Очевидно, что временной континуум был искажен, создав новую временную последовательность событий, приведшую к этой альтернативной реальности.*

Марти: *Переведите на английский, док!*

Все эти перемещения назад и вперед во времени кажутся довольно запутанными, и поэтому доктору Брауну приходится объяснять Марти некоторые последствия запланированного путешествия на машине времени с помощью рисунка на доске, подобного приведенной здесь диаграмме.



Некоторые трудности осмысления этих и других историй о путешествиях во времени коренятся в нашем восприятии реальности как единого потока событий из прошлого в будущее. Возможность путешествовать во времени затрудняет такое представление и раскрывает потенциал для многих альтернативных реальностей. Тем не менее путешествие во времени и альтернативные реальности не совсем чужды нам. Прежде всего все мы так или иначе путешествуем во времени, хотя и в весьма ограниченных пределах. Как сказал американский астроном Карл Саган (Carl Sagan), "Мы все путешественники во времени со скоростью ровно секунда в секунду" [1]. Мы часто размышляем об альтернативных реальностях,

когда планируем что-нибудь заранее или вспоминаем события из прошлого, хотя никогда и не испытываем альтернативные реальности на своем опыте.

Путешествие во времени — конечно, пленительная тема, но какое отношение она имеет к вычислению вообще и рекурсии в частности? Как следует из описанных выше историй и повседневных видов деятельности, вычисление соответствует последовательности действий, выполняемых компьютером (человеком, машиной или другим исполнителем). Следовательно, путешествие в прошлое для выполнения определенных действий соответствует их внедрению в поток действий, приводящих к настоящему состоянию окружающего мира. Цель такого путешествия состоит в том, чтобы привести состояние окружающего мира в желаемую форму, позволяющую предпринять конкретные действия в настоящем.

Например, когда Марти, Дженнифер и доктор Браун возвращаются из 2015 года в 1985 год, Марти собирается отправиться вместе с Дженнифер в давно запланированный турпоход. Но жуткое состояние, в котором они обнаруживают окружающий мир, не позволяет им этого сделать, и поэтому они отправляются в прошлое, чтобы изменить его, отобрав спортивный альманах у Биффа, и тем самым направить последовательность действий к такому состоянию в настоящем, которого они ожидают. Но путешествие во времени не ограничивается только одним только шагом в прошлом. Едва Марти и доктор Браун справились со своим заданием (отобрали спортивный альманах у Биффа), как в машину времени ударила молния, и доктор Браун перенесся в 1885 год. В дальнейшем Марти обнаруживает, что доктор Браун был убит через некоторое время после прибытия в 1885 год преступником Бьюфордом Танненом по кличке "Бешеный пес". Поэтому он отправляется обратно за доктором Брауном в 1885 год, воспользовавшись машиной времени, спрятанной доктором в заброшенном золотом руднике в 1885 году и вновь обретенной Марти в 1955 году. Оказавшись в 1885 году, он помогает доктору Брауну избежать смерти от рук Бьюфорда Таннена, и в конечном итоге ему удается вернуться в 1985 год.

Нечто подобное делают и рекурсивные алгоритмы, которые можно понимать как внедрение инструкций в поток инструкций. Каждый шаг такого алгоритма состоит из какой-нибудь элементарной инструкции или более сложной инструкции для выполнения другого алгоритма. Это, по существу, приводит к вводу инструкций данного алгоритма в текущую последовательность инструкций. В случае рекурсии это означает, что инструкции самог*о* текущего алгоритма вносятся там, где вызывается данный алгоритм.

Поскольку выполнение каждого алгоритмического шага дает некоторое промежуточное значение или некоторый иной результат, рекурсивное выполнение алгоритма делает это значение или результат доступным в том месте, откуда он вызывается. Иными словами, рекурсивный вызов алгоритма можно рассматривать как инструкцию для путешествия назад во времени, чтобы начать вычисление, которое делает требующийся результат доступным в настоящий момент.

В качестве первого примера рассмотрим действия Марти по рекурсивному алгоритму. В частности, мы можем определить алгоритм *ToDo* с помощью нескольких приведенных ниже уравнений, сообщающих Марти, что и когда ему делать [2].

```
ToDo(1985) = ToDo(1955); пойти в турпоход
ToDo(1955) = вернуть альманах; ToDo(1885); вернуться в 1985 год
ToDo(1885) = спасти дока от Бьюфорда Таннена; вернуться в 1985 год
```

Данный алгоритм можно было бы расширить, включив в него путешествие во времени в 2015 год, но и трех приведенных выше примеров такого путешествия будет достаточно, чтобы продемонстрировать ряд особенностей рекурсии. Во-первых, в уравнении для *ToDo*(1985) обнаруживается, что для действий, которые должны быть предприняты в 1985 году, требуются действия, которые должны быть предприняты в 1955 году, поскольку для турпохода Марти вместе с Дженнифер окружающий мир должен оказаться в другом состоянии. Это требование выражается в алгоритме *ToDo* с помощью рекурсии. В частности, на одном из шагов алгоритма *ToDo* выполняется сам алгоритм *ToDo*. И во-вторых, при рекурсивном выполнении уравнения *ToDo*(1955) используется аргумент, отличный от аргумента в уравнении *ToDo*(1985), частью которого он является. Это означает, что рекурсия не приводит к точному воспроизведению, в отличие от упоминавшегося ранее изображения комнаты с телевизором. И это весьма существенно для поведения вычисления в отношении завершения.

А теперь рассмотрим порядок развертывания вычислений, когда рекурсивный алгоритм выполняется с аргументом 1985. На первом шаге вычисления уравнения *ToDo*(1985) выполняется уравнение *ToDo*(1955), а это означает, что Марти придется вернуться обратно в 1955 год, чтобы отнять у Биффа спортивный альманах, прежде чем он сможет отправиться в турпоход. Но прежде чем он сможет возвратиться в 1985 год, ему придется предпринять шаги, описанные в уравнении *ToDo*(1885), т.е. он должен будет отправиться еще дальше во времени, назад в 1885 год, чтобы спасти доктора Брауна. После этого Марти возвращается в 1985 год, когда он может, наконец-то, отправиться со своей подругой в давно запланированный поход.

Тщательно проанализировав третье уравнение в данном алгоритме, мы обнаружим в нем нечто особенное. Вместо возврата в 1955 год, т.е. к тому моменту времени, из которого Марти вернулся, когда началось вычисление уравнения *ToDo*(1885), данный алгоритм возвращается непосредственно в 1985 год. Именно это и происходит в третьем фильме из кинотрилогии *Назад в будущее.* И в этом есть смысл, поскольку переходить обратно в 1955 год лишь для того, чтобы затем вернуться обратно в 1985 год, было бы не очень удобно. Ведь многие люди предпочитают прямые авиарейсы авиарейсам с пересадками.

Тем не менее рекурсия обычно действует иначе. Когда рекурсивное вычисление завершается, оно автоматически возвращается в ту точку, где оно было

оставлено и откуда затем продолжается. В данном примере это означало бы переход в 1885 год, возвращающий в 1955 год. Причина такого поведения состоит в том, что при рекурсивном вычислении, как правило, неизвестно, каким образом продолжается вычисление. Поэтому надежнее всего вернуться в исходную точку, чтобы не пропустить ничего важного, хотя в данном конкретном случае прямой переход вполне оправдан, поскольку на следующем шаге предстоит вернуться в 1985 год. Выполнять ли два последовательных шага или только один — вопрос эффективности алгоритма. В истории из кинотрилогии *Назад в будущее* это действительно важно, поскольку потоковый накопитель, обеспечивающий путешествие во времени, требует немало энергии для каждого времен*н*ого перехода. Доктор Браун в 1955 году жалуется на конструкцию своей машины 1985 года:

> *Как я мог оказаться таким легкомысленным? 1,21 гигаватта! О Том [Томас Эдисон], как же мне получить такую мощность?! Это невозможно, решительно невозможно!*

Не важно, когда

Чтобы воспользоваться машиной времени доктора Брауна, нужно указать точную целевую дату и время путешествия во времени. Но поскольку цель путешествия во времени состоит в изменении цепочки причинно-следственных событий, то указывать точную целевую дату и время на самом деле не обязательно, при условии, что прибытие происходит прежде события, которое должно быть изменено. Если допустить, что имеется таблица, содержащая даты и время для всех событий, в изменении которых мы заинтересованы, то алгоритм ToDo можно выразить иначе, употребив предназначенные причинно-обусловленные изменения вместо явных времен*н*ых переходов. Фактически алгоритмический стиль употребления прямых переходов в алгоритме ToDo довольно стар. Именно таким образом действуют низкоуровневые языки для программирования микропроцессоров, помечая метками отдельные фрагменты кода, а затем применяя инструкции перехода для перемещения между этими фрагментами кода. Конечно, все управляющие структуры, обсуждавшиеся в главе 10, “Намылить, смыть, повторить”, могут быть реализованы с помощью подобных переходов. Тем не менее программы, в которых употребляются прямые переходы, трудно понимать и анализировать, особенно если переходы выполняются случайным образом. Обычно такой очень запутанный код называется *спагетти* (макаронным). Такие прямые переходы являются весьма устаревшим способом выражения алгоритмов, хотя они до сих пор применяются в низкоуровневом представлении кода, предназначенного для выполнения на аппаратном обеспечении компьютеров. Вместо этого в алгоритмах употребляются условные конструкции, циклы и рекурсия.

Поскольку безусловные переходы являются устаревшими и не рекомендованными к применению управляющими структурами, то как же выразить алгоритм *ToDo* без них? В этом случае вместо конкретных лет для пометки последовательности действий можно употребить имена для обозначения целей, которые предполагается достичь в описываемом алгоритме. Если алгоритм вызывается с конкретной целью, то среди имеющихся уравнений можно найти уравнение, подходящее для этой цели. И как только рекурсивное выполнение завершится, произойдет автоматический возврат в ту точку, где оно было начато. Хотя конкретные годы очень важны для установления разных культурных контекстов в фильме, они не имеют особого значения для установления правильной последовательности шагов в алгоритме, которая зависит лишь от относительного упорядочения, учитывающего причинно-следственные зависимости действий. Таким образом, алгоритм *ToDo* можно заменить новым алгоритмом *Goal* (Цель), который определяется следующим образом:

```
Goal(жить сейчас)        = Goal(восстановить мир); пойти в турпоход
Goal(восстановить мир) = отнять альманах; Goal(спасти доктора)
Goal(спасти доктора)     = спасти доктора от Бьюфорда Таннена
```

Вычисление данного алгоритма развертывается таким же образом, как и алгоритма *ToDo*, за исключением того, что моменты времени в нем не указаны явно, а возврат в 1985 год происходит в два этапа.

Точно в срок

Вычисление, происходящее в результате выполнения рекурсивного алгоритма *ToDo* или *Goal*, оказывает влияние на состояние окружающего мира и не является непосредственно видимым при прослеживании шагов алгоритмов по мере их развертывания. Чтобы более отчетливо продемонстрировать связь между рекурсивным выполнением и вычислением, давайте рассмотрим в качестве еще одного примера простой алгоритм подсчета элементов в списке. В качестве конкретной аналогии этого алгоритма может служить подсчет карт в колоде.

В этом алгоритме должны различаться два случая. Во-первых, если колода пуста, то она не содержит карт. И во-вторых, если колода не пуста, то количество карт в ней можно определить, прибавляя 1 к текущему количеству карт в колоде, не считая первой карты сверху. Если представить колоду карт в виде списка, то в данном алгоритме должны соответственно различаться пустой и непустой списки. В последнем случае единица прибавляется к результату применения данного алгоритма к хвосту списка, т.е. к содержимому списка без первого его элемента. А поскольку любое рекурсивное обращение к непустому списку само приведет к прибавлению единицы, то по данному алгоритму будет прибавлено столько единиц, сколько элементов в списке. Такой алгоритм под названием *Count*

(Подсчет) может быть описан с помощью двух приведенных ниже уравнений, в которых учитываются случаи пустого и непустого списков.

```
Count( )            = 0
Count(x → rest) = Count(rest) + 1
```

Оба случая различаются благодаря разным формам представления аргументов алгоритма Count в левой части уравнений. В первом уравнении пробел обозначает, что данное уравнение применяется к пустым спискам, не содержащим ни одного элемента. А во втором уравнении шаблон x→rest представляет непустой список, где x — первый элемент, а rest — хвост списка. По определению алгоритма в данном случае подсчитываются все элементы в хвосте списка путем вызова Count(rest) и прибавления 1 к полученному результату. Такой способ выбора между разными случаями для применения алгоритма называется *сопоставлением с шаблоном.* Зачастую сопоставление с шаблоном→ может быть использовано вместо условных конструкций с целью отчетливо разделить разные случаи, которые должны быть рассмотрены в алгоритме. Кроме того, сопоставление с шаблоном обеспечивает прямой доступ к отдельным частям структуры данных, которой оперирует алгоритм, что иногда позволяет сократить определения. В данном случае x обозначает первый элемент списка, хотя и не используется в определении, указанном в правой части уравнения. Но поскольку параметр rest обозначает хвост списка, то может использоваться в качестве аргумента для рекурсивного вызова алгоритма Count. Еще одно примечательное преимущество сопоставления с шаблоном заключается в том, что оно явным образом отделяет рекурсивную часть определения от нерекурсивной.

Рекурсивное уравнение для алгоритма Count можно пояснить как следующее гипотетическое утверждение: если бы было известно количество элементов в хвосте списка, то общее количество элементов можно было бы получить, просто прибавив 1 к этому количеству. Представьте, что кто-то уже подсчитал количество карт в колоде, не считая верхней карты, и оставил наклейку с написанным на ней количеством карт. В таком случае общее количество карт можно было бы определить, просто прибавив 1 к количеству, указанному на наклейке.

Но поскольку эти сведения недоступны, то их придется вычислить, рекурсивно применяя алгоритм Count к хвосту списка rest. И здесь прослеживается связь с путешествием во времени. Рассмотрим для этого моменты времени, когда происходят разные шаги вычисления. Если в настоящий момент требуется прибавить 1, то вычисление алгоритма Count(rest) должно быть уже завершено, а следовательно, оно должно было быть начато в какой-то момент времени в прошлом. Аналогично, если требуется немедленно вычислить общее количество карт в колоде, то можно было бы попросить кого-нибудь другого подсчитать это количество, кроме верхней карты, несколько минут назад, чтобы иметь результат теперь. Таким образом, рекурсивный вызов алгоритма можно рассматривать как

путешествие в прошлое для создания вычислительных шагов, которые завершатся в настоящем, чтобы оставшиеся операции могли быть выполнены немедленно.

Чтобы продемонстрировать пример вычисления по рекурсивному алгоритму, подсчитаем разные предметы, которые Марти собирается взять с собой в 1885 год: ковбойские сапоги (B), пару портативных раций (W) и ховерборд (H). Чтобы применить алгоритм `Count` к списку B→W→H, нам придется воспользоваться вторым уравнением для определения этого алгоритма, поскольку данный список непустой. С этой целью мы должны сопоставить список с шаблоном x→rest, в результате чего параметр x будет ссылаться на первый элемент B, а параметр rest — на элементы W→H в хвосте списка. В таком случае в уравнении, определяющем алгоритм `Count`, предписывается прибавить 1 к рекурсивному вызову `Count(rest)`, как показано ниже.

```
Count(B→W→H) = Count(W→H) + 1
```

Чтобы выполнить такое сложение, потребуется результат вызова `Count(W→H)`, который, как известно, должен быть равен 2. Но соответствующее вычисление по алгоритму `Count` должно было быть начато раньше, если требуется, чтобы его результатом можно было бы воспользоваться теперь.

Чтобы уточнить эти временн*ые* рамки, допустим, что основной шаг вычисления (например, сложение двух чисел) отнимает одну единицу времени. И тогда продолжительность вычисления можно рассматривать в подобных единицах времени. Начало и завершение вычисления относительно настоящего может быть выражено в виде расстояния, измеряемого в единицах времени. Так, если обозначить настоящее как момент времени 0, то основной шаг, который предпринимается теперь, завершится в момент времени +1. Аналогично вычисление, отнимающее два шага и завершающееся теперь, должно было быть начато в момент времени -2.

Сколько времени продлится рекурсивное вычисление, совсем не очевидно, поскольку это во многом зависит от того, как часто совершаются рекурсивные шаги, что, в свою очередь, зависит, в общем, от входных данных. Так, в примере алгоритма `Count` количество рекурсий, а следовательно, и время выполнения, зависит от длины списка. Как и в тех алгоритмах, в которых применяются циклы, время выполнения рекурсивных алгоритмов может быть описано только в виде функции от объема входных данных. В связи с этим наблюдением возникает следующий вопрос о представлении рекурсии в виде путешествия в прошлое: как далеко во времени назад должен распространяться рекурсивный вызов алгоритма `Count`, чтобы завершить его работу вовремя и предоставить его результат для сложения? По-видимому, нам придется отправиться довольно далеко в прошлое, чтобы иметь достаточно времени для всех требующихся расширений и дополнений данного алгоритма. Но поскольку длина списка нам неизвестна, то мы не знаем, как далеко назад следует отправиться во времени.

Правда, нам совсем не обязательно знать объем входных данных, чтобы эффективно воспользоваться путешествием во времени. Главное наблюдение состоит в том, что рекурсивное вычисление достаточно начать лишь на одну единицу в прошлом, поскольку любое дополнительное время, которое может потребоваться для выполнения дальнейших рекурсивных вычислений, наверстывается отправкой соответствующих рекурсивных вызовов еще дальше в прошлое независимо от того, сколько времени отнимает рекурсивное вычисление. Принцип действия такого рекурсивного вычисления наглядно показан на рис. 12.1.

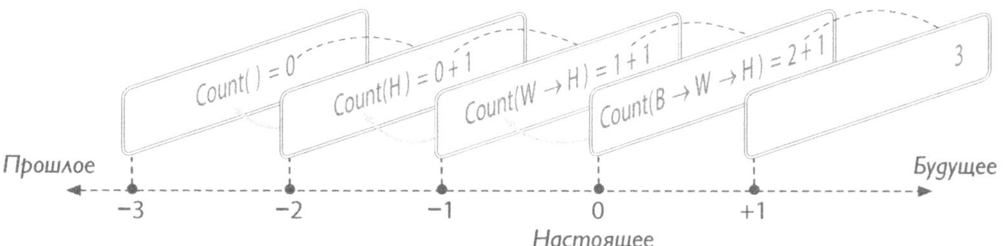


Рис. 12.1. Рекурсивный подсчет количества элементов в списке посредством путешествия в прошлое. При выполнении алгоритма **`Count`** *над непустым списком запускается вычисление, прибавляющее 1 к результату вычисления алгоритма* **`Count`** *для хвоста списка. Благодаря выполнению вычисления для хвоста списка на один шаг в прошлом его результат становится доступным в настоящем, а результат сложения будет доступен на один шаг в будущем. А выполнение алгоритма* **`Count`** *над непустым списком в прошлом приводит к дальнейшему выполнению данного алгоритма еще дальше в прошлом*

Как показано на рис. 12.1, выполнение алгоритма `Count(B→W→H)` порождает вычисление суммы `Count(W→H)+1`. Как только рекурсивное вычисление завершится, алгоритм `Count` продвинется ровно на один шаг, чтобы прибавить 1 к результату рекурсивного вызова, а окончательный результат `3` будет получен в момент времени `+1`. Для того чтобы результат рекурсивного вызова стал доступным в какое-то конкретное время, достаточно начать вычисление на одну единицу времени раньше. Например, чтобы получить результат вызова `Count(W→H)` в настоящем, т.е. в момент `0`, необходимо отправить вычисление лишь на одну единицу времени в прошлое. Как показано на рис. 12.1, это вычисление приведет к выражению `1+1` в момент времени `-1`, которое вычисляется за один шаг и дает результат `2`, доступный в момент времени `0`, т.е. именно тогда, когда нужно.

А можно ли мгновенно получить выражение `1+1` в результате вызова `Count(W→H)`? И этого можно добиться, отправив соответствующий рекурсивный вызов `Count(H)` на один шаг в прошлое, т.е. к моменту времени `-2`, когда получается выражение `0+1`, которое также вычисляется за один шаг и дает результат `1`, доступный на одну единицу времени позднее, т.е. в момент времени `-1`. В этом выражении `0` обозначает результат вызова `Count( )`, который

был отправлен обратно к моменту времени `-3`. В общем, можно заметить, что, неоднократно начиная рекурсивные вычисления в прошлом, исходное вычисление можно завершить лишь на 1 единицу времени позже в будущем. И это справедливо независимо от длины списка. Ведь чем длиннее список, тем дальше назад во времени придется отправиться.

Следует, однако, иметь в виду, что аналогия с путешествием во времени может создать ложное впечатление, будто рекурсия выглядит сложнее, чем она есть на самом деле. Для выполнения рекурсивного алгоритма компьютером не требуется никаких временн*ых* хитростей и замысловатого планирования. Это было очевидно при рассмотрении бинарного поиска в главе 4, "Записная книжка сыщика", а также быстрой сортировки вместе с сортировкой слиянием в главе 6, "Сортировка алгоритмов сортировки".

Аналогия с путешествием во времени наглядно показывает две формы рекурсии и их взаимосвязь. С одной стороны, в определениях таких алгоритмов, как *`Goal`* или *`Count`*, рекурсия применяется через рекурсивное обращение к самому себе. Такая форма рекурсии называется ***описательной*** (descriptive), поскольку рекурсия происходит в описании алгоритма. С другой стороны, когда выполняется рекурсивный алгоритм, результирующая последовательность сходных событий или вычислений составляется из экземпляров рекурсивного описания с разными значениями, используемыми в качестве параметров. Такая форма рекурсии называется ***развернутой*** (unfolded). Как показано на рис. 12.1, вследствие повторяющегося раскрытия применений рекурсивного алгоритма описательная рекурсия преобразуется в развернутую. Иными словами, выполнение описательной рекурсии (т.е. рекурсивного алгоритма) приводит к соответствующему развернутому рекурсивному вычислению. Такое представление можно подытожить следующим уравнением:

```
Выполнение(Описательная рекурсия) = Развернутая рекурсия
```

Рекурсивное изображение комнаты с телевизором, на экране которого отображается та же самая комната, служит характерным примером развернутой рекурсии. Принимая во внимание упомянутую выше взаимосвязь, имеется ли такая описательная рекурсия, выполнение которой привело бы к подобной развернутой рекурсии? Да, имеется. В качестве примера можно привести инструкции по съемке неподвижного кадра комнаты на видеокамеру и выводу этого кадра на экран телевизора, находящегося в той же самой комнате. Следуя этим инструкциям, можно получить развернутое рекурсивное изображение.

В кинотрилогии *Назад в будущее* описательная рекурсия фиксируется в алгоритмах *`ToDo`* и *`Goal`*. В частности, цели и планы изменить прошлое являются формой описательной рекурсии. И когда планы выполняются, история разворачивается в события, которые нередко оказываются очень похожими одно на другое, например похожие сцены в кафе или баре и на скейтборде или ховерборде.

И дело не ограничивается только вымышленными персонажами фильмов о путешествиях во времени, определяющими цели и планирующими изменить прошлое. В действительности все мы иногда вовлечены в описательную рекурсию, спрашивая, что бы произошло, если бы поступить иначе? Но в отличие от персонажей фильмов, мы не можем выполнить подобные планы.

Борьба с парадоксами с помощью фиксированных точек

Путешествие во времени оказывается столь увлекательным и захватывающим предметом отчасти потому, что оно может создавать парадоксы, невозможные ситуации, различающиеся логическими противоречиями — когда нечто одновременно и существует, и в то же время не существует. Известным тому примером служит *парадокс дедушки* в отношении путешественника во времени, который возвращается в прошлое, чтобы убить своего дедушку, прежде чем тот родит его отца или мать. В таком случае становится невозможным существование самого путешественника во времени, включая его путешествие во времени и убийство дедушки. Парадокс дедушки может быть использован как аргумент в пользу того, что путешествие во времени в прошлое невозможно, поскольку оно может привести к таким логически невероятным ситуациям, которые противоречат нашему пониманию причинности. Во втором фильме кинотрилогии *Назад в будущее* доктор Браун так предупреждает о возможных последствиях встречи Дженнифер с самой собой в будущем:

> *Такая встреча могла бы создать временной парадокс, который мог бы привести к цепной реакции, способной разрушить ткань пространственно-временного континуума и уничтожить всю вселенную! Правда, это наихудший вариант. На самом деле разрушение могло бы носить весьма локальный характер, ограничиваясь лишь нашей галактикой.*

В качестве одного из ответов на вопрос парадоксов можно допустить, что действия, вызывающие парадокс, предпринять фактически невозможно. Например, даже если бы и можно было путешествовать во времени в прошлое, убить своего дедушку все равно было бы невозможно. Допустим, в частности, что вы пытаетесь убить своего дедушку из огнестрельного оружия. Но тогда вам, возможно, не удастся раздобыть оружие, а если вы и попытаетесь воспользоваться им, то будет осечка или ваш дедушка увернется от пули, и даже если она попадет в него, он может быть лишь ранен и уцелеет. Природа пространственно-временного континуума может быть такова, что допускает лишь те действия в прошлом, которые согласованы с тем, что, определенно, произойдет в будущем.

А существует ли в области вычислений нечто равнозначное парадоксу дедушки? Да, существует. Подобным парадоксом может, например, считаться любое

бесконечное выполнение рекурсивного алгоритма. Такое представление о парадоксах будет более подробно разъяснено ниже с помощью понятия *фиксированной точки.*

Чтобы создать парадокс, по-видимому, необходимо определить рекурсивное вычисление, вызывающее путешествие в прошлое и изменяющее его таким образом, что само вычисление, которое вызывает это путешествие, исчезает или становится невозможным. Останавливаясь поближе при перемещении во времени назад, пожалуй, можно получить алгоритм, который удаляет сам себя или уничтожает компьютер, на котором выполняется. В таком случае выполнение алгоритма просто прервется, и никакого настоящего парадокса, который следовало бы разрешить, не возникнет. Но в данном примере рассматривается лишь развернутая рекурсия. А ведь имеется немало случаев описательной рекурсии, которые приводят к парадоксам. Напомним рекурсивное определение алгоритма `Count`, которое задается уравнением, в котором количество элементов списка получается прибавлением 1 к количеству элементов в хвосте списка. Здесь нет никакого парадокса, но допустим, что мы немного изменили определение алгоритма `Count`, чтобы применять его рекурсивно ко всему списку в целом, а не только к его хвосту, как показано ниже.

```
Count(list) = Count(list) + 1
```

Похоже, что здесь определяется количество элементов, которое на один больше, чем их имеется в списке, а это уже явное противоречие. Аналогичным примером служит следующее уравнение, в котором предпринимается попытка определить число *n*, которое на 1 больше самого себя.

```
n = n + 1
```

Здесь также имеется противоречие или парадокс, и поэтому данное уравнение не имеет решения. С точки зрения аналогии с путешествием во времени попытка вернуться во времени назад, чтобы вычислить значение *n* или `Count(list)`, не прекратится, а следовательно, так и не будет достигнут момент, когда можно прибавить 1. Иными словами, требующиеся в прошлом действия никак нельзя согласовать с действием в настоящем, что вызывает следующий парадокс: количество элементов в списке должно быть одновременно равно двум разным значениям.

В реальности очевидные парадоксы нередко разрешаются накладываемыми физикой ограничениями. Например, кажущаяся бесконечной рекурсия в изображении комнаты с телевизором ограничивается разрешающей способностью видеокамеры. Как только глубоко вложенное изображение телевизора достигнет величины одного пикселя (т.е. элемента изображения), рекурсия прервется, поскольку она не сможет далее представить изображение комнаты как часть этого пикселя. Подобное ограничение срабатывает и при возникновении акустической обратной связи, когда сигнал с выхода усилителя подается (по цепи положительной

обратной связи) на его вход через подключенный к нему микрофон, и в итоге усиление не продолжается до бесконечности. В данном случае парадокс разрешается ограниченными физическими возможностями микрофона и усилителя в отношении амплитуды сигналов, которые они способны обрабатывать.

Как обсуждалось в главе 9, "Поиск нужного тона: смысл звука", у языков имеется своя семантика, а семантика алгоритма, выражаемая на каком-то языке программирования, является вычислением, которое производится при выполнении алгоритма. С этой точки зрения смысловое значение противоречивых и приводящих к парадоксам алгоритмов не определено, т.е. не существует вычисления, которое сделало бы именно то, чего требует такой алгоритм. Как же узнать, является ли рекурсивный алгоритм противоречивым и вызывающим парадокс?

Пример рекурсивного определения числа поможет пролить свет на этот вопрос. Рассмотрим следующее уравнение, в котором число определяется как равное своему квадрату:

$$n = n \times n$$

На самом деле здесь нет никакого противоречия. Имеются два натуральных числа, 1 и 0, для которых данное уравнение истинно. И это самое главное для понимания рекурсивных определений. Определенное число является решением уравнения, т.е. это число, будучи подставленным вместо переменной n, приведет к истинному утверждению, а не к противоречию. В данном случае подстановка числа 1 дает истинное утверждение $1=1\times1$, а следовательно, это число является решением данного уравнения. А вот уравнение $n=n+1$ решения не имеет. Не имеет решения и уравнение `Count(list)=Count(list)+1`, хотя исходное уравнение его имеет, и для него при вычислении подсчитывается именно количество элементов в списке.

Уравнение, содержащее одну и ту же переменную в обеих своих частях, можно рассматривать как определяемое преобразованием. Например, уравнение $n=n+1$ гласит, что переменная n определяется прибавлением 1 к ней самой, а уравнение $n=n\times n$ — что переменная n определяется ее умножением на саму себя. Любое число n, на которое преобразование не оказывает никакого влияния, называется *фиксированной точкой* данного преобразования. Как подразумевает само название, значение в данной точке не изменяется и остается фиксированным.

Примерами фиксированных точек, для которых понятие *точки* подходит идеально, служат геометрические преобразования. Рассмотрим в качестве примера вращение изображения вокруг его центра. Свое положение изменяют все точки изображения, кроме центральной точки, которая остается на своем месте. Таким образом, центр изображения служит фиксированной точкой для преобразования вращением. В действительности центр изображения оказывается единственной фиксированной точкой вращения. В качестве другого примера рассмотрим отражение изо-

бражения по одной из его диагоналей. В этом случае фиксированными точками преобразования отражением оказываются все точки на диагонали. И наконец, у сдвига изображения влево фиксированные точки вообще отсутствуют, поскольку это преобразование оказывает влияние на все точки изображения. Если же обратиться к числовым примерам, то у преобразования “прибавлением единицы” отсутствуют фиксированные точки, тогда как у преобразования “умножение на самого себя” имеются две фиксированные точки: 1 и 0.

Какое же преобразование соответствует уравнению, в котором определяется алгоритм *Count*? Прежде всего, следует заметить, что измененное определение `Count(list)=Count(list)+1` очень похоже на определение $n=n+1$, за исключением того, что в нем имеется параметр. Данное определение соответствует преобразованию, которое означает, что алгоритм *Count*, применяемый к указанному параметру *list*, определяется прибавлением 1 к результату его применения. У такого преобразования, как у преобразования в уравнении $n=n+1$, фиксированная точка отсутствует. А преобразование из первоначального определения `Count(x→rest)=Count(rest)+1` означает, что алгоритм *Count*, применяемый к параметру *list*, определяется применением *Count* к списку с удаленным первым элементом и последующим прибавлением 1. У такого преобразования (наряду с уравнением, определяющим случай пустого списка) имеется фиксированная точка, которой оказывается функция, подсчитывающая элементы в списке.

Смысл рекурсивного уравнения заключается в фиксированной точке положенного в его основу преобразования, т.е. если такая фиксированная точка существует [3]. Когда рекурсивное уравнение описывает алгоритм, фиксированная точка обозначает вычисление, которое остается устойчивым при преобразовании, что делает его применимым для большого количества случаев. Как правило, преобразование адаптирует аргумент алгоритма и может также изменить результат рекурсивного вызова. Так, для алгоритма *Count* в качестве аргумента задается список с первым удаленным элементом, а получаемый в итоге результат увеличивается на 1. Что касается алгоритма *Goal*, то рекурсивное уравнение выполняет *Goal* с разными целями и добавляет другие действия, уместные для каждого конкретного случая.

С точки зрения путешествия во времени фиксированная точка рекурсивного алгоритма описывает вычисление, действие которого в прошлом согласовано с результатами в настоящем. Именно поэтому фиксированные точки так важны для рекурсии. Рекурсивные алгоритмы, чтобы достичь успеха, должны, подобно путешественникам во времени, корректно вести себя в прошлом, избегая парадоксов. Если у рекурсивного алгоритма имеется фиксированная точка, то она обозначает значащее вычисление, в противном случае мы имеем парадокс. В отличие от парадоксов путешествия во времени Вселенная при этом не уничтожается — просто не вычисляется то, что требуется. Понимание смысла рекурсивного

алгоритм как фиксированной точки достигается непросто. Поэтому в главе 13, "Все дело в интерпретации", демонстрируется другой способ придания смысла описаниям рекурсивных алгоритмов.

Зацикливать или не зацикливать

Рекурсия является управляющей структурой, эквивалентной циклам. Это означает, что любой цикл в алгоритме можно заменить рекурсией и наоборот. В некоторых случаях то или иное решение представляется более естественным, но такое впечатление нередко обусловлено отклонением от первоначального представления о любой из этих управляющих структур. Например, как представить алгоритм поиска Гензелем и Гретель обратного пути домой не в виде цикла? Напомним этот алгоритм:

> Найти не посещавшийся прежде камешек и направиться к нему, и так до тех пор, пока не окажешься дома.

Очевидно, что это пример цикла с повторением. И хотя это ясное и простое описание алгоритма, следующая равнозначная рекурсивная его версия под названием *FindHomeFrom* ("Отыскать дом из") оказывается лишь немного длиннее [4]:

```
FindHomeFrom(дом)  = ничего не делать (завершение алгоритма)
FindHomeFrom(лес)  = FindHomeFrom(следующий не посещавшийся камешек)
```

Подобно другим представленным здесь рекурсивным алгоритмам, алгоритм *FindHomeFrom* задается несколькими уравнениями, в которых разные случаи различаются параметрами. В данном случае параметр обозначает место, откуда начинается поиск пути домой по данному алгоритму, а два возможных случая — это текущее положение Гензеля и Гретель уже дома или все еще в лесу.

Безусловно, рекурсивный алгоритм оказывается немного более точным, чем его циклический вариант, поскольку он сразу же завершится, если отец отведет Гензеля и Гретель во двор их дома. В таком случае Гензелю вообще не придется разбрасывать камешки, потому что они с Гретель не покинут свой дом. Циклический же алгоритм предписывает им искать камешек даже в этой ситуации, что может привести к бесконечному вычислению. Но это не изъян циклов в общем случае, а скорее, следствие применения цикла с повторением для описания данного алгоритма, в котором более уместным был бы следующий цикл с проверкой условия: "Пока еще не дома, найти не посещавшийся прежде камешек и направиться к нему". В таком цикле условие завершения проверяется до выполнения тела цикла.

Приведенное выше рекурсивное описание алгоритма наглядно показывает, каким образом цикл можно выразить с помощью рекурсии. Во-первых, оба результата проверки условия завершения (уже дома или все еще в лесу) явно представлены в виде параметров уравнений. Во-вторых, тело цикла становится

частью того уравнения, параметр которого представляет случай незавершенности (т.е. местоположение в лесу). И в-третьих, продолжение цикла выражается в виде рекурсивного выполнения алгоритма с соответственно измененным аргументом (т.е. местоположением следующего еще непосещенного камешка).

Как было показано, рекурсивная версия цикла Дня Сурка в главе 10, "Намылить, смыть, повторить", использовала условную конструкцию. Но ее можно выразить и с помощью уравнений и сопоставления с шаблоном следующим образом:

```
GroundhogDay(true) = ничего не делать
GroundhogDay(false) = пережить день; GroundhogDay(Фил хороший человек?)
```

По сравнению с циклом **repeat** *пережить день* **until** *Фил хороший человек* приведенный выше вариант алгоритма кажется более сложным. Но было бы неверно сделать из этого вывод, что программировать циклы всегда легче, чем рекурсию. Так, рекурсия оказывается особенно хорошо подходящей для решения задач, которые могут быть разложены на подзадачи (см. алгоритмы "разделяй и властвуй" в главе 6, "Сортировка алгоритмов сортировки"). Рекурсивные алгоритмы для решения подобных задач нередко оказываются более простыми и ясными, чем их аналоги, основанные на циклах. В качестве поучительного упражнения можете попытаться реализовать алгоритм быстрой сортировки *без* рекурсии, чтобы оценить применение рекурсии по достоинству.

Многоликая рекурсия

Рекурсия нередко изображается как нечто таинственное и трудное для применения, что весьма прискорбно, поскольку она не заслуживает такой репутации. Б*о*льшую часть недоразумений, связанных с рекурсий, можно разрешить, рассмотрев различные аспекты рекурсии и их взаимосвязь. Можно разделить разные формы рекурсии по целому ряду категорий, таких как следующие [5].

- Выполнение: развернутая или описательная рекурсия.
- Завершение: ограниченная или неограниченная рекурсия.
- Область действия: прямая или косвенная рекурсия.

Отличие развернутой рекурсии от описательной и взаимосвязь этих разновидностей рекурсии через вычисление уже обсуждались ранее в этой главе. Напомним, что выполнение описательной рекурсии приводит к развернутой рекурсии, что может оказать помощь в осмыслении рекурсивной ситуации. С одной стороны, сталкиваясь с развернутой рекурсией, можно пытаться рассматривать описательную рекурсию, выполнение которой приведет к развернутой рекурсии. Описательная рекурсия зачастую предоставляет компактную характеристику

ситуации, в особенности если рекурсия еще и неограниченная. С другой стороны, если задана описательная рекурсия, то нередко оказывается полезным выполнить ее определение, чтобы увидеть ее развернутый вариант, особенно в тех случаях, когда рекурсия имеет более сложную форму, например когда она подразумевает несколько рекурсивных вхождений. Если снова обратиться к примеру изображения комнаты с телевизором, то какой результат получится, если добавить еще одну видеокамеру и телевизор, чтобы видеокамеры могли фиксировать проецируемое на экран изображение друг друга плюс изображения обоих телевизоров? Наблюдение за тем, как из описательной рекурсии получается развернутая рекурсия, помогает понять смысл рекурсии.

Ограниченной является рекурсия, которая завершается. Различать ограниченную и неограниченную рекурсии имеет смысл только для развернутой рекурсии. Но в связи с этим может возникнуть вопрос: какие требования следует предъявить к развернутой рекурсии, чтобы получить ограниченную рекурсию? Одно из условий состоит в том, чтобы описание рекурсии непременно содержало некоторую часть, которая не содержит вызов рекурсии, как, например, определения для `Goal(спасти доктора)` или `Count( )`. Такие уравнения называются ***базовыми случаями***. Несмотря на то что базовый случай требует завершения рекурсии, завершение рекурсии это не гарантирует, поскольку рекурсивные случаи могут оказаться такими, что базовый случай просто не будет достигнут. Напомним в этой связи определение алгоритма `Count(list)=Count(list)+1`, который никогда не завершится, если применить его к непустому списку, несмотря на наличие у него базового случая пустого списка.

Полезна ли вообще неограниченная рекурсия? Ведь если рекурсивные вычисления не завершаются, то они не в состоянии дать какие-либо результаты, а следовательно, они бессмысленны. Это вполне возможно, если рассматривать подобные вычисления обособленно, но как составляющие других вычислений неограниченные рекурсии могут быть весьма полезны. Допустим, что вычисление производит бесконечный поток случайных чисел. Такой поток может оказаться удобным для реализации различных моделей. При условии, что используется лишь конечная часть бесконечного потока, вычисление может вести себя вполне корректно, игнорируя бесконечные части данного потока. В качестве еще одного примера рассмотрим приведенное ниже определение бесконечного списка единиц, которое означает, что первым элементом этого списка является единица, а далее следует список единиц.

```
Ones = 1 → Ones
```

Выполнение этого определения приведет к бесконечной последовательности единиц:

```
1→1→1→1→1→1→ ...
```

Подобно изображению комнаты с телевизором, этот список содержит сам себя как свою часть. Это видно как из приведенного выше уравнения, так и из развернутого списка. Бесконечный список единиц начинается с единицы, а далее следует список единиц, который также бесконечен.

Такое представление о самовложенности помогает объяснить самоподобие как результат рекурсии. Если написать список, вычисленный с помощью алгоритма *Ones* (Единицы), в одной строке, а список, вычисленный с помощью алгоритма 1→*Ones*, строкой ниже, то станет очевидно, что оба списка совершенно одинаковы. А поскольку оба списка бесконечны, то список во второй строке не содержит дополнительного элемента.

Неограниченную рекурсию можно обнаружить в музыке, где имеются бесконечные песенки вроде *У попа была собака*, или в таких произведениях нидерландского художника М.К. Эшера, как *Рисующие руки* и *Картинная галерея* [6]. На литографии *Рисующие руки* одна рука рисует другую, которая, в свою очередь, рисует первую руку. Здесь отсутствует базовый случай, и поэтому рекурсия не завершается. Точно так же на литографии *Картинная галерея* показана бесконечная рекурсия — это изображение города с галереей, где человек рассматривает изображение того же самого города, включая и его самого в галерее, где он рассматривает картины.

Обе литографии Эшера наглядно показывают отличие прямой рекурсии от косвенной. На литографии *Рисующие руки* рекурсия оказывается косвенной, поскольку каждая рука рисует не себя, а другую руку, которой она рисуется. А на литографии *Картинная галерея* изображение содержит само себя непосредственно, поскольку на нем самом показан город с картинной галереей, где человек рассматривает картину. Кроме того, литография *Рисующие руки* демонстрирует, что косвенная рекурсия не гарантирует завершение. Эту ситуацию можно сравнить с базовыми случаями: они обязательны для завершения рекурсии, хотя и не гарантируют его.

Распространенным примером косвенной рекурсии служит определение алгоритмов *Even* (Четно) и *Odd* (Нечетно), позволяющих выяснить, делится ли положительное целое число на 2, т.е. является ли оно четным, как показано ниже. Так, определение алгоритма *Even* гласит, что 0 — это четное число, а любое другое число считается четным, если предшествующее ему число не четное. Во втором уравнении из определения алгоритма *Even* делается ссылка на алгоритм *Odd*. Определение алгоритма *Odd* означает, что 0 не является нечетным числом, а любое другое число считается нечетным, если предшествующее ему число четно. Во втором уравнении из определения алгоритма *Odd* делается ссылка на алгоритм *Even*.

```
Even(0) = true                Odd(0) = false
Even(n) = Odd(n - 1)          Odd(n) = Even(n - 1)
```

Таким образом, алгоритм *Even* рекурсивно обращается к самому себе косвенно — через обращение к алгоритму *Odd*, и наоборот. Это ясно видно из следующего простого примера вычисления:

```
Even(2) = Odd(2 - 1) = Odd(1) = Even(1 - 1) = Even(0) = true
```

Как видите, вызов `Even(2)` сводится к вызову `Even(0)`, но косвенно — через алгоритм *Odd*. Определения алгоритмов *Even* и *Odd* похожи на литографию ***Рисующие руки*** в том плане, что каждый из них определяет другой. Но главное их различие заключается в том, что рекурсия в них ограничена (любое вычисление завершится одним из базовых случаев) [7], тогда как рекурсия на литографии ***Рисующие руки*** не ограничена.

Еще одним примером прямой рекурсии служит следующее определение рекурсии в словарном стиле [8]:

Рекурсия [n], см. *Рекурсия.*[1]

Это лукавое определение включает в себя несколько важных составляющих рекурсивных определений, в частности, используя то, что определяется, в собственном определении (с помощью его имени). Незавершенность и бессмысленность такого "определения" отражает то поразительное ощущение, которое порой вызывают рекурсивные определения. В главе 13, "Все дело в интерпретации", представлены два способа разгадывания рекурсивных определений и придания им смысла.

---

[1] В словарном стиле возможна и косвенная рекурсия. Классическим ее примером является четырнадцатое путешествие Ийона Тихого (см. С. Лем, *Звездные дневники Ийона Тихого*), в котором статья "Сепульки" в энциклопедии отсылала читателя к статье "Сепулькарии", статья "Сепулькарии" — к статье "Сепуление", которая, как нетрудно догадаться, отсылала читателя к статье "Сепульки"... — *Примеч. ред.*

Состояние дел

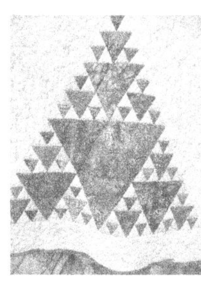
Итак, вы сделали всю свою дневную работу и возвращаетесь домой. Перед ужином у вас есть немного времени, чтобы заняться своим швейным хобби. Выбранный вами лоскутный узор определяет, какие ткани и в каком количестве нужны. А поскольку вы приступили к этому проекту несколько недель назад, то все ткани приобретены, разрезаны, б*о*льшая часть лоскутов разглажена, а часть из них даже сшита.

Лоскутный узор и соответствующие инструкции по его стежке составляют определенный алгоритм. Для создания лоскутной аппликации требуется немало труда, и завершить всю работу в один присест не удастся. Поэтому данный алгоритм придется неоднократно прерывать, чтобы продолжить его выполнение позже. Несмотря на все прилежание и внимание, которые требуются для создания такого лоскутного одеяла, его сшивание стежкой прервать, а позднее возобновить необычайно легко, поскольку состояние процесса сшивания стежкой на каждой его стадии идеально представимо достигнутыми результатами. Если у вас нет подходящей ткани, вы должны ее докупить, если у вас уже есть все подобранные ткани, но из них еще не вырезаны лоскуты — их нужно вырезать, и т.д. Нечто подобное встречается и в других "ремесленных" проектах, например при изготовлении скворечника или складывании фигур оригами. Однако имеются и такие задачи, которые требуют дополнительных усилий для того, чтобы представить текущее состояние вычисления. Допустим, вы подсчитываете количество бейсбольных карточек в коробке с коллекционируемыми предметами и ваше занятие прерывает телефонный звонок. Чтобы продолжить подсчет, вам нужно отделить подсчитанные карточки от еще не подсчитанных и запомнить количество уже подсчитанных к этому моменту.

Промежуточные результаты могут служить не только для обеспечения возможности перерывов в работе, но и для пояснения вычислений, выполняемых согласно алгоритму, поскольку в них отслеживаются вычислительные шаги и что было сделано до текущего момента. Такая трассировка задается последовательностью вычислительных состояний, которые начинаются с простого элемента (например, подборки тканей для лоскутного шитья или простого листа бумаги для оригами), а затем в них постепенно накапливаются все более точные аппроксимации конечного результата. Каждая последующая аппроксимация получается из предыдущей внесением в нее изменений, как описано в алгоритме, определяющем вычисление. Последовательность, состоящая из первоначального элемента, всех промежуточных шагов и конечного результата, является *следом* (trace) вычисления. Подобно тому, как следы ног на песке указывают, как двигался человек

из одного места в другое, след вычисления поясняет процесс преобразования первоначального элемента в конечный результат.

Построение следа оказывается эффективным методом и для понимания рекурсивного описания. Несмотря на то что большинство лоскутных узоров не рекурсивны, среди них все же можно найти очаровательные образцы, например узоры, составленные из треугольников Серпинского. Изображение наглядно показывает рекурсию, на которой базируется узор. Основной треугольник узора состоит из трех светлых треугольников вокруг перевернутого темного треугольника. Каждый из этих светлых треугольников, в свою очередь, состоит из трех более мелких светлых треугольников вокруг более мелкого перевернутого темного треугольника. Каждый из более мелких светлых треугольников, в свою очередь, состоит из..., и т.д. Следует обратить внимание на сходство этого узора с косвенной рекурсией на литографии *Рисующие руки* Эшера и упоминавшимися ранее алгоритмами `Even` и `Odd` (глава 12, “Своевременный стежок вычисляется впрок”).



Имеются разные виды следов. В некоторых случаях алгоритм полностью отделен от следа. Например, многие описания сборок состоят из пронумерованных пунктов, описывающих порядок действий, а также из отдельного, соответственно пронумерованного ряда рисунков, на которых показаны результаты выполнения действий, указанных в пунктах. Но имеются и такие следы, которые состоят из рисунков с непосредственно указанными на них инструкциями. У обоих видов следов имеются свои достоинства и недостатки, и особенно это касается следов рекурсивных алгоритмов. Трудности выполнения рекурсивных алгоритмов состоят, например, в необходимости отслеживать самые разные виды выполнения и разные его параметры. Например, выполнение алгоритма `Count(B→W→H)` приводит к тому, что алгоритм `Count` выполняется еще три раза с разными списками в качестве параметров. С этим вполне можно справиться без дополнительной помощи, используя инструкции как часть следов. Отдельные шаги следа служат лишь представлениями вычисления и содержат все сведения, требующиеся для того, чтобы продолжить вычисление. Тем не менее инструкции в следе могут быть запутанными, а кроме того, подобный метод может привести к огромным следам, содержащим немало лишнего и чрезмерно много моментальных снимков. Другой метод состоит в том, чтобы хранить инструкции отдельно от следов и тем самым поддерживать соответствие между алгоритмом и следом, но в то же время получать более лаконичные следы.

Смысловое значение алгоритма задается множеством всех вычислений, которые он способен производить [1]. Следы делают вычисления конкретными, а потому способствуют лучшему пониманию алгоритмов. Таким образом, методы получения следов являются важными средствами, позволяющими пролить свет на взаимосвязь между алгоритмами и их вычислениями.

13 Все дело в интерпретации

Основное внимание в главе 12, “Своевременный стежок вычисляется впрок”, было сосредоточено на разъяснении сущности рекурсии, ее различных форм и связи с циклами. На примерах алгоритмов `ToDo` и `Count` было продемонстрировано, что выполнение описательной рекурсии приводит к развернутой рекурсии, обнаруживая взаимосвязь между рекурсивной ссылкой и самоподобием как вычислением. Но мы еще не рассмотрели подробно, каким образом действует рекурсивное вычисление.

В этой главе наглядно показано, каким образом могут выполняться рекурсивные алгоритмы. Одна из занимательных сторон этого вопроса состоит в том, что выполнение алгоритма приводит (через рекурсию) к неоднократному выполнению того же самого алгоритма. Динамическое поведение рекурсивных алгоритмов можно продемонстрировать двумя методами.

Во-первых, с помощью *подстановки* можно построить следы вычисления из рекурсивных определений. Подстановка аргумента вместо параметра является основополагающим действием, вызываемым всякий раз, когда выполняется алгоритм. Подстановка дополнительно применяется для выполнения рекурсивного алгоритма, чтобы заменить вызов алгоритма его определением. Подобным способом подстановка позволяет исключить описательную рекурсию и преобразовать ее в след, который может служить в качестве пояснения рекурсивного алгоритма.

И во-вторых, понятие *интерпретатора* предоставляет альтернативный метод пояснить рекурсивный алгоритм. Интерпретатор — это особого рода компьютер, выполняющий алгоритмы с помощью стека (см. главу 4, “Записная книжка

сыщика") для отслеживания рекурсивных (и нерекурсивных) вызовов и многих копий аргументов, возникающих вследствие выполнения алгоритма. Механизм действия интерпретатора более сложный, чем механизм действия подстановки, но он дает иную точку зрения на выполнение рекурсивных алгоритмов. Кроме того, интерпретатор способен получать следы вычислений, которые оказываются более простыми, чем те, которые формируются подстановкой, поскольку они содержат только данные и никаких инструкций. Помимо пояснения механизма действия рекурсии, обе эти модели помогают разъяснить еще одну сторону рекурсии: отличие линейной рекурсии от нелинейной.

Переписывание истории

Алгоритм как средство решения задач представляет интерес лишь в том случае, если он способен решать ряд сходных задач (см. главу 2, "От слов к делу: когда действительно происходит вычисление"). Если бы конкретный алгоритм мог решать лишь одну задачу, например поиск кратчайшего пути из дома на работу, можно было бы выполнить его лишь один раз, а затем запомнить маршрут и забыть о самом алгоритме. А если алгоритм параметризирован и способен находить кратчайшие маршруты между разными интересующими нас местами, он становится весьма полезным, поскольку применяется во многих случаях.

Когда алгоритм выполняется, результирующее вычисление обрабатывает входные значения, *подставляемые* вместо параметров. Так, упоминавшийся в главе 2 алгоритм подъема по утрам состоит из инструкции "Подъем в *время подъема*". Чтобы выполнить этот алгоритм, необходимо предоставить конкретное время, установив, например, будильник на 6:30 утра, и тогда благодаря подстановке значения *6:30 утра* вместо параметра *время подъема* инструкция станет следующей: "Подъем в 6:30 утра".

Механизм подстановки применяется ко всем алгоритмам и их параметрам, включая количество чашек воды для заваривания кофе, камешки для поиска пути, погодные условия для сводки погоды и т.д. Разумеется, подстановка параметров применяется и к рекурсивным алгоритмам. Например, для быстрой сортировки или же сортировки слиянием в качестве входных данных требуется сортируемый список, а для бинарного поиска — два параметра: искомый элемент и структура данных (дерево или массив), в которой следует произвести поиск; тогда как алгоритм `Count` (см. главу 12, "Своевременный стежок вычисляется впрок") в качестве входных данных для своего параметра принимает список, для которого подсчитывается количество элементов.

Еще одной разновидностью подстановки, играющей роль в выполнении рекурсивных алгоритмов, является подстановка определения алгоритма вместо его наименования, как демонстрировалось ранее в примере вычисления с помощью

алгоритма `Count` количества предметов, которые Марти взял с собой в путешествие во времени в 1885 год. Рассмотрим еще раз уравнение, определяющее порядок обработки непустых списков по алгоритму `Count`.

```
Count(x→rest) = Count(rest) + 1
```

Прежде всего, список в качестве аргумента подставляется в этом уравнении вместо параметра. Так, выполнение алгоритма `Count` для списка `B→W→H` означает, что этот список подставляется вместо параметра в алгоритм `Count`. А поскольку этот параметр представлен в уравнении алгоритма `Count` в виде шаблона, состоящего из двух частей, то в процессе сопоставления списка с данным шаблоном производятся следующие две подстановки: значение `B` вместо параметра `x` и значение `W→H` вместо параметра `rest`. Подстановка оказывает влияние на правую сторону данного уравнения, в котором определяются шаги алгоритма — в данном случае `rest` обрабатывается по алгоритму `Count`, а к полученному в итоге числу прибавляется единица. Это приводит к следующему уравнению:

```
Count(B→W→H) = Count(W→H) + 1
```

Это уравнение можно понять как определение алгоритма, экземпляр которого получен для конкретного случая. Но его можно также рассматривать как подстановку определения алгоритма вместо его вызова.

Это становится яснее, если применить обозначение выводов, впервые упоминавшихся в главе 8, “Сквозь призму языка”. Напомним, что стрелка указывает, как нетерминальный символ в соответствии с определением расширяется до правой части грамматического правила. Последовательность таких расширений может быть использована для вывода символьной строки или синтаксического дерева по грамматическим правилам языка. Уравнения, определяющие рекурсивные алгоритмы, могут быть представлены таким же образом, как и правила для вывода вычисления. Так, используя обозначение стрелкой, предыдущее уравнение можно переписать следующим образом:

$$\texttt{Count(B→W→H)} \xrightarrow{Count_2} \texttt{Count(W→H) + 1}$$

Обозначение стрелкой подчеркивает, что выражение `Count(W→H)+1` появляется в результате замены или *подстановки* определения алгоритма `Count` вместо его вызова. А метка $Count_2$ над стрелкой означает, что для этой цели было использовано второе уравнение из определения алгоритма `Count`. Но поскольку результат содержит вызов алгоритма `Count`, данную стратегию можно применить снова, подставив определение этого алгоритма, но с новым списком `W→H` в качестве аргумента вместо его параметра. И в этом случае придется воспользоваться вторым уравнением, поскольку в качестве аргумента подставляется непустой список, как показано ниже.

$$\texttt{Count(B→W→H)} \xrightarrow{Count_2} \texttt{Count(W→H) + 1} \xrightarrow{Count_2} \texttt{Count(H) + 1 + 1}$$

Этот последний шаг показывает, что подстановки, как правило, происходят в контексте, не оказывающем влияния на саму подстановку. Иными словами, подстановка заменяет часть более крупного выражения и вносит изменения лишь локально. Это можно сравнить с заменой старой лампочки на новую без замены патрона и остальной части электропроводки. В данном примере подстановка выражения `Count(H)+1` вместо вызова `Count(W→H)` происходит в контексте "+1". Чтобы завершить расширение и удалить все рекурсивные вхождения алгоритма `Count`, требуются еще два шага подстановки, как показано ниже.

```
Count(B→W→H)  --Count2-->  Count(W→H)+1
              --Count2-->  Count(H)+1+1
              --Count2-->  Count()+1+1+1
              --Count1-->  0+1+1
```

Обратите внимание на то, что на последнем шаге подстановки используется первое уравнение из определения алгоритма `Count`, которое применяется к пустому списку. А теперь, когда рекурсия исключена полностью и получено арифметическое выражение, можно его вычислить и получить результат.

Можно применить ту же стратегию к алгоритмам `ToDo` и `Goal` и воспользоваться подстановкой, чтобы проследить выполнение рекурсивного алгоритма путешествия во времени. В итоге получится такая последовательность действий:

```
Goal(жить сейчас)
  --Goal1--> Goal(восстановить мир); пойти в турпоход
  --Goal2--> отнять альманах; Goal(спасти доктора); пойти в турпоход
  --Goal3--> отнять альманах; спасти доктора от Бьюфорда Таннена;
               пойти в турпоход
```

Эти примеры показывают, как благодаря повторяющейся подстановке определения вместо наименования раскрывается кажущаяся таинственной рекурсивная ссылка в описательной рекурсии. Повторяющаяся подстановка определений оказывается пригодной даже для рекурсивного изображения комнаты с телевизором, как показано на рис. 13.1.

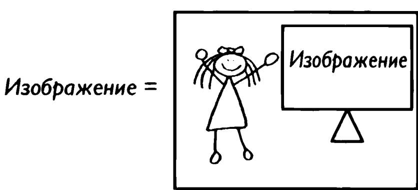
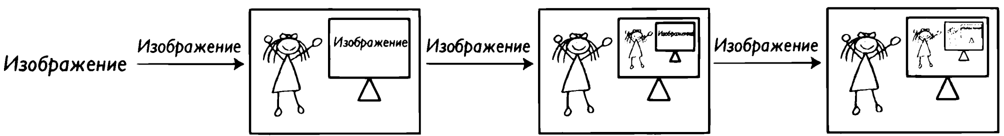


Рис. 13.1. Определение рекурсивного изображения. Имя, присвоенное изображению, содержит это имя, а значит, обращается к нему. Смысловое значение такого самоссылающегося определения может быть получено путем повторяющейся подстановки уменьшенной копии изображения вместо его наименования, и в итоге поэтапно получается развернутая рекурсия

Ниже приведено несколько первых шагов, наглядно показывающих, каким образом повторяющаяся подстановка преобразует описательную рекурсию в развернутую.



Разумеется, этот процесс подстановки бесконечен, поскольку рекурсия ничем не ограничена и не имеет базового случая. Эта ситуация сродни определению бесконечного списка единиц:

```
Ones = 1→Ones
```

При выполнении этого определения подстановка производит непрерывно расширяющийся список. На каждом шаге в этот список добавляется единица, как показано ниже.

$\texttt{Ones} \xrightarrow{\texttt{Ones}} \texttt{1}\rightarrow\texttt{Ones} \xrightarrow{\texttt{Ones}} \texttt{1}\rightarrow\texttt{1}\rightarrow\texttt{Ones} \xrightarrow{\texttt{Ones}} \texttt{1}\rightarrow\texttt{1}\rightarrow\texttt{1}\rightarrow\texttt{Ones} \ \ldots$

Процесс повторяющейся подстановки определений вместо наименований называется *перезаписью*. Поэтому, рассматривая путешествия Марти во времени как вычисления, можно сказать, что он действительно переписывает историю.

Уменьшение следов

Подстановка является простым механизмом для получения следов вычисления, которые, по существу, являются моментальными снимками промежуточных результатов или состояний. И хотя подстановка равно пригодна и для рекурсивных, и для нерекурсивных алгоритмов, в особенности она полезна для рекурсивных алгоритмов, поскольку исключает обращение алгоритма к самому себе и систематически преобразует описательную рекурсию в соответствующую развернутую. Если нас интересует только результат вычисления, а не его промежуточные шаги, то подстановка делает для этого даже больше, чем требуется. Но настоящая ценность следа заключается в том, что он способен предоставить пояснение произведенного вычисления. Характерным тому примером служит след вычисления по алгоритму `Count`.

Но несмотря на то, что след подстановки способен прояснить ситуацию, он может и мешать, отвлекая внимание, если становится слишком крупным. Обратимся снова к примеру алгоритма сортировки вставками, упоминавшегося в главе 6, “Сортировка алгоритмов сортировки”. Ниже приведено рекурсивное определение алгоритма `Isort`, который использует два списка: один — с еще не отсортированными элементами, а другой — с уже отсортированными элементами.

```
Isort( ; list)        = list
Isort(x→rest; list)   = Isort(rest; Insert(x; list))

Insert(w; )           = w
Insert(w; x→rest)     = if w ≤ x then w→x→rest
                        else x→Insert(w; rest)
```

У алгоритма `Isort` имеются два аргумента. Он обходит первый список и выполняет вспомогательный алгоритм `Insert` для обработки каждого элемента списка. Если список сортируемых элементов пуст, то сортировка больше не требуется, а второй параметр `list` содержит окончательный результат. В противном случае алгоритм `Insert` перемещает элемент *w* из неотсортированного списка на нужное место в отсортированном списке; если этот список пуст, то элемент *w* сам составляет результирующий отсортированный список. В противном случае алгоритм `Insert` сравнивает элемент *w* с первым элементом (*x*) того списка, в который он вносится. Если элемент *w* оказывается меньше или равен элементу *x*, значит, нужное место в списке найдено, и элемент *w* располагается в начале списка. В противном случае алгоритм `Insert` сохраняет элемент *x* на месте и пытается внести элемент *w* в остальную часть списка `rest`. Безусловно, все это работает лишь в том случае, если список, в который вводится элемент, уже отсортирован, что, конечно же, так и есть — потому что список построен исключительно с использованием алгоритма `Insert`.

Как показано на рис. 13.2, для фактического составления отсортированного списка требуется немало шагов, а результаты различных выполнений алгоритма `Insert` частично замаскированы наличием условной конструкции и тем фактом, что промежуточные списки временно представлены *дважды* в ветвях этой конструкции. И хотя след, полученный при подстановке, точно показывает, что именно делает алгоритм, чтобы разобрать его подробно, отделив данные от инструкций, требуется немало внимания и усердия.

Еще одна особенность метода подстановки, способная вызвать недоразумение, состоит в том, что зачастую возможны разные подстановки. И хотя выбор конкретной подстановки обычно не оказывает влияния на конечный результат, он все же может сказаться на размере следа и его понятности. Так, на рис. 13.2 показано, что первая подстановка приводит к вызову `Isort(W→H;Insert(B; ))`, к которому применимы две подстановки: можно применить первое уравнение для алгоритма `Insert` или второе уравнение для алгоритма `Isort`.

В главе 6, "Сортировка алгоритмов сортировки", был показан след, содержащий только данные, чтобы продемонстрировать разные алгоритмы сортировки. В частности, для каждого перемещенного элемента были показаны только неотсортированный и отсортированный списки (пример сортировки вставками приведен на рис. 6.2). Если применить то же самое наглядное представление в данном примере, то получится показанный ниже след, намного более короткий и лаконичный, чем приведенный на рис. 13.2.

$$
\begin{aligned}
Isort(\mathtt{B} \to \mathtt{W} \to \mathtt{H},\ ) &\xrightarrow{Isort_2} Isort\ (\mathtt{W} \to \mathtt{H}, Insert(\mathtt{B},\ ))\\
&\xrightarrow{Insert_1} Isort(\mathtt{W} \to \mathtt{H}, \mathtt{B})\\
&\xrightarrow{Isort_2} Isort(\mathtt{H}, Insert(\mathtt{W}, \mathtt{B}))\\
&\xrightarrow{Insert_2} Isort(\mathtt{H}, \mathbf{if}\ \mathtt{W} \le \mathtt{B}\ \mathbf{then}\ \mathtt{W} \to \mathtt{B}\ \mathbf{else}\ \mathtt{B} \to Insert(\mathtt{W},\ ))\\
&\xrightarrow{\mathbf{else}} Isort(\mathtt{H}, \mathtt{B} \to Insert(\mathtt{W},\ ))\\
&\xrightarrow{Insert_1} Isort(\mathtt{H}, \mathtt{B} \to \mathtt{W})\\
&\xrightarrow{Isort_2} Isort(\ , Insert(\mathtt{H}, \mathtt{B} \to \mathtt{W}))\\
&\xrightarrow{Isort_1} Insert(\mathtt{H}, \mathtt{B} \to \mathtt{W})\\
&\xrightarrow{Insert_2} \mathbf{if}\ \mathtt{H} \le \mathtt{B}\ \mathbf{then}\ \mathtt{H} \to \mathtt{B} \to \mathtt{W}\ \mathbf{else}\ \mathtt{B} \to Insert(\mathtt{H}, \mathtt{W})\\
&\xrightarrow{\mathbf{else}} \mathtt{B} \to Insert(\mathtt{H}, \mathtt{W})\\
&\xrightarrow{Insert_2} \mathtt{B} \to (\mathbf{if}\ \mathtt{H} \le \mathtt{W}\ \mathbf{then}\ \mathtt{H} \to \mathtt{W}\ \mathbf{else}\ \mathtt{W} \to Insert(\mathtt{H}, rest))\\
&\xrightarrow{\mathbf{then}} \mathtt{B} \to \mathtt{H} \to \mathtt{W}
\end{aligned}
$$

Рис. 13.2. След подстановки для выполнения сортировки вставками

Неотсортированный список	*Отсортированный список*
B→W→H	|
W→H	| **B**
H	| **B→W**
	| **B→H→W**

След, содержащий только данные, выглядит намного проще, поскольку не содержит ни одной инструкции алгоритма. (Это раскрывает механизм действия интерпретатора, сохраняющего описание алгоритма или программы отдельно от данных.) Также, след, содержащий только данные, показывает только результаты обработки двух списков алгоритмом `Isort`, опуская подробности перемещения элементов во втором списке алгоритмом `Insert`. Кроме того, программа представлена лишь один раз и больше не изменяется. А когда алгоритм или программа интерпретируется, то след, содержащий только данные, представляет лишь данные в их развитии.

Если в методе подстановки след выполняет двойную обязанность, отслеживая развитие данных и ход вычисления, то в интерпретаторе для каждой из этих двух задач применяется стек. В частности, для рекурсивного алгоритма в интерпретаторе необходимо отслеживать место, где делается каждый рекурсивный вызов, чтобы можно было вернуться туда и продолжить вычисление по завершении рекурсивного вызова. А поскольку у каждого рекурсивного выполнения алгоритма имеется свой аргумент, то интерпретатор должен также поддерживать несколько вариантов параметров.

Все эти требования можно удовлетворить, сохраняя адреса программ и значения параметров в стеке. Рассмотрим это на примере выполнения алгоритма *ToDo*. Чтобы упростить возврат из рекурсивных вызовов, придется пометить номерами соответствующие позиции в алгоритме. Ни в одном из определений алгоритма *ToDo* его параметр не используется, поэтому им можно пренебречь, сохранив только позиции в стеке. Данный алгоритм воспроизводится в несколько измененном виде: номерами помечены позиции между инструкциями, как показано ниже.

```
ToDo(1985) = ❶ ToDo(1955) ❷ пойти в поход ❸ вернуться
ToDo(1955) = ❹ уничтожить альманах ❺ ToDo(1885) ❻ вернуться
ToDo(1885) = ❼ спасти доктора ❽ вернуться
```

Интерпретатор выполняет алгоритм, применяемый к заданному аргументу, например, *ToDo*(1985), путем выполнения инструкций одна за одной, сохраняя адреса возврата в стеке. Всякий раз, когда инструкция требует рекурсивного выполнения алгоритма, следующий после нее адрес размещается на вершине стека, прежде чем интерпретатор перейдет к инструкции, указанной в рекурсивном вызове. В данном примере первая инструкция состоит в том, чтобы совершить переход, который приводит к размещению следующего после нее адреса ❷ в стеке и выполнению последующей инструкции ❹.

Это наглядно показано в первых двух строках таблицына рис. 13.3, на котором демонстрируется состояние отдельной инструкции и стека по ходу выполнения алгоритма. Во втором столбце таблицы отображается состояние стека, его вершина — слева, а дно — справа. На этом рисунке показано, как часть окружающего мира изменяется по ходу выполнения инструкций, например год или владение спортивным альманахом. Эти конкретные сведения об окружающем мире изменились после того, как Марти уничтожил спортивный альманах в 1955 году. Следует, однако, иметь в виду, что доктор Браун находится в опасности, что не является следствием выполнения текущего алгоритма. Но изменение является частью последующих шагов алгоритма. После путешествия в 1885 год, приведшего к размещению в стеке очередного адреса возврата ❻, действие по спасению доктора Брауна изменяет сведения о том, что он в опасности. Следующей инструкцией в алгоритме является возврат туда, где алгоритм был оставлен, когда был совершен рекурсивный переход. Цель этого возврата — перейти по последнему адресу, находящемуся на вершине стека. Следовательно, инструкция *вернуться* вызывает переход к следующей инструкции по адресу ❻, т.е. к очередной инструкции *вернуться*, извлекающей адрес возврата из стека. Тем самым обнаруживается целевой адрес ❷ для следующей инструкции *вернуться* к тому моменту, когда Марти и Дженнифер смогут, наконец-то, пойти в турпоход.

Текущая инструкция	*Стек*	*Состояние мира*
❶ *ToDo*(1995)	—	*Год*: 1985, *альманах у Биффа, жуть в 1985*
❹ *уничтожить альманах*	❷	*Год*: 1955, *альманах у Биффа, жуть в 1985*
❺ *ToDo*(1885)	❷	*Год*: 1955, *доктор в опасности в 1885*
❼ *спасти доктора*	❻ ❷	*Год*: 1885, *доктор в опасности в 1885*
❽ *вернуться*	❻ ❷	*Год*: 1885
❻ *вернуться*	❷	*Год*: 1955
❷ *пойти в поход*	—	*Год*: 1985

Рис. 13.3. Интерпретация вызова `ToDo(1985)`. *Если текущая инструкция является рекурсивным вызовом, то следующий после нее адрес запоминается в стеке, чтобы вычисление можно было продолжить после ее завершения. Это происходит всякий раз, когда встречается инструкция* `вернуться`. *После возврата назад адрес удаляется из стека*

Пример алгоритма *ToDo* наглядно показывает, каким образом вложенные вызовы алгоритма приводят к возврату адресов, сохраняемых в стеке, хотя он и не требует хранения значений параметров в стеке. Чтобы продемонстрировать эту особенность интерпретатора, снова рассмотрим алгоритм *Isort*, в котором требуется хранить значения параметров в стеке, но в то же время не требуется хранить адреса возврата, поскольку каждый алгоритмический шаг задается лишь одним выражением, а не последовательностью, состоящей из нескольких выражений, как это имеет место в алгоритме *ToDo*.

Чтобы отсортировать список предметов, которые Марти берет с собой в путешествие во времени, интерпретатор начинает вычислять вызов `Isort(B→W→H; )` с пустым стеком. Сопоставление аргументов с шаблонами параметров алгоритма *Isort* приводит к *связываниям* (binding), сопоставляющим имена параметров из шаблона со значениями. Эти связывания размещаются в стеке, содержимое которого приведено на рис. 13.4, *а*. Может показаться странным, что в результате применения алгоритма *Isort* к непустому списку в качестве первого аргумента входных данных в стеке получаются связывания трех параметров, хотя алгоритм *Isort* принимает лишь два параметра. Дело в том, что первый аргумент сопоставляется с шаблоном `x→rest`, содержащим два параметра, предназначенных для разделения списка аргументов на две части: первый элемент и хвост списка. Первое уравнение дает связывание лишь одного параметра, поскольку известно, что в качестве первого аргумента входных данных задается пустой список, к которому не требуется обращаться по имени.

После сопоставления с шаблоном, которое обеспечивает связывания параметров в стеке, алгоритм выдает инструкцию вычислить выражение `Isort(rest; Insert(x; list))`. Как и в методе подстановки, у интерпретатора теперь имеются два дальнейших возможных пути: продолжить внешний вызов

алгоритма *Isort* или сначала обработать вложенный вызов алгоритма *Insert*. В большинстве языков программирования аргументы вычисляются до выполнения алгоритма [1]. Придерживаясь именно этой стратегии, интерпретатор вычисляет выражение *Insert*(*x*;*list*), прежде чем обрабатывать вызов *Isort*. Значения параметров *x* и *list* могут быть извлечены из стека, что приведет к вычислению выражения *Insert*(B;). Это вычисление может быть выполнено с помощью отдельного стека и дает в итоге список B. Это означает, что вычисление вызова *Isort* превращается в вызов *Isort*(*rest*;B).

Интерпретатор вычисляет вызов *Isort*(*rest*;B) с помощью стека, показанного на рис. 13.4, *а*. Сначала значение параметра *rest* извлекается из стека, а это означает, что интерпретатор фактически вычисляет вызов *Isort*(W→H;B). Затем сопоставление с шаблоном приводит к новым связываниям параметров алгоритма *Isort*, размещаемым в стеке, как показано на рис. 13.4, *б*.

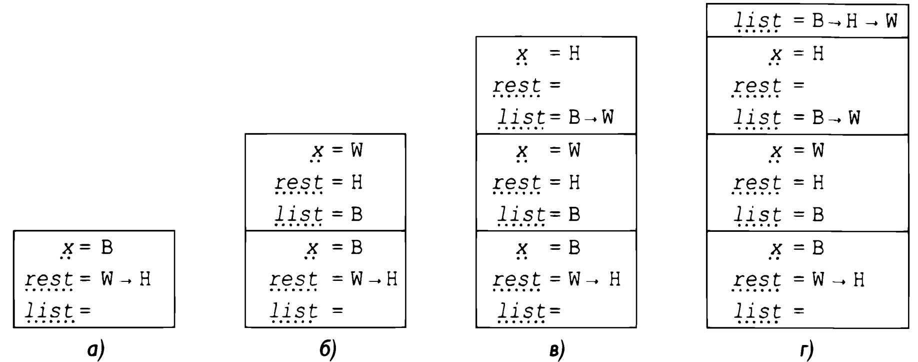


Рис. 13.4. Моментальные снимки содержимого стека в процессе интерпретации вызова Isort(W→H;B)

Кажется странным, что параметры *x*, *rest* и *list* появляются в стеке дважды. Дело в том, что рекурсивный вызов алгоритма *Isort* оперирует иными аргументами для своих параметров, а следовательно, для их хранения требуется отдельное место. Как только вложенный вызов алгоритма *Isort* будет вычислен интерпретатором, связывания с параметрами удаляются (извлекаются) из стека, и вычисление предыдущего вызова алгоритма *Isort* может продолжаться — его аргументы доступны интерпретатору [2].

Но прежде чем вызов алгоритма *Isort* завершится, он вновь приводит к выполнению второго уравнения для алгоритма *Isort*, требуя вычисления выражения *Isort*(*rest*;*Insert*(*x*;*list*)). Связывания параметров снова оказываются в стеке. Но поскольку с именем каждого параметра имеется несколько связываний, то возникает вопрос: какие из них следует выбрать и как их найти. Здесь снова вступает в действие тип данных стека. Значения параметров для последнего вызова алгоритма *Isort* были размещены в стеке последними, и поэтому они

находятся на вершине стека, что приводит к вызову *Isort*(H; *Insert*(W;B)). Поскольку в результате вызова *Insert*(W;B) получается список B→W, следующий вычисляемый вызов *Isort*(H;B→W) снова задействует второе уравнение для *Isort* и приводит к другому вызову *Isort*(rest; *Insert*(x; list)) со стеком, состояние которого показано на рис. 13.4, *в*.

Получая аргументы для параметров на вершине стека, этот вызов, в свою очередь, ведет к вычислению вызова *Isort*(; *Insert*(H,B→W)), а он, в свою очередь, дает *Isort*(;B→H→W) после вычисления вызова *Insert*(H;B→W). А поскольку первым аргументом алгоритма *Isort* теперь оказывается пустой список, интерпретатор получает параметр list для вычисления по первому уравнению алгоритма *Isort*. Это вычисление выполняется в контексте стека, приведенного на рис. 13.4, *г*.

В этот момент значение параметра list находится в стеке и возвращается в качестве окончательного результата. Наконец, завершение каждого вызова алгоритма *Isort* сопровождается удалением связываний его параметров из стека, опустошая последний. Таким образом, мы можем наблюдать, как стек растет с каждым (рекурсивным) вызовом и сокращается по окончании вызова.

Упоминавшийся ранее след с двумя списками для сортировки вставками может быть систематически реконструирован из стеков, приведенных на рис. 13.4. В действительности для этой цели достаточно стека, представляющего полное вложение всех вызовов алгоритма *Isort* (рис. 13.4, *г*). В частности, каждая группа связываний в стеке производит один шаг следа. Напомним, что каждый непустой список при вызове алгоритма *Isort* разделяется для получения связываний двух параметров — x и rest. Это означает, что входной список для первых трех вызовов алгоритма *Isort* задается списком x→rest, а выходной список — параметром list. Для последнего вызова алгоритма *Isort* входной список не представлен параметром, поскольку он уже пуст, тогда как выходной список представлен параметром list. Таким образом, на дне стека оказывается пара списков, состоящая из списка B→W→H и пустого списка, во втором элементе стека — списки W→H и B, в третьем элементе стека — списки H и B→W, а на вершине стека — пустой список (входной, не связанный с параметром), а также список B→W→H.

Подстановка и интерпретация являются двумя способами, позволяющими разбираться в выполнении алгоритмов вообще и рекурсивных в частности. Подстановка проще, поскольку она оперирует следом, перезаписывающимся шаг за шагом, тогда как для интерпретации применяется вспомогательный стек. Подстановка смешивает код и данные, в то время как интерпретация четко разделяет их, упрощая извлечение простых следов. В случае неограниченной рекурсии подстановка производит нечто полезное, тогда как интерпретация в этом случае не завершается.

Двойные двойники удваиваются

Когда Марти возвращается в 1955 год во второй раз (после возвращения с доктором Брауном из 2015 года в жуткий 1985 год), он оказывается в 1955 году *дважды*, поскольку прошлое, в которое он отправляется, является тем прошлым, где он уже бывал прежде (случайно — в первом фильме кинотрилогии *Назад в будущее*). Но оба Марти — это два разных, не взаимодействующих друг с другом субъекта. Первый Марти пытается добиться, чтобы его родители влюбились, тогда как второй Марти пытается отнять спортивный альманах у Биффа. Аналогично старый Бифф, который отправляется из 2015 года в 1955 год, чтобы передать себе молодому спортивный альманах, также оказывается в 1955 году дважды. Но в отличие от обоих Марти, оба Биффа взаимодействуют друг с другом. В частности, старый Бифф передает молодому Биффу спортивный альманах. Правда, пространственно-временного парадокса, который мог бы привести к уничтожению вселенной, при этом не возникает [3]. Более того, машина времени, с помощью которой Марти возвращается из 1985 года в 1955 год, оказывается той же самой, что у и старого Биффа, отправляющегося из 2015 года в 1955 год. Таким образом, к тому моменту, когда Марти наблюдает за тем, как старый Бифф передает спортивный альманах молодому Биффу, обе копии машины времени должны находиться в 1955 году. На самом деле в 1955 году должны одновременно присутствовать *три* машины времени, поскольку первый Марти также прибыл в 1955 год с ее помощью.

Данный пример демонстрирует, что неизбежное следствие путешествий во времени в прошлое — наличие дубликатов путешествующих во времени объектов и людей (по крайней мере, когда несколько путешествий в прошлое происходит одновременно). Нечто подобное происходит и с рекурсией. Поскольку второе уравнение алгоритма `Count` содержит только одно рекурсивное упоминание `Count`, оно будет создавать только по одному экземпляру в любой момент прошлого, поскольку каждый рекурсивный вызов отправляется на одну единицу времени в прошлое относительно момента времени, когда этот вызов произошел. Такая форма рекурсии, в которой ссылка на определяемый алгоритм делается в его определении лишь один раз, называется *линейной рекурсией*. В алгоритмах линейная рекурсия приводит к изолированным вызовам алгоритма в прошлом. Линейная рекурсия может быть легко преобразована в циклы, но в общем случае не дает возможности параллельного выполнения.

И напротив, если обращение к определяемому алгоритму выполняется в его определении неоднократно, возникает рекурсия, именуемая *нелинейной*. Поскольку все вызовы в этом случае происходят одновременно, соответствующее их выполнение в прошлом начинается одновременно, а следовательно, они выполняются параллельно. Но это совсем не означает, что компьютер (электронный или

человек) *должен* действительно выполнять вызовы параллельно. Это лишь означает, что они *могут быть* выполнены параллельно. И это является поразительной особенностью грамотно разработанных алгоритмов “разделяй и властвуй”. Они не только быстро разделяют задачу на части, чтобы ее можно было решить за несколько шагов, но и поддерживают параллельное выполнение на многих компьютерах.

Двумя примерами подобных алгоритмов служат быстрая сортировка, а также сортировка слиянием (см. главу 6, “Сортировка алгоритмов сортировки”). Быстрая сортировка определяется так, как показано ниже. В первом уравнении указывается, что пустой список уже отсортирован, а во втором уравнении — что для сортировки непустого списка необходимо взять из его хвоста (параметр `rest`) все элементы, которые оказываются меньше элемента `x`, отсортировать их и расположить перед элементом `x`. Аналогично все элементы, которые оказываются больше или равны элементу `x`, следует отсортировать и расположить после элемента `x`.

```
Qsort( )        =
Qsort(x→rest)  = Qsort(Smaller(rest;x))→x→Qsort(Larger(rest;x))
```

Обратите внимание на то, что во втором уравнении обнаруживается нелинейный характер рекурсии в алгоритме `Qsort`. Этот алгоритм выполняется лучше всего, когда его параметр `x` позволяет регулярно разбивать хвост списка на два подсписка приблизительно равной длины. Этого может не произойти в наихудшем случае, когда список уже (или почти) отсортирован, но в среднем быстрая сортировка работает очень хорошо.

Было бы любопытно выполнить быструю сортировку или же сортировку слиянием с помощью группы людей. Чтобы выполнить быструю сортировку, все люди выстраиваются в очередь, и первый человек начинает сортировку, применяя второе из приведенных выше уравнений, чтобы разделить список элементов на два подсписка в зависимости от того, оказываются ли они меньше или больше первого элемента. Он удерживает первый экземпляр и передает каждый подсписок следующему в очереди человеку, который выполняет быструю сортировку переданного ему списка, возможно, привлекая других людей из очереди для сортировки подсписков. Человек, получивший пустой список, немедленно завершает работу и может вернуть этот пустой список, следуя определению первого уравнения данного алгоритма [4]. Как только человек завершит сортировку своего списка, он возвратит отсортированный список тому человеку, который передал ему этот список. Каждый человек, получающий отсортированный подсписок, составляет свой отсортированный список, располагая меньшие элементы перед элементом `x`, а б*о*льшие элементы — после него. Если останавливать рекурсию, когда списки содержат лишь один элемент, то для сортировки списка потребуется столько людей, сколько элементов содержится в списке, поскольку каждому

человеку передается лишь один элемент для сортировки. Такая стратегия, на первый взгляд, кажется напрасно расходующей ресурсы для простой сортировки списка. Но, принимая во внимание постоянно снижающиеся затраты на вычисления и повышение вычислительных мощностей, она демонстрирует истинный потенциал алгоритмов "разделяй и властвуй" и наглядно показывает, что во многих руках дело спорится.

Дополнительное исследование

Рекурсивный алгоритм для решения задачи пытается сначала решить такую же задачу для меньшего объема входных данных. Путешествие во времени из кинотрилогии *Назад в будущее* наглядно показывает, что решение задачи в прошлом является предпосылкой для его продолжения в настоящем. Аналогичную ситуацию можно обнаружить в сюжетах фильмов *Двенадцать обезьян* и *Дежавю*, в которых путешествие во времени используется для решения задачи в настоящем через решение связанной с ней задачи в прошлом. С подобной предпосылкой, но с иной точки зрения роботы в серии фильмов *Терминатор* отсылаются в настоящее из будущего с целью изменить нынешнюю реальность и сотворить иное будущее. Путешествие из будущего в настоящее составляет также основу сюжета в фильме *Петля времени.*

Решение задачи временн*ых* парадоксов в историях о путешествиях во времени обычно обходится стороной, поскольку чтение противоречивого рассказа не приносит особого удовольствия. Попытки справиться с подобными парадоксами, а значит, и с неопределенными рекурсиями, можно обнаружить в научно-фантастическом романе *11/22/63* Стивена Кинга (Stephen King), в котором учитель перемещается через временн*ой* портал в прошлое, чтобы предотвратить убийство американского президента Джона Кеннеди, что приводит к нарушению реальности в настоящем. Аналогичную историю можно обнаружить и в научно-фантастическом романе *Панорама времен* Грегори Бенфорда (Gregory Benford).

Идея осмыслить рекурсивные определения через фиксированные точки ярко показана в научно-фантастическом рассказе *Уроборос* (*Все вы, зомби*) Роберта Хайнлайна (Robert Heinlein), в котором повествуется о женщине, которая влюбляется в своего бывшего мужчиной двойника, прибывшего во времени из будущего. Родившаяся от этой любовной связи девочка похищается и возвращается назад во времени, когда взрослеет и становится женщиной, чтобы вся история началась с самого начала. Любопытно, что все эти необычные события могут быть согласованы, поскольку женщина одновременно приходится себе матерью и отцом, и данное обстоятельство приводит к фиксированной точке причинных ограничений в этой истории. На основе этой истории был снят фильм *Патруль*

времени. А в фильме *Временная петля* фиксированная точка получается созданием доппельгенгера главного героя.

Когда выполняются рекурсивные определения, они разворачиваются во вложенные структуры. Это ясно видно из примеров вышивки треугольников Серпинского, кукол-матрешек или вложенных изображений телевизора. Но нечто подобное происходит и в историях. Вероятно, одним из самых древних тому примеров служат арабские сказки *Тысяча и одна ночь.* В них речь идет о персидской царице Шахерезаде, рассказывающей истории о людях, рассказывающих свои истории, и т.д. В книге *Гедель, Эшер, Бах: эта бесконечная гирлянда* Дугласа Хофштадтера (Douglas Hofstadter) содержится диалог, в котором рассказывается вложенная история, демонстрирующая, среди прочего, стековую модель рекурсии. В этой книге можно обнаружить немало материала, основанного на рекурсии, включая несколько рисунков упоминавшегося ранее нидерландского художника М.К. Эшера.

Роман *Облачный атлас* Дэвида Митчелла (David Mitchell) состоит из нескольких вложенных одна в другую историй. Эти вложенные истории играют центральную роль в фильме *Начало*, в котором группа людей крадет мысли из умов других людей, проникая в их сны и манипулируя ими. Их задача усложняется тем, что им приходится манипулировать чужими снами рекурсивно. Это означает, что, проникнув в сон своей жертвы, они должны манипулировать ею в самом сне.

Взаимная рекурсия возникает в книге *Разделенные миры*[1] английского писателя Ричарда Каупера (Richard Cowper), в которой повествуются две истории: одна — о женатом школьном учителе на Земле, придумавшем мир супружеской пары, а другая — о женатом инопланетянине, придумавшем историю о женатом школьном учителе на Земле. Аналогичная история раскрывается в фильме *Кадр* — о парамедике, смотрящем телепередачу о воре, который смотрит телепередачу о жизни парамедика. Взаимная рекурсия возникает и в книге *Алиса в Зазеркалье* Льюиса Кэрролла, когда Алиса встречается с Единорогом и оба считают друг друга вымышленными существами.

[1] *Worlds Apart* — книга в настоящее время на русский язык не переведена. — *Примеч. пер.*

Типы и абстракция

Гарри Поттер

Пора ужинать

Завершив работу над лоскутным шитьем, вы решаете, что пора поужинать. Какие приборы вам потребуются, чтобы накрыть стол, зависит от того, что будет подано на ужин. Если вы готовите суп, вам потребуются ложки, для спагетти — еще и вилки, а для мясного блюда — не только вилки, но и ножи. Свойства отдельных блюд требуют разных свойств столовой утвари. Так, форма ложки позволяет переносить жидкость, зубцы вилки — захватывать и накручивать спагетти прядями, а лезвие ножа — резать мясо на мелкие куски. Правила пользования ложками, вилками и ножами наглядно показывают ряд важных сторон языка.

Во-первых, правила сообщают о *типах* столовой утвари, т.е. не различают отдельные ложки, вилки или ножи. Все эти столовые приборы сгруппированы в категории, а правила сообщают о произвольных членах этих категорий. И это очень важно, поскольку позволяет описывать предметы как можно меньшим числом правил. Допустим, что для описания всех разновидностей вилок в вашем буфете отсутствует слово *вилка*, а вместо него вы дали каждой вилке свое имя, такое как "Острячка", "Колючка", "Зубатка" и т.д. Чтобы выразить правило, предписывающее пользоваться вилкой для еды спагетти, вам пришлось бы упомянуть каждую разновидность вилки в отдельности. И если бы потребовалось высказаться аналогичным образом о ложках, ножах, тарелках и прочей столовой утвари, вам пришлось бы нелегко, придумывая имена каждой тарелке и запоминая их, а правила их употребления стали бы крайне сложными.

Если все это кажется вам нелепым, то потому, что так оно и есть, хотя и наглядно показывает, насколько важна роль типов в становлении эффективного языка. Слова *суп*, *спагетти* и *мясо* также служат в качестве типов в правилах, поскольку обозначают общие категории пищи, а не конкретные блюда, приготавливаемые в определенное время и в соответствующем месте. Возможно, вы обратили внимание и на то, что слово *пища* также является типом, охватывающим понятия *суп*, *спагетти* и *мясо*. Это тип более высокого уровня абстракции. Данный пример показывает, что типы и отдельные объекты можно организовывать в иерархии понятий, которые делают языки более эффективными благодаря обобщениям. Типы являются весьма действенным языковым средством для организации знаний в поддержку эффективных рассуждений.

Во-вторых, правила выражают отношения между разными типами, например между пищей и столовыми приборами. Правила на деле применяют знания о тех объектах, которые представлены типами. В частности, они упрощают рассуждения об объектах и позволяют делать выводы об их поведении. Так, правило использования вилки для спагетти указывает на то, что вилка подходит для этого блюда лучше, чем ложка. Правило кодирует прежний опыт взаимодействия с объектами, а с помощью типов оно способно представить этот опыт в краткой форме.

И в-третьих, правила предсказуемы, а это означает, что они позволяют выбрать столовые приборы и накрыть стол еще до того, как еда будет готова, в полной уверенности, что выбранные приборы подойдут для готовящихся блюд. (Это также означает, что, нарушая правила, вы можете остаться голодным.) Все это имеет значение, поскольку делает алгоритм приготовления ужина более эффективным, позволяя накрыть стол во время готовки пищи. А поскольку правило отражает предыдущий опыт приема пищи, оно избавляет вас от необходимости выяснять всякий раз, чем есть суп: ложкой или вилкой.

Есть суп вилкой или ножом было бы ошибкой. И не нужно пытаться так поступить, чтобы понять этот факт, — понимание может прийти от правил надлежащего пользования столовыми приборами. Правила сообщают о типах объектов независимо от вида употребляемой ложки, вилки или блюда, и поэтому любое действие, нарушающее такое правило, называется *ошибкой (несоответствия) типов*. Есть две разновидности ошибок типов. Во-первых, имеются ошибки, приводящие к неудачному исходу как к немедленному следствию. Например, есть суп вилкой — неудачная мысль. Такие ошибки реальны в том смысле, что делают предполагаемое действие, например, прием пищи, невозможным. В таком случае алгоритм должен быть преждевременно прекращен из-за того, что дальнейшее его выполнение становится невозможным. Во-вторых, имеются ошибки, не приводящие к неудачному исходу, но описывающие ситуации, которые, тем не менее, являются неверными или неблагоразумными. Например, пить воду можно и ложкой, но мало кто из людей это обычно делает, — наверное, потому, что такой способ неэффективен и не доставляет удовольствия от питья. Точно так же пить напитки через соломинку привычно, хотя и кощунственно для изысканного вина. Это, конечно, не ошибка, делающая соответствующее действие невозможным, но считается глупостью.

При вычислении операции выполняются над значениями. Способность правил, основанных на типах, налагать структуру на составляемое по ним вычисление, помогает осмысливать алгоритмы, предсказывать их поведение и выявлять ошибки в их выполнении. Подобно тому, как телефонные и электрические розетки имеют разную форму для защиты соответствующих приборов от повреждения, а людей — от травм, применение типизации в алгоритмах способно предотвратить вычисления, приводящие к неудачным последствиям. Волшебная сила типов описывается в главе 14, “Волшебный тип”, на примере приключений Гарри Поттера и его друзей.

14

Волшебный тип

Из всех историй, упоминаемых в этой книге, сказка о Гарри Поттере может оказаться наиболее известной. Ко многим факторам, способствовавшим популярности этой сказки, относится и то обстоятельство, что в ее центре оказывается волшебство, подчиняющееся иным законам, чем остальная физическая Вселенная. Следовательно, в подобных историях должны быть выработаны определенные правила касательно того, что возможно и что невозможно в царстве волшебства, а также указано, каким образом эти правила взаимодействуют с обычными законами природы. Эта последняя сторона очень важна для приключений Гарри Поттера, поскольку они, в отличие от многих других историй с волшебниками и магами, происходят в современной Англии, а не в каком-то отдаленном месте и времени.

Если бы волшебство в книгах о Гарри Поттере было произвольным и не подчинялось никаким законам, то все истории о нем быстро бы лишились смысла (и притягательности). Если у читателей не создаются обоснованные ожидания того, что произойдет дальше или что могло послужить причиной того или иного события, у них будет меньше стимулов продолжать чтение. Правила, отражающие законы природы, "законы" волшебства и прочие закономерности нашей жизни, очень важны для понимания событий и их причин. Они важны и для предварительного планирования. Такие законы по необходимости являются общими и касаются типов, а не отдельных предметов, поскольку сила закона кроется в его способности представлять большой ряд отдельных случаев. Типы не только характеризуют объекты, но и разделяют действия на отдельные категории.

Так, телепортация, поездка на поезде или обратный путь Гензеля и Гретель из леса домой — все это служит примером движения. Простой закон движения состоит в том, что нечто движущееся изменяет свое положение. А поскольку такой закон справедлив для любого движения и объекта, он имеет отношение к типам, а не к отдельным предметам.

В области информатики законы, описывающие закономерности вычислений, называются *правилами типизации.* Такие правила накладывают ограничения на допустимые входные и выходные данные алгоритмов, позволяя обнаруживать ошибки при выполнении алгоритмов. А поскольку типы работают на "мелкозернистом" уровне, с операциями и их аргументами на каждом шаге алгоритма, то правила типизации могут применяться для поиска *ошибок несоответствия типов* в алгоритмах. И это настолько важная задача, что она отдельно выполняется алгоритмами, называемыми *средствами контроля типов.* Так, если алгоритм не нарушает ни одно из правил типизации, то он считается *корректным в отношении типов,* а это гарантирует, что его выполнение свободно от ошибок определенного рода. Типы и правила типизации имеют большое значение для обеспечения корректного поведения алгоритмов и могут служить руководством при построении надежных алгоритмов.

Типы волшебства и волшебство типов

Волшебство, вероятно, замечательно, прежде всего, тем, что делает невозможное возможным. Превращение объектов и людей за пределами ограничений законов природы весьма привлекательно и возбуждает воображение. В книгах о *Гарри Поттере* можно найти немало тому примеров. Но применение волшебства подвержено многим ограничениям и подчиняется ряду правил. Одна из причин ограничения силы волшебства состоит в том, что оно делает истории более таинственными и увлекательными. А поскольку не все логически мыслимое оказывается фактически возможным в мире Гарри Поттера, то читатель постоянно задается вопросом, как Гарри Поттеру и его друзьям удастся преодолеть отдельные трудности в разных приключениях. Если бы они всегда могли воспользоваться каким-нибудь сверхъестественным заклинанием для решения любой конкретной задачи, то история их приключений была бы неинтересной. Поскольку волшебство не всесильно, значительная часть книг о Гарри Поттере посвящена разъяснению правил волшебства, включая его возможности и ограничения.

Для упрощения осмысления волшебного мира он разделен по категориям на ряд взаимосвязанных концепций. Например, человек, способный совершать волшебство, называется чародеем (или магом), тогда как обыкновенный человек, не способный совершать волшебство, называется магглом (т.е. простаком). Маги различаются далее по роду своих занятий на авроров (мракоборцев), аримантов

(гадателей по числам), снимающих заклятия, травников (знахарей, лечащих травами) и многих других. Волшебные действия называются заклинаниями и далее делятся на категории заговоров, заклятий, перевоплощений и прочих. Название, присваиваемое отдельной категории, является произвольным и не имеет особого значения. Важнее другое — это свойства и виды поведения, общие для всех объектов данной категории. А отношения между отдельными категориями волшебства столь же важны, как и их смысловое значение. Например, чары могут накладываться только магами, но заклинание способно воздействовать как на магов, так и на магглов. Волшебство в книгах о Гарри Поттере может быть довольно сложным. Чтобы наложить заклинание, магу, как правило, требуются волшебная палочка и магическая формула. Но опытные чародеи могут накладывать несловесные чары и без волшебных палочек. Действие заклинаний обычно ограничено по времени, а иногда может быть защищено от контрзаклятия. Волшебство может также храниться в зельях, что дает возможность совершать чудеса даже магглам, если им доступны эти зелья. Волшебство — дело непростое, что иллюстрирует тот факт, что молодым магам и магиням приходится проходить школу волшебства в течение семи лет, чтобы овладеть этим мастерством.

Классификация людей или объектов на разные категории по отдельным свойствам или способностям распространяется, конечно, не только на волшебство; она встречается буквально во всех областях жизни. Классификация повсеместно применяется в науке и является основной составляющей нашего представления об окружающем мире. Мы бессознательно пользуемся ею всякий раз, когда рассуждаем о чем-нибудь. Примеры классификации разнятся от таких обыденных задач, как выбор приборов для обеденного стола или одежды по погоде, до более абстрактных областей наподобие классификации и обдумывания философских и политических идей. Сам процесс классификации систематически изучается в информатике, поскольку способен значительно расширить наше представление о вычислениях и помочь нам в создании более надежного программного обеспечения на практике.

В информатике класс объектов, ведущих себя определенным образом, называется *типом.* В предыдущих главах книги типы встречались уже не раз. Так, в главе 4, "Записная книжка сыщика" были представлены различные *типы данных* (множество, стек и очередь) предназначенные для хранения и поддержания коллекций объектов. У каждого из этих типов данных имеется свой способ вставки, извлечения и удаления элементов из коллекции. Например, из стека объекты извлекаются в порядке, обратном их попаданию в стек (по принципу "последним пришел, первым вышел"). Из очереди же объекты извлекаются в том порядке, в каком они в нее поступают (по принципу "первым пришел, первым вышел"). Таким образом, в отдельном типе данных инкапсулируется конкретное поведение для работы с коллекциями элементов. Благодаря своему поведению типы данных оказываются пригодными для поддержки конкретных вычислительных

задач. Например, в главе 13, “Все дело в интерпретации”, тип данных стека был использован для объяснения механизма действия интерпретатора и отслеживания разных аргументов при рекурсивном вызове алгоритмов.

Типы применяются также для описания требующихся и ожидаемых входных и выходных данных алгоритмов. Например, тип алгоритма для сложения двух чисел может быть описан следующим образом:

(*Number; Number*) → *Number*

Здесь тип аргументов алгоритма указывается слева от стрелки, а тип результата — справа от нее. Данная запись гласит, что алгоритм принимает пару чисел и получает число в качестве результата. Обратите внимание, что тип алгоритмов для вычитания, умножения или любой другой операции над числами будет таким же, как и для сложения. Это показывает, что тип не связан с конкретным алгоритмом или вычислением, что согласуется с тем, что типы служат описаниями классов объектов. В данном случае тип описывает обширный ряд вычислений.

Рассмотрим в качестве еще одного примера алгоритм подъема по утрам, упоминавшийся в главе 2, “От слов к делу: когда действительно происходит вычисление”, и обладающий параметром *время подъема*, предписывающим данному алгоритму время звонка будильника. Этот параметр в качестве входных данных требует конкретное время, т.е. пару чисел, обозначающих часы и минуты подъема. Кроме того, оба числа не должны быть произвольными: часы должны находиться в пределах от 0 до 23 [1], а минуты — в пределах от 0 до 59. Числа вне этих пределов не имеют никакого смысла и способны вызвать неожиданное поведение алгоритма. Следовательно, тип алгоритма подъема по утрам можно описать так, как показано ниже.

(*Hour; Minute*) → *Alarm*

Здесь *Hour* и *Minute* — это типы для описанных выше подмножеств чисел, обозначающих часы и минуты соответственно, тогда как *Alarm* — тип, описывающий поведение будильника. В частности, будильник издает особый звук в указанное время, и подобно тому, как тип *Minute*, содержит числа в пределах от 0 до 59, тип *Alarm* содержит 24×60 = 1440 различных поведений будильника — по одному на каждую минуту суток. Если звуковой сигнал будильника можно настраивать, то в данном алгоритме требуется еще один параметр, который может быть также выражен типом. Следовательно, и тип результата *Alarm* должен быть более общим, включающим в себя разные виды звуковых сигналов.

Как видите, тип алгоритма сообщает нам нечто о самом алгоритме. Он не поясняет, что именно делает алгоритм, но сужает его функциональность. И этого зачастую оказывается достаточно для выбора алгоритма. Очевидно, что никто не стал бы применять алгоритм сложения чисел для решения задачи срабатывания будильника в заданное время. Даже не анализируя подробно алгоритмы

сложения чисел и подъема по утрам, можно сразу сказать, что это разные алгоритмы. Тип алгоритма содержит стрелку, отделяющую тип его аргументов (слева) от типа его результата (справа). Эта стрелка означает, что соответствующее вычисление преобразует входные данные одного типа в результаты другого типа. Тип результата может быть значением, как при сложении чисел, или некоторым действием, как в алгоритме подъема по утрам.

Поскольку заклинания также являются преобразованиями, для описания их действия можно воспользоваться соответствующими типами. Например, в книгах о Гарри Поттере объекты можно заставить парить в воздухе с помощью заклинания "Вингардиум Левиоза" (Wingardium Leviosa), в ходе которого маг должен указать волшебной палочкой на объект и произнести магическую формулу "Вингардиум Левиоза". Если допустить, что типы `Object` и `Levitating` обозначают объект и его парение в воздухе, а типы *`Wand`* и `Incantation` — волшебную палочку и магическую формулу соответственно, то тип заклинания может быть описан следующим образом:

```
(Wand; Incantation; Object) → Levitating
```

Типы служат удобной записью для обсуждения свойств алгоритмов и вычислений. Стрелка, отделяющая аргументы от типов результатов, является одной из самых важных составных частей такого обозначения. Другой его важной частью являются *параметры типа*. Рассмотрим в качестве примера алгоритмы сортировки, упоминавшиеся в главе 6, "Сортировка алгоритмов сортировки". В качестве входных данных алгоритм сортировки принимает список элементов и производит отсортированный список этих же элементов в качестве результата. Кроме того, элементы списка должны каким-то образом сравниваться, чтобы выяснить, равны ли они или же один из них больше другого. Поскольку числа можно сравнивать между собой, следующий тип является корректным типом для любого алгоритма сортировки списка чисел:

```
List(Number) → List(Number)
```

Обозначение типа списков чисел как `List(Number)` указывает на возможность получать разнотипные списки, заменяя аргумент `Number`. Например, сортировать можно и текстовую информацию, поэтому соответствующий алгоритм сортировки может иметь тип `List(Text)→List(Text)`. Чтобы выразить возможность алгоритмов иметь разные, хотя и связанные типы, тип любого конкретного элемента (например, `Number` или `Text`) можно заменить параметром типа, вместо которого можно затем подставить конкретный тип. Таким образом, тип алгоритмов сортировки можно выразить по приведенному ниже шаблону, где `comparable` означает любой тип, элементы которого можно сравнивать [2].

```
List(comparable) → List(comparable)
```

Любой конкретный тип, такой как *List(Number)*→*List(Number)*, может быть получен из этого шаблона подстановкой типа элемента вместо параметра типа *comparable*.

Присоединяя тип к наименованию алгоритма с помощью двоеточия, можно указать, что алгоритм имеет именно этот конкретный тип. Например, чтобы указать, что алгоритм сортировки *Qsort* имеет этот тип, можно написать следующее выражение:

```
Qsort : List(comparable) → List(comparable)
```

Алгоритм *Count*, упоминавшийся в главе 12, “Своевременный стежок вычисляется впрок”, также принимает список элементов в качестве входных данных, но в нем не требуется, чтобы элементы можно было сравнивать, а в качестве результата он возвращает число. Таким образом, тип алгоритма *Count* может быть описан так, как показано ниже, где *any* — параметр типа, обозначающий произвольный тип.

```
Count : List(any) → Number
```

Зная тип объекта, можно выяснить, что он может делать и что можно делать с ним. Зная типы алгоритмов, можно выбирать и корректно их применять. Информация о типе может оказать в этом помощь самыми разными способами.

Во-первых, если требуется выполнить конкретный алгоритм, его входные данные подскажут, какого типа аргументы следует для этого предоставить. Например, для наложения заклятия требуются магическая формула и волшебная палочка, тогда как чудодейственное зелье должно быть выпито. Пить заклятья не имеет никакого смысла так же, как и произносить магическую формулу и совершать движение волшебной палочкой над зельем для его применения (не изготовления). Аналогично алгоритм сортировки вроде *Qsort* можно применить к списку, но вряд ли стоит применять его к моменту времени. Без аргументов правильного типа выполнить алгоритм нельзя.

Во-вторых, тип объекта говорит нам о том, какими алгоритмами мы можем воспользоваться для его преобразования. Так, если у нас есть список, мы знаем, что можем подсчитать его элементы. А если элементы в списке можно сравнивать, то мы знаем также, что можем его отсортировать. Маг может заставить объект парить в воздухе с помощью заклинания “Вингардиум Левиоза”. Объекты конкретного типа могут быть преобразованы с помощью любого алгоритма, принимающего аргументы данного типа.

В-третьих, если нам требуется вычислить нечто конкретного типа, то типы результатов алгоритмов сообщают нам, какие именно вычисления в принципе могут быть выполнены для решения данной задачи. Принимая во внимание также типы имеющихся аргументов, можно еще больше сузить круг подходящих алгоритмов. Например, чтобы заставить объект парить в воздухе, вероятно, следует

выбрать заклинание "Вингардиум Левиоза". Объекты требующегося типа могут быть построены только с помощью алгоритма, имеющего тот же тип результата.

Повсеместное применение типов объясняется их способностью эффективно организовывать знания. Типы дают нам возможность абстрагироваться от отдельных примеров и обдумывать сложившееся положение на общем уровне. Например, зная, что только ведьма или маг может летать на метле, мы можем сделать вывод, что Гарри Поттер мог бы летать на метле, в то время как его тетка Петуния не смогла бы этого сделать, поскольку относится к магглам. А когда Гарри Поттер пользуется своим плащом-невидимкой, он полагается на его свойство делать невидимым все, что он покрывает. Многочисленные примеры применения типов можно найти и в повседневной, не волшебной, жизни, когда мы выбираем какой-нибудь конкретный предмет (отвертку, зонтик, часы, блендер и т.д.), потому что его общие свойства связаны с нашими насущными потребностями, или когда мы прогнозируем результаты конкретных взаимодействий, размышляя о сталкивающихся объектах, подключении электроприбора к сети или поведении людей в разных ролях — клиента, пациента, родителя и т.д.

Правило для правил

Помимо поддержки эффективного получения сведений о конкретных объектах, типы позволяют рассуждать о взаимодействии объектов, а следовательно, делать прогнозы относительно окружающего мира. Этого можно достичь с помощью ряда правил, называемых *правилами типизации* и описывающих возможность и порядок взаимодействия разнотипных объектов. Так, если какой-нибудь хрупкий предмет упадет на твердый пол, он, скорее всего, разобьется. В таком случае прогноз можно вывести из правила, включающего несколько типов: типы хрупких и разбивающихся предметов, тип ускоренных движений, а также тип твердых поверхностей. Такое правило гласит, что объект, относящийся к типу хрупких предметов, скорее всего, будет относиться к типу разбитых предметов после того, как испытает ускоренное движение и столкновение с твердой поверхностью. Это общее правило может быть использовано для прогнозирования в самых разных ситуациях, включая различные виды хрупких предметов (яйца, бокалы, мороженое в стаканчике), твердые поверхности (мощеные дороги, кирпичные стены, паркетный пол), а также ускоренные движения (падение, бросание). Такое правило сводит большой ряд фактов в одно краткое описание. И подобное приведение достигается путем применения типов вместо отдельных названий для представления подходящих свойств.

Правило типизации описывает тип объекта при определенных условиях. Условия, которые должны быть выполнены для применения правила, называются *предпосылками*, а высказывание о типе объекта, который правило выводит

из предпосылок, называется *заключением.* Так, у правила о разбивании хрупких предметов имеются три следующие предпосылки: объект хрупкий, он падает, а поверхность, на которую он опускается, оказывается твердой. Из этого правила следует заключение, что объект, как правило, разбивается. Правило типизации, имеющее лишь одно заключение, но не имеющее предпосылок, называется *аксиомой*, что указывает на безусловную истинность правила. Примерами аксиом служат следующие утверждения: "Гарри Поттер — маг", "3 — это число" и "Тонкое стекло — хрупкое".

Выявление типов и их присваивание объектам — это вопрос разработки, которая зависит от конкретной ситуации или преследуемых целей. Например, в контексте истории о Гарри Поттере очень важно отличать магглов от магов, но вряд ли это уместно в контекстах, в которых волшебство отсутствует. Большинство правил, касающихся объектов из повседневной жизни, первоначально были выведены для того, чтобы нам было легче выжить в естественном мире. Они помогают нам избегать фатальных ошибок. По мере развития техники и культуры возникает потребность в новых правилах, чтобы эффективно ориентироваться во всех областях современной жизни, как, например, пользоваться лифтом или сотовым телефоном, страховать свое здоровье. Эти примеры демонстрируют, что разработка типов и правил типизации является непрерывным процессом, обусловленным конкретными целями.

Самым важным и фундаментальным правилом типизации для вычислений является *правило применения.* Оно связывает тип алгоритма с типом аргумента, к которому он применяется, а также с типом производимого им результата. Данное правило требует, чтобы алгоритм можно было применять только к тем аргументам, тип которых оказывается таким же, как и тип входных данных этого алгоритма. В таком случае тип результата, выводимого алгоритмом, оказывается таким же, как и выходной тип этого алгоритма. Правила типизации нередко представляются в виде отношения с предпосылками в числителе и заключением в знаменателе. В таком представлении правило применения алгоритма к его аргументам будет выглядеть следующим образом:

$$\frac{\mathtt{Alg}\colon \mathtt{Input} \rightarrow \mathtt{Output} \qquad \mathtt{Arg}\colon \mathtt{Input}}{\mathtt{Alg(Arg)}\colon \mathtt{Output}}$$

Первая предпосылка данного правила требует, чтобы у алгоритма (`Alg`) был тип входных (`Input`) и выходных (`Output`) данных. Это требование тесно связано с законом элементарных превращений Гэмпа из книги *Гарри Поттер и дары смерти*, гласящим, что пища не может быть создана из разреженного воздуха. Ее можно только вызывать из разных мест и увеличивать в размере. Правило применения отражает фундаментальное свойство алгоритмов: для получения переменных выходных данных они зависят от входных данных.

Заключение может быть сделано из правила лишь в том случае, если удовлетворяются все его предпосылки. Например, яйцо, упавшее на проезжую дорогу, удовлетворяет правилу о разбитии хрупких предметов. Из этого можно сделать заключение, что яйцо разобьется. Если 6 — правильное обозначение часов, 30 — правильное обозначение минут, а (*Hour*; *Minute*) — обозначение типа аргумента в алгоритме подъема по утрам, то данный алгоритм можно применить к двум аргументам и заключить, что тип результата *Alarm* поведет себя правильно. Правило типизации применяется путем подстановки конкретного алгоритма вместо типа *Alg* и аргументов вместо типа *Arg*, а также входных данных вместо типа *Input* и выходных данных вместо типа *Output* в приведенном выше представлении. Так, применение правила типизации к алгоритму подъема по утрам будет выглядеть следующим образом:

$$\frac{\mathit{WakeUp}\colon (\mathit{Hour},\mathit{Minute}) \to \mathit{Alarm} \qquad (6,30)\colon (\mathit{Hour},\mathit{Minute})}{\mathit{WakeUp}(6,30)\colon \mathit{Alarm}}$$

Аналогично, если *L* — это список чисел, то к алгоритму быстрой сортировки *Qsort* можно применить правило типизации *List*(*Number*)→*List*(*Number*) и заключить, что в результате применения алгоритма *Qsort* к списку *L* получится отсортированный список чисел, как показано ниже.

$$\frac{\mathit{Qsort}\colon \mathit{List}(\mathit{Number}) \to \mathit{List}(\mathit{Number}) \qquad L\colon \mathit{List}(\mathit{Number})}{\mathit{Qsort}(L)\colon \mathit{List}(\mathit{Number})}$$

Возможность прогнозировать исход ситуаций исходя лишь из типов причастных к ситуации объектов является эффективным механизмом рассуждений, позволяющим здравомыслящим людям пользоваться ложкой, когда они едят суп, и не одеваться перед принятием душа. Типы и правила типизации дают нам компактное представление сходных объектов и событий, делая данный процесс эффективным. Это справедливо не только для рассуждений о повседневных ситуациях, но для таких предметных областей, как волшебный мир Гарри Поттера или мир вычислений.

В мире Гарри Поттера сила волшебства перекрывает множество законов мира природы. Поэтому маги иногда по-разному рассуждают о повседневных объектах и действуют соответственно. Например, маги могут избежать многих неприятных работ, которые вынуждены делать магглы, как в том случае, когда мать Рона Уизли пользуется волшебством как вспомогательным средством при приготовлении пищи и уборке. Магам просто не имеет смысла делать все это вручную. Еще одним примером служит обычная езда в автомобиле, которой маги гнушаются, поскольку это неэкономично по сравнению с такими волшебными формами перемещения, как телепортация, дымолетный порошок, порткючи и, конечно, полет.

Типы алгоритмов предсказывают результаты вычислений и могут использоваться в рассуждениях о них. Допустим, что вам требуется отсортировать список, но вы не знаете, как это сделать. Вам вручили три запечатанных конверта, А, Б и В, с разными алгоритмами, и вы не знаете их содержимое, но известно, что в одном из конвертов находится нужный вам алгоритм сортировки. В качестве подсказки на каждом конверте написан тип алгоритма, как показано ниже.

```
Конверт А: (Hour; Minute)    → Alarm
Конверт Б: List(any)         → Number
Конверт В: List(comparable)  → List(comparable)
```

Какой из конвертов вы выберете?

Когда правила неприменимы

Типы и правила типизации служат каркасом для рассуждений, повышая их эффективность. Но нередки случаи, когда правила неприменимы. Всякий раз, когда не удовлетворена одна или несколько предпосылок, правило неприменимо, а это означает, что заключение из него сделано неверно. Например, молоток, падающий на проезжую дорогу, не удовлетворяет предпосылке о хрупких предметах, а когда яйцо падает в ящик с песком, то не удовлетворяется предпосылка о твердой поверхности. Следовательно, в этих случаях заключение о разбивании объекта сделано быть не может. Аналогично, когда Рон Уизли пытает поднять перышко в воздух с помощью магической формулы "Вингардиум Левиоза", перышко не сдвигается с места, поскольку он неверно произносит магическую формулу.

> *"Вингардиум Левиоза!" — выкрикнул он [Рон], махая длинными руками, как мельница. "Ты произносишь неверно, — услышал Гарри слова Гермионы. — Произнеси «Вин-гар-диум Леви-о-за», удлинив и смягчив слог «гар»".*

В этом случае перышко не поднимается в воздух, поскольку не удовлетворена одна из предпосылок произнесения магической формулы. Следовательно, правило не применяется и из него не следует заключение о парении в воздухе.

Если встречается ситуация, когда правило типизации неприменимо, это совсем не означает, что данное правило неверно. Это означает лишь, что область его действия ограничена, а значит, им нельзя воспользоваться, чтобы спрогнозировать поведение в данной ситуации. Более того, если правило неприменимо, из этого совсем не следует, что оно, определенно, ложно. Оно может быть истинным в других условиях, не охваченных данным правилом. Например, правило "Если идет дождь, то проезжая дорога мокрая" неприменимо в солнечный день. Тем не менее проезжая дорога может оказаться мокрой, если по ней проехалась поливочная машина. А если вернуться к предыдущему примеру из истории о Гарри

Поттере, то перышко может все же воспарить, даже если Рон неверно произнесет магическую формулу "Вингардиум Левиоза", поскольку кто-нибудь другой может в тот же момент произнести ее правильно. Молоток все же может разбиться, упав на пол, если его рукоятка уже была частично разбита. Следовательно, когда правило неприменимо, из него нельзя сделать никакого заключения.

Таким образом, неприменимость правила типизации может казаться не столь уж важной. Но на самом деле это зависит от того, насколько важно, чтобы конкретное заключение оказалось истинным. Если воспарение перышка или разбивание упавшего предмета — скорее забава, то имеется немало ситуаций, когда заключение имеет важное значение. Как тут не вспомнить шутки о "знаменитых последних словах", в которых обыгрываются неверные заключения, например "Красный провод можно смело резать", "Какой милый песик", "Эти грибы явно съедобны" и т.д. Не менее важно достичь правильности типов в вычислении. И хотя работа со значениями неверного типа редко приводит к фатальным последствиям, она, как правило, приводит к неверному результату вычислений. И вот почему. На каждом шаге алгоритма выполняется преобразование одного или нескольких значений, и каждая операция, применяемая в данном процессе, основывается на том, что аргументы данного алгоритма относятся к конкретному типу, поскольку только для него алгоритм в состоянии сделать нечто значимое. Напомним, что время подъема нельзя сортировать, как и вычислять квадратный корень из списка. Если на любом шаге алгоритма встречается значение неожиданного типа, то алгоритм не будет знать, что с ним делать. Как следствие вычисление не сможет быть успешно завершено и предоставить значимый результат. Короче говоря, успех вычисления зависит от предоставления операциям значений только правильных типов.

Таким образом, правила типизации являются весьма важной составляющей вычислений, поскольку способны гарантировать, что операции не дадут сбой при неверных значениях и что вычисление в целом может быть успешно завершено [3]. Правила типизации могут быть использованы не только для выявления бессмысленных предложений наподобие "Гарри Поттер выпил заклинание", но и нелепых примеров применения алгоритмов наподобие `Qsort(6:30am)`. А поскольку алгоритм состоит из множества шагов, включающих применение операций и других алгоритмов к значениям, с помощью правил типизации можно выявлять ошибки в определении алгоритма и активно поддерживать построение правильно типизированных алгоритмов. Такой процесс иногда еще называется *типонаправленным программированием* (type-directed programming).

Правила типизации предотвращают не только невозможные применения операций к значениям, но и те операции, которые не имеют смысла в определенном контексте. Например, ваш рост и возраст можно выразить числами, но не имеет никакого смысла складывать эти два числа. Такое сложение не имеет смысла

потому, что эти два числа представляют разные понятия (см. главу 3, "Тайна знаков"), а следовательно, их сумма не означает ничего осмысленного [4]. Подобные примеры можно найти и в книгах о Гарри Поттере. Например, в игре в квиддич две команды пытаются забросить мяч (квоффл) в кольцо на одной из стоек ворот противника. Это, конечно, можно было бы сделать с помощью заклинания для телепортации, но тогда игра потеряла бы всякий интерес и цель, которая состоит в том, чтобы выявить лучшую команду игроков в квиддич. Следовательно, применять волшебство (кроме летания на метле) игрокам в квиддич запрещается. Подобные ограничения весьма распространены в играх. Так, требование ходить в масть в карточной игре или передвигать фигуру слона только по диагонали совсем не отражает невозможность карточной игры с другими правилами или физическую невозможность передвинуть фигуру слона по другой прямой линии. Это лишь ограничение, наложенное в конкретном контексте, чтобы игра отвечала своим целям. Типы могут использоваться для отслеживания подобных представлений по ходу вычислений, чтобы в операциях принимались во внимание правила их сочетания для представленных значений.

Нарушение правил типизации в алгоритме называется *ошибкой несоответствия типов*. Такая ошибка указывает на несогласованность сочетания операций и значений, над которыми они выполняются. Алгоритм, в котором нет ошибок несоответствия типов, считается *правильно типизированным*. Ошибка несоответствия типов может иметь разные последствия. Во-первых, она может вызвать сбой при вычислении, что, в свою очередь, может привести к преждевременному прекращению вычисления и к сообщению об ошибке. К такому последствию, как правило, приводит попытка деления числа на нуль. В книге *Гарри Поттер и тайная комната* волшебная палочка Рона ломается, и вследствие этого все заклинания перестают действовать. Например, заклинание "Съешь слизняки", которое он накладывает на Малфоя, оборачивается против него самого, а его попытка превратить крысу в кубок приводит к появлению пушистой чаши с хвостом. И хотя конкретный результат мог оказаться неожиданным, сами неудачи Рона в сотворении волшебства были вполне прогнозируемыми, поскольку его сломанная волшебная палочка нарушила одну из предпосылок правил типизации для магии. В подобных случаях последствия неработающей магии становятся сразу же заметными невооруженным глазом. Во-вторых, ошибка несоответствия типов может и не привести к какому-нибудь сразу же заметному последствию, а это означает, что вычисление продолжится и в конечном итоге завершится. Но при этом оно будет работать с бессмысленными значениями и в конечном счете произведет столь же бессмысленный результат. Характерным примером служит сложение возраста и роста человека.

И если нет ничего хорошего во внезапном прерывании вычисления без получения окончательного результата, то вычисление, которое продолжается, но приводит к бессмысленному результату, — еще хуже. Дело в том, что в таком

случае можно не распознать неверный результат и использовать его для принятия важного решения. Примером тому служит ситуация, когда Гарри Поттер неверно произносит "Diagonally" (По косой) вместо "Diagon Ally" (Косой переулок) в попытках путешествия с помощью дымолетного порошка и попадает совершенно не туда, куда нужно. Само путешествие не прервалось, дымолетный порошок действовал безукоризненно... но результат получился неверный. Гарри Поттеру было бы лучше прервать свое путешествие, поскольку его едва не похитили и ему пришлось вступить в схватку с какими-то темными предметами в магазине, где он в конечном счете оказался.

Обеспечение соблюдения законов

Правильность типов является очень важной предпосылкой для правильного функционирования любого алгоритма, поэтому было бы неплохо проверять правильность типов автоматически, используя отдельный алгоритм, который называется *средством контроля типов* (type checker). Это средство определяет, соответствуют ли шаги алгоритма правилам типизации для управляющих структур, переменных и значений. Средство контроля типов способно обнаруживать ошибки несоответствия типов в алгоритме и тем самым предотвращать ошибочные вычисления.

Типы в алгоритме можно контролировать двумя разными способами. Одна из возможностей состоит в том, чтобы контролировать типы во время выполнения алгоритма. Такой способ называется *динамическим контролем типов*, поскольку происходит во время выполнения алгоритма. Прежде чем выполнять операцию, определяется соответствие типов аргументов требованиям типизации применяемой операции. Недостаток такого способа заключается в том, что он не приносит никакой пользы, если обнаруживается ошибка несоответствия типов. Как правило, единственная возможность в таком случае — прервать вычисление, что может быть весьма досадно, особенно тогда, когда бо́льшая часть предполагаемого вычисления уже проведена и алгоритм прерывается непосредственно перед получением конечного результата. Поэтому намного лучше заранее знать, сможет ли вычисление быть успешно завершено, не доводя дело до ошибки несоответствия типов и не тратя зря ресурсы на вычисление, которое обречено на неудачу.

Другой способ состоит в том, чтобы проверить все шаги алгоритма на соблюдение правил типизации, не выполняя его. Алгоритм при этом выполняется лишь в том случае, если ошибки несоответствия типов не обнаружены. Такой способ называется *статическим контролем типов*, поскольку происходит без перехода алгоритма в динамический режим выполнения. Статический контроль типов подсказал бы Рону не пытаться творить волшебство своей сломанной волшебной палочкой.

Еще одно преимущество статического контроля типов заключается в том, что алгоритм приходится проверять лишь один раз. После этого он может выполняться неоднократно для разных аргументов без дополнительной проверки типов. И если при динамическом контроле типов приходится многократно проверять операции, выполняемые в цикле (по существу, в каждой итерации цикла), то при статическом контроле типов любую операцию приходится проверять лишь один раз, благодаря чему алгоритм выполняется быстрее. Динамический контроль типов приходится проводить всякий раз, когда алгоритм выполняется.

Тем не менее преимущество динамического контроля типов заключается в том, что он, как правило, оказывается более точным, поскольку в нем принимаются во внимание конкретные значения, обрабатываемые в алгоритме. Если при статическом контроле типов известны только типы параметров, то при динамическом контроле типов — конкретные их значения. Статический контроль типов вообще не дал бы Рону пользоваться своей сломанной волшебной палочкой и тем самым предотвратил бы любые ошибки, а динамический контроль типов позволил бы ему опробовать заклинания и преуспеть или потерпеть неудачу в зависимости от конкретной ситуации. Например, когда Рон пытается превратить крысу в чашу, он достигает успеха лишь наполовину, поскольку чаша все равно получается с хвостом.

Определение точного поведения типов в алгоритме, так же как и выяснение его завершимости, является неразрешимой задачей (см. главу 11, "Счастливый конец не гарантируется"). Рассмотрим в качестве примера алгоритм, в котором ошибка несоответствия типов имеется только в ветви `then` условной конструкции. Эта ошибка возникает в алгоритме лишь в том случае, если вычисление условия, заданного в условной конструкции, дает *истинный* результат и затем выбирается ветвь `then`. Но дело в том, что для проверки значения заданного условия может потребоваться произвольно сложное вычисление (например, в качестве условия может быть задана переменная, значение которой вычисляется в каком-нибудь цикле). А поскольку нельзя выяснить, завершится ли когда-либо такое вычисление (ведь задача останова не имеет решения), нельзя и заранее знать, проявится ли в алгоритме или программе ошибка несоответствия типов.

Чтобы справиться с отсутствием возможности предсказать поведение алгоритма, при статическом контроле типов производится аппроксимация типов в алгоритме. А для того чтобы избежать вычислений, которые могли бы привести к невозможности завершения алгоритма, при статическом контроле типов предпринимаются особые меры предосторожности и сообщается об ошибке несоответствия типов всякий раз, когда такое может произойти. В случае условной конструкции это означает, что средство статического контроля типов, будучи не в состоянии вычислить значение условия, требует, чтобы ошибка несоответствия типов отсутствовала в обеих альтернативных ветвях данной конструкции.

Следовательно, сообщение об ошибке несоответствия типов будет выведено даже в том случае, если она содержится только в одной ветви условной конструкции. И хотя при выполнении алгоритма такая ошибка несоответствия типов может и не проявить себя, средство статического контроля типов все равно признает проверяемый алгоритм неверно типизированным и запретит его выполнение. Это цена, которую приходится платить за надежность статического контроля типов, когда некоторые алгоритмы будут отвергаться, несмотря на то что при их выполнении ошибка может фактически и не возникнуть.

Статический контроль типов обеспечивает безопасность за счет точности. Хотя ошибка может и не возникнуть, это все же не исключено, а поскольку рисковать неудачным исходом вычисления нельзя, лучше сообщить о потенциальной ошибке в алгоритме и исправить ее, прежде чем выполнять такой алгоритм. Столь осмотрительный подход, выбранный для статического контроля типов, характерен не только для информатики. Имеется немало предметных областей, в которых проверка выполняется заранее. Если вы летите самолетом, то полагаетесь на то, что пилоты убедятся в наличии достаточного запаса топлива и проверят функционирование всех бортовых систем до взлета. Тем самым вы ожидаете, что будет произведен статический контроль типов по правилам, обеспечивающим безопасный полет. А теперь на минутку представьте альтернативный подход к проверке бортовых систем — уже после взлета, когда обнаруженную неполадку, скорее всего, будет невозможно устранить.

Если доктор назначает вам медицинскую процедуру или прописывает вам лекарства, он непременно должен заранее выявить любое противопоказание, чтобы не нанести никакого вреда вашему здоровью. Статический контроль типов следует принципу "береженого Бог бережет". Следуя этому принципу статического контроля типов, Рон не должен был пользоваться заклинаниями — из-за сломанной волшебной палочки правило сотворения волшебства оказалось неприменимым и предвещало ошибку во время выполнения алгоритма заклинания. При динамическом контроле типов применяется менее строгий подход к соблюдению правил типизации, который не оставляет никакой возможности не выполнять вычисление и тем самым подвергает его риску столкнуться с ошибками во время выполнения алгоритма. Именно так Рон и поступил, положившись на удачу, но ничего хорошего из этого не вышло.

Построение кода

Нарушение правил типизации служит признаком чего-то неверного. Ошибка несоответствия типов в алгоритме указывает на то, что он, вероятнее всего, будет действовать неправильно. В лучшем случае, когда все составные части алгоритма удовлетворяют правилам типизации, некоторые виды ошибок

могут и не возникнуть, и алгоритм в определенной степени может считаться правильным. Безусловно, такой алгоритм все равно может давать неверные результаты. Так, если нам нужен алгоритм для сложения двух чисел, имеющий тип `(Number;Number)→Number`, но случайно создан алгоритм для вычитания этих чисел, то с точки зрения типов он будет считаться правильным, хотя будет вычислять неверные значения. Тем не менее в корректно типизированной программе исключено весьма большое количество ошибок и по крайней мере можно быть уверенным, что шаги алгоритма соответствуют важному уровню согласованности.

Защита от неверного или бессмысленного вычисления является важной функцией типов и правил типизации, но это не единственное преимущество, которое они дают. Типы действуют также в качестве пояснений шагов алгоритма. Не разбираясь в подробностях выполнения каждого шага алгоритма, можно составить общее представление о том, что вычисляется, глядя только на типы. Возможность различать вычисления по их типам была продемонстрирована ранее на примере выбора конверта с нужным алгоритмом. Способность типов подытоживать большое число вычислений дает представление, которое нельзя составить, рассмотрев только ряд конкретных примеров. Таким образом, типы и правила типизации имеют также пояснительную ценность.

В качестве примера снова рассмотрим игру в квиддич. Эту игру можно объяснить, даже не показывая ее, но описав общие правила и роли разных игроков. Именно так Гарри Поттер и узнает о ней от Оливера Вуда в книге *Гарри Поттер и философский камень*. Правила игры ограничивают допустимые действия игроков и определяют, как ведется счет в игре. При этом важно отметить, что правила игры тождественны правилам типизации, в которых типы используются для описания ролей игроков (например, роли "ловца") и видов объектов (например, мяча "квоффла"). И хотя было бы полезно увидеть игру воочию, этого было бы недостаточно, чтобы понять ее правила. Даже просмотрев много игр, не зная их правил, можно легко быть захваченным врасплох. Например, игра в квиддич завершается, как только ловец захватывает Золотой Снитч (маленький шарик с крылышками, произвольно летающий над игровым полем). Если этого не случалось в просмотренных ранее играх, зритель будет удивлен, когда это произойдет на его глазах впервые. У других игр имеются свои особые правила, которые также могут удивить. Например, офсайд в футболе или правило взятия на проходе в шахматах. Понять игру, только наблюдая ее, трудно. Чтобы оказались задействованными все правила, может потребоваться немало времени и игр.

Точно так же нелегко разобраться в алгоритме, только наблюдая за тем, как его входные данные преобразуются в соответствующие выходные данные. И хотя типов, как правило, оказывается недостаточно для точного описания действия алгоритма, они все же поясняют на общем уровне его поведение и поддерживают

представление о том, как его составные части (отдельные шаги и управляющие структуры) согласуются и действуют вместе.

Типы структурируют объекты предметной области, а правила типизации поясняют, как их можно содержательно сочетать. В информатике типы и правила типизации обеспечивают взаимодействие разных частей алгоритма. Поэтому они являются важными инструментами для построения крупных систем из более мелких. Именно об этом речь и пойдет в главе 15, "С высоты птичьего полета: от абстракции к деталям".

В конце дня

Когда долгий день вычислений подходит к концу, вы начинаете размышлять над всем происшедшим и делаете заметки о некоторых событиях в своем дневнике. Подъем этим утром ничем особенным не отличался от подъема в другие дни. Сегодня не произошло ничего достойного записи в дневнике. Чистка зубов дольше обычного или закончившийся тюбик зубной пасты — не те события, которые стоит запоминать на целые годы. То же самое относится и к большинству событий, происходящих в течение дня. Каждый будний день вы завтракаете и едете на работу, и поэтому начинать запись в дневнике с фразы вроде "Это был еще один заурядный будний день" — все равно что выполнить стандартную процедуру подъема и завтрака по утрам. В частности, наименование *будний день* заменяет подробное описание подъема, завтрака, поездки на работу и всех остальных процедур, выполняемых в течение обыкновенного рабочего дня.

Присваивание (краткого) названия (длинному) описанию и последующее употребление этого названия для обозначения этого описания служит формой абстракции. Подобно тому, как разные города обозначаются на карте одинакового вида точкой, в названии *будний день* игнорируются различия в днях недели и все дни считаются одинаковыми. Однако, поскольку точки находятся в разных местах на карте, они различны — как различны *будние дни*, по-разному упоминаемые в разное время. Разное положение в пространстве и времени придает ссылкам особый контекст и дополнительное смысловое значение. Например, конкретный город находится у моря или соединен с остальным миром отдельным шоссе, а некоторый будний день приходится на выборы или праздник.

Обычное наименование или символ, несмотря на всю свою эффективность во множестве случаев, может оказаться слишком абстрактным представлением. Чтобы снабдить разные ссылки дополнительной информацией, наименования и символы могут быть дополнены параметрами. Например, точки для обозначения городов на карте иногда параметризируются размером или цветом, чтобы отличать крупные города от мелких, а шрифт названия может варьироваться, чтобы различать столицы стран, и т.д. Аналогично может быть параметризировано наименование, обозначающее день недели. Кстати, наименование *будний день* уже отличает рабочий день от выходного дня, когда не нужно идти на работу.

Истинный потенциал параметров полностью раскрывается, когда они упоминаются в кратких описаниях. Допустим, что вам требуется сделать в своем дневнике запись о важной встрече или приеме у врача. Подобно наименованию *будний день*, термины *встреча* и *прием у врача* являются абстракциями, обозначающими целый ряд событий, которые обычно происходят во время подобных мероприятий. Помимо самой важной встречи, вы, вероятнее всего, укажете в своей записи лицо, с которым вы должны встретиться, а также цель встречи. Что касается приема у врача, то вы, скорее всего, упомянете специалиста, у которого хотели бы проконсультироваться, а также причину вашего обращения к нему. Подобные сведения могут быть представлены параметрами абстракции и использоваться в ее описании. Подобно тому, как в булочной предлагаются торты ко дню рождения, украшенные именем и возрастом именинника или именинницы, абстракция *встречи* дополняется параметрами *кто* и *цель*, на которые затем можно ссылаться в описании самой абстракции. Так, если используется абстракция "встреча с Джилл по поводу работы", то в ней параметры *кто* и *цель* заменяются конкретными значениями "Джилл" и "работа", а описание этой абстракции читается так, если бы оно было написано для данной конкретной встречи.

Абстракции определяют шаблоны и делают их повторно используемыми. С помощью параметров абстракции можно гибко приспосабливать к конкретным ситуациям и при этом передавать большое количество информации в краткой форме. Присваивание наименования отдельной абстракции и определение ее параметров задает ее интерфейс, определяющий надлежащее применение этой абстракции. Львиную долю ежедневных действий специалистов в области информатики составляют выявление, создание и применение абстракций для описания вычислений и рассуждений о них. Информатика изучает природу абстракций, формализует их определения и применение для помощи программистам и разработчикам в создании более совершенного программного обеспечения.

В завершающей главе книги поясняется, что такое абстракции и какую роль они играют в вычислениях. Поскольку абстракции играют важную роль как в естественных языках, так и в информатике, то не удивительно, что различные истории могут многое прояснить в вычислениях.

15

С высоты птичьего полета: от абстракции к деталям

Все примеры вычислений, приведенные до сих пор в этой книге, были достаточно малы, что совсем не плохо для иллюстрации рассматриваемых понятий и принципов. Но нельзя ли расширить их до б*о*льших масштабов? Вопрос масштабируемости возникает в самых разных местах.

Во-первых, встает вопрос о способности алгоритмов к обработке крупных массивов входных данных. Этот вопрос решается путем анализа временн*о*й и пространственной сложности алгоритмов, обсуждавшихся в первой части книги, в частности в главах 2–7. Если одни алгоритмы вполне допускают масштабирование (такие, как следование по пути, приготовление кофе, поиск и сортировка), то другие алгоритмы не масштабируются (например, оптимизация выбора блюд при ограниченном бюджете). Как пояснялось в главе 7, "Трудноразрешимые задачи", время выполнения экспоненциальных алгоритмов, по существу, препятствует решению некоторых задач.

Во-вторых, возникает вопрос, как создавать, понимать и сопровождать крупные программные системы. Разработать и написать небольшую программу относительно легко, но создание крупных программных систем представляет немалую головную боль для разработчиков программного обеспечения.

Чтобы стало понятнее, в чем же здесь трудность, представьте карту города или страны, в которой вы живете. Какой масштаб карты вам больше подходит? Как поясняется в последнем романе Льюиса Кэрролла *Сильвия и Бруно*, масштаб 1:1 неудобен, поскольку развернутая карта "покроет всю страну и заслонит солнечный свет". Следовательно, любая удобная карта должна быть намного меньше

того, что она представляет, а значит, в ней должны быть опущены многие подробности. При создании карты возникают важные вопросы: какой же масштаб будет достаточно мелким, чтобы с картой можно было работать, но при этом все же достаточно крупным, чтобы представить существенные подробности? какими подробностями можно пренебречь, а какие необходимо оставить? Ответ на последний вопрос нередко зависит от контекста применения карты. В одних случаях требуется видеть дороги и крытые автостоянки, в других случаях интерес представляют велосипедные дорожки и кофейни. Это означает, что карта должна быть составлена под конкретные нужды.

Любое описание, которое более краткое, чем то, о чем оно говорит, сталкивается с проблемой поиска подходящего уровня обобщения и предоставления средств конфигурирования. Такие описания называются *абстракциями.* Информатика занимается в основном вопросом определения и эффективного применения абстракций. Это наиболее ярко проявляется в алгоритме как абстракции самых разных вычислений, а параметры алгоритма определяют конкретное вычисление при выполнении данного алгоритма. Алгоритм манипулирует представлениями, которые также являются абстракциями, сохраняющими подробности, применяемые в алгоритме, и игнорирующими другие детали. Уровни абстракции алгоритма и его аргументов взаимосвязаны. Выбор подходящего уровня обобщения для алгоритма нередко требует компромисса между обобщенностью и эффективностью. Для обработки обширного ряда входных данных часто требуется более высокий уровень их абстракции, а это означает, что в алгоритме будет использоваться меньше подробностей (что может сопровождаться меньшей эффективностью).

Алгоритм выражается с помощью языка, а выполняется компьютером. Все эти понятия также используют абстракции. И наконец, для абстрагирования от отдельных вычислений в алгоритмах требуется абстракция понятий времен*ной* и пространственной эффективности, поскольку описание характеристики эффективности алгоритма не должно зависеть от объема разных входных данных.

В этой главе наглядно демонстрируется, что абстракция распространяется на все основные понятия информатики. Сначала в ней обсуждаются вопросы, возникающие в связи с определением и применением абстракций, и с этой целью рассматривается абстракция историй, упоминаемых в этой книге. Затем в главе поясняется, каким образом абстракция применяется в алгоритмах, представлениях, исполняющих средах, компьютерах и языках.

Как сделать длинный рассказ короче

В этой книге упоминались самые разные истории: сказка, детектив, приключение, музыкальная фантазия, романтическая комедия, научно-фантастическая комедия и научно-фантастический роман. И хотя все эти истории сильно разнятся,

у всех них имеется ряд общих свойств. В частности, все они разворачиваются вокруг одного или нескольких главных героев, которым приходится сталкиваться с трудностями и преодолевать их, и все эти истории оканчиваются счастливо. Все эти истории, кроме одной, существуют в книжном варианте, и во всех, кроме одной, в той или иной форме действует волшебство или сверхъестественная сила. И хотя в кратких описаниях каждой из этих историй опущены подробности, этих описаний достаточно, чтобы помочь отличить их, например, от спортивных репортажей. Однако между уровнем детализации описания и количеством примеров его применения заметен явный компромисс. Такой же компромисс возможен между количеством предоставляемых подробностей и временем, требующимся для понимания описания. Чем больше подробностей — тем длиннее описание. Данное затруднение можно разрешить, введя наименования для отдельных описаний, например *детективная история*, в которой главным героем является сыщик, расследующий преступление. Ключевые фразы, киноанонсы и прочие краткие описания важны потому, что они служат эффективным средством предоставления важной информации о том, что ожидается в конкретной истории; кроме того, они помогают быстрее принимать решения. Так, если вам не нравятся детективные истории, то вы можете предвидеть, что чтение повести *Собака Баскервилей* не доставит вам особого удовольствия.

Каким образом тогда сформировать краткую сводку о собрании историй? Для этого можно было бы начать со сравнения двух историй и запоминания всех общих для них черт. Затем можно было бы сравнить полученный результат с содержанием третьей истории и сохранить то общее, что имеется у всех трех историй, и т.д. В ходе такого процесса поэтапно отсеиваются все свойства, которые характерны лишь для конкретных историй, а сохраняются лишь общие для всех историй свойства. Такое исключение отличающихся подробностей и называется *абстракцией*, о которой иногда еще говорят как об "абстрагировании от подробностей" [1].

В информатике описание, получаемое в результате абстрагирования, называется *абстракцией*, и примерами тому служат *экземпляры* абстракции. Употребление одного и того же наименования *абстракция* для обозначения процесса и его результата может вызвать недоразумение. Почему бы тогда не воспользоваться похожим термином, например *обобщение*? Такой вариант кажется вполне применимым, поскольку обобщение также обозначает сводку, соответствующую целому ряду конкретных примеров. Но в информатике термин *абстракция* означает нечто большее, чем просто обобщение. Помимо обобщения самого описания, абстракция, как правило, имеет наименование и содержит один или несколько параметров, которые могут принимать в экземплярах конкретные значения. Имя и параметры абстракции называются *интерфейсом*, который предоставляет механизм использования абстракции, а параметры связывают ключевые элементы

экземпляров в абстракции. В то время как обобщение управляется только экземплярами, потребность в определении интерфейса делает абстракцию более спланированным процессом, включающим принятие решений относительно опускаемых деталей. Например, обобщение истории о том, "Как главный герой преодолевает трудности", можно довести до абстракции, присвоив ей наименование *Story* (история) и идентифицировав ее параметры protagonist (главный герой) и challenge (трудности):

```
Story(protagonist; challenge) = Как protagonist решает задачу challenge
```

Подобно параметрам алгоритма, параметры абстракции *Story* используются в описании как заполнители, вместо которых подставляются соответствующие аргументы, к которым должна применяться абстракция *Story*. (Употребление существительного protagonist в единственном числе не означает, что героев не может быть несколько.)

В истории Гензеля и Гретель главными героями являются Гензель и Гретель, а решаемая ими задача состоит в поиске обратного пути домой. Это обстоятельство можно выразить, применив абстракцию *Story* к соответствующим значениям параметров, как показано ниже. В данном применении абстракции обозначается история двух главных героев, Гензеля и Гретель, решающих задачу поиска обратного пути домой.

```
Story(Гензель и Гретель; поиск пути домой)
```

Над тем, что здесь совершено, стоит поразмыслить. Допустим, что вам требуется быстро объяснить кому-то суть истории Гензеля и Гретель. С этой целью вы можете сначала сказать, что это история, т.е. сослаться на абстракцию *Story*. Но это подействует, конечно, лишь в том случае, если другому человеку известно, что такое история, т.е. если он понимает абстракцию *Story*. В этом случае ссылка на абстракцию вызывает в другом человеке воспоминание описания того, что такое история. Далее вы предоставляете подробности, чтобы заполнить роли, представленные параметрами protagonist и challenge, в результате чего обобщенное описание истории превращается в более конкретное.

Технически применение абстракции *Story* приводит к замене наименования абстракции ее определением и к подстановке значений "Гензель и Гретель" и "поиск пути домой" вместо двух параметров в данном определении (см. главу 2, "От слов к делу: когда действительно происходит вычисление"). Такая подстановка приводит к приведенному ниже экземпляру.

Как Гензель и Гретель решают задачу поиска пути домой

Взаимосвязь между экземпляром и абстракцией наглядно показана на рис. 15.1.

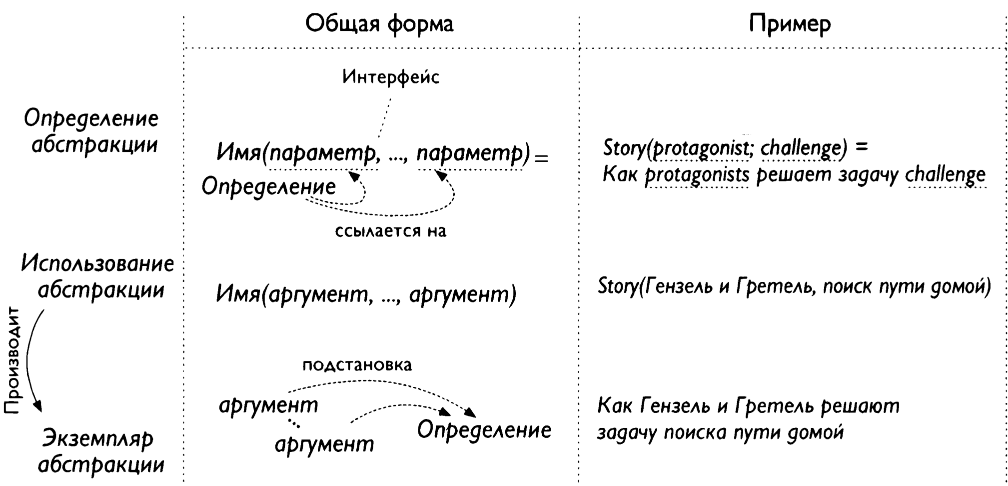


Рис. 15.1. Определение и применение абстракции. В определении абстракции ей присваивается наименование и устанавливаются ее параметры, на которые выполняются ссылки в ее определении. Наименование и параметры абстракции образуют ее интерфейс, описывающий порядок применения абстракции с использованием ее имени и аргументов для ее параметров. В результате такого применения абстракции формируется экземпляр, получаемый путем подстановки аргументов вместо параметров в определении абстракции

Как только краткое описание истории Гензеля и Гретель будет предоставлено на языке абстракции `Story`, для более краткого обращения к этому описанию ему можно будет присвоить соответствующее имя. В данном случае имя истории совпадет с именами ее главных героев:

```
Гензель и Гретель = Story(Гензель и Гретель; поиск пути домой)
```

Это уравнение гласит, что `Гензель и Гретель` — это история Гензеля и Гретель, решающих задачу поиска пути домой. В данном случае наименование истории совершенно случайно совпадает с именами ее главных героев (хотя это происходит не так уж редко). Далее приведен пример, в котором такое совпадение отсутствует: `День Сурка` — это название истории *День Сурка*.

```
День Сурка = Story(Фил Коннорс; побег из бесконечно повторяющегося дня)
```

И в этом случае экземпляр, получающийся в результате применения абстракции, может быть получен заменой имени абстракции ее определением и подстановкой конкретных значений вместо параметров:

```
День Сурка = Как Фил Коннорс решает задачу побега
             из бесконечно повторяющегося дня
```

В истории Гензеля и Гретель поиск их пути домой — далеко не единственная трудность, которую им приходится преодолевать. Еще один яркий эпизод в их

истории связан с тем, что им нужно каким-то образом избавиться от ведьмы, которая хочет их съесть. Следовательно, историю Гензеля и Гретель можно описать и таким образом:

```
Story(Гензель и Гретель; бегство от ведьмы)
```

В этом описании используется та же самая абстракция `Story`, только в ней изменен аргумент для второго параметра. В связи с этим возникает ряд вопросов по поводу абстракций. Во-первых, что делать с неоднозначностью в абстракциях? Историю Гензеля и Гретель можно рассматривать как экземпляр абстракции `Story` разными путями. Оба приведенных примера предоставляют правильные сведения о данной истории, так что ни один из них нельзя считать более правильным, чем другой. Означает ли это, что определение абстракции неверно? И во-вторых, абстракция `Story` не только *абстрагируется* от подробностей одной конкретной задачи, но и *сосредоточивается* на ней (по крайней мере, для истории, в которой таких задач несколько). Например, в каждом из описаний истории Гензеля и Гретель с помощью абстракции `Story` не упоминается по крайней мере одна задача. В связи с этим возникает следующий вопрос: можно ли определить абстракцию `Story` таким образом, чтобы учесть в ней несколько задач, решаемых в истории? Сделать это нетрудно, но каким для такой абстракции должен быть подходящий уровень детализации?

Скажите, когда

Когда абстракция оказывается достаточно общей, чтобы охватить все экземпляры? Когда она опускает слишком много подробностей и теряет свою точность? Всякий раз, образуя абстракцию, нам приходится решать, насколько общей она должна быть и сколько подробностей она должна предоставлять. С одной стороны, мы могли бы вместо абстракции `Story` воспользоваться описанием "последовательность событий", чтобы охарактеризовать истории, упоминаемые в этой книге. И это было бы точное описание (хотя и не идеально точное — некоторые важные особенности в таком описании оказались бы опущены). С другой стороны, мы могли бы воспользоваться такими более конкретными абстракциями, как "сказка" или "комедия". И хотя такие абстракции предоставляют больше подробностей, чем абстракция `Story`, они недостаточно общие, чтобы их можно было применять ко всем историям. Фактически даже довольно общая абстракция `Story` оказывается недостаточно общей, чтобы охватить истории, в которых главным героям не удается решить стоящую перед ними задачу. Этот недостаток можно исправить, введя еще один параметр, вместо которого может быть подставлено значение "решает" или "не решает" — в зависимости от конкретной ситуации. Пригодность такого обобщения абстракции `Story` зависит от ее

применения. Так, если случай "не решает" никогда не возникает, то дополнительная обобщенность излишня, и более предпочтительной окажется текущее, более простое определение данной абстракции. Но контекст применения абстракции может измениться, а следовательно, в выбранном уровне обобщенности абсолютной уверенности нет.

Помимо поиска подходящего уровня обобщенности, при определении абстракции возникает еще одно затруднение, связанное с тем, что приходится решать, сколько подробностей следует предоставить в описании и сколько параметров использовать. Например, в абстракцию `Story` можно было бы ввести параметр, отражающий способ преодоления трудностей главным героем истории. Добавление параметров в абстракцию, с одной стороны, делает ее более выразительной, поскольку предоставляет механизм раскрытия тонких отличий в разных экземплярах. С другой же стороны, интерфейс абстракции при этом становится более сложным и требует при применении абстракции указания дополнительных аргументов. Более сложный интерфейс не только усложняет применение абстракции, но и затрудняет ее понимание, поскольку возрастает количество подстановок аргументов вместо параметров. А это противоречит одной из главных причин использования абстракций, которая состоит в предоставлении кратких и легко усваиваемых описаний.

Баланс между сложностью интерфейса и точностью абстракции является одной из главных задач в области разработки программного обеспечения. Эту задачу разработчиков программного обеспечения можно было бы охарактеризовать с помощью следующего экземпляра абстракции `Story`:

```
Разработка ПО = Story(Разработчики; поиск верного уровня абстракции)
```

Безусловно, программистам приходится преодолевать и другие трудности, и некоторые из них могут быть так же ясно и кратко описаны с помощью абстракции `Story`. Ниже приведен экземпляр данной абстракции, описывающий еще одну задачу, которую приходится решать каждому программисту:

```
Правильное ПО = Story(Программисты; поиск и устранение ошибок)
```

Вопрос поиска подходящего уровня абстракции встает и для абстракции `Story`. Как пояснялось ранее, экземпляр `Гензель и Гретель` абстракции `Story` можно определить двумя способами. Выбор одного экземпляра, сосредоточенного на одной задаче, означает абстрагирование от (или игнорирование) другого экземпляра. Но что если для более полного раскрытия истории Гензеля и Гретель требуются оба экземпляра? Это можно сделать по-разному. С одной стороны, можно просто упомянуть рядом оба экземпляра:

```
Story(Гензель и Гретель; поиск пути домой) и
Story(Гензель и Гретель; избавления от ведьмы)
```

Такое представление выглядит не совсем складно. В частности, повторяющееся упоминание главных героев и абстракции `Story` кажется избыточным. Это становится очевидным при выполнении подстановки:

Как Гензель и Гретель решают задачу поиска своего пути домой и как Гензель и Гретель решают задачу избавления от ведьмы

С другой стороны, можно было бы просто объединить обе задачи в один аргумент для параметра *challenge* следующим образом:

```
Story(Гензель и Гретель; поиск пути домой и избавление от ведьмы)
```

Это уже сравнительно неплохо. Обратите внимание на то, что то же самое было сделано и с именами главных героев данной истории. В частности, имена `Гензель и Гретель` были сгруппированы и подставлены одним блоком вместо параметра *protagonist*. Но можно также заметить, что удобство упоминания одного или нескольких главных героев в абстракции не дается даром. Формально в определении абстракции `Story` существительное *protagonist* и глагол `решает` следует употреблять по правилам грамматики для обозначения как одиночных, так и многих объектов, как *protagonist(s)* и `решает(ют)` (то же самое следует сделать и для описания задачи или задач). Поэтому было бы неплохо, если бы абстракция `Story` позволяла формировать на каждый случай отдельные, грамматически правильные экземпляры.

Этого можно добиться, определив абстракцию `Story` таким образом, чтобы ее первым параметром служил список главных героев. И тогда получатся два определения данной абстракции, несколько различающихся в зависимости от того, обозначает ли первый параметр одного главного героя или список из двух героев, как показано ниже [2]:

```
Story(protagonist; challenge) = Как protagonist решает задачу challenge
Story(protagonist1 → protagonist2; challenge) = Как protagonist1 и
                                    protagonist2 решают задачу challenge
```

Если теперь применить абстракцию `Story` к одному главному герою истории (например, к Филу Коннорсу), то будет выбрано первое ее определение с глаголом в единственном числе. А если применить данную абстракцию к списку из двух главных героев (например, Гензель → Гретель), то будет выбрано второе ее определение с глаголом во множественном числе. В последнем случае список будет разложен на два своих элемента, между которыми вставляется союз *и*.

По-видимому, абстракция `Story` предоставляет средства для составления предложений на естественном языке. В главе 8, “Сквозь призму языка”, было продемонстрировано, что пригодным для этой цели механизмом является грамматика. Так нельзя ли иначе определить грамматику для кратких описаний историй? Можно. Приведем один из возможных вариантов грамматики, соответствующей последнему определению абстракции `Story` [3]. Помимо минимальных отличий

в условных обозначениях и, в частности, употребления стрелки вместо знака равенства или пунктирных рамок для обозначения нетерминалов вместо подчеркивания параметров пунктиром, оба приведенных ниже грамматических механизма, по существу, действуют одинаково, подставляя значения (или терминалы) вместо параметров (или нетерминалов).

```
story → Как protagonist решает задачу challenge
story → Как protagonists решают задачу challenge
protagonists → protagonist и protagonist
```

В табл. 8.1 главы 8, "Сквозь призму языка", приводится сравнение грамматик, уравнений и алгоритмов. При этом показаны общие роли составных частей разных формализаций, а следовательно, подчеркивается, что грамматика, уравнения и алгоритмы являются разными, хотя и сходными механизмами описания абстракций.

Как обсуждалось ранее, разработка абстракции — дело непростое. Сталкиваясь с новыми прецедентами использования, можно распознать в них изменяющиеся требования, вынуждающие внести коррективы в определение абстракции. Такие коррективы могут привести к более общей абстракции или же к такой абстракции, которая раскрывает разные подробности. В некоторых случаях это может обусловить изменения в интерфейсе, когда, например, вводится новый параметр или изменяется его тип. Примером тому служит замена единственного значения параметра *protagonist* списком. Когда изменяется интерфейс абстракции, должны быть изменены и все существующие варианты ее применения, чтобы соответствовать новому интерфейсу. Это требует немало труда и может давать эффект цепной реакции, вызывающей изменения и в других интерфейсах. Поэтому разработчики программного обеспечения стараются не вносить изменения в интерфейсы, насколько это возможно, прибегая к данному средству лишь в крайнем случае.

Последовательный ряд абстракций

Рассмотрим последнее определение абстракции *Story*, в котором применяется список главных героев. Несмотря на то что уравнения в этом определении работают только со списками, состоящими из одного или двух элементов, данное определение можно без труда расширить до списков с произвольным количеством элементов, используя циклы или рекурсию, как демонстрировалось в главах 10, "Намылить, смыть, повторить", и 12, "Своевременный стежок вычисляется впрок". Применение отдельных уравнений для того, чтобы различать разные случаи, а также идея обработки списков указывает на то, что абстракция *Story* может фактически стать алгоритмом для производства кратких описаний историй. Но здесь все сложнее, чем кажется на первый взгляд, поэтому взаимосвязь между алгоритмами и абстракциями будет рассмотрена далее более подробно.

Если принять во внимание особую важность абстракции в информатике, то не удивительно, что алгоритм как одно из ее центральных понятий сам служит примером абстракции. В частности, алгоритм описывает общность целого ряда сходных вычислений, будь то следование по камешкам или сортировка списков, как показано на рис. 15.2 [4]. Когда алгоритм выполняется с разными аргументами, подставляемыми вместо его параметров, в конечном итоге выполняются разнообразные вычисления.

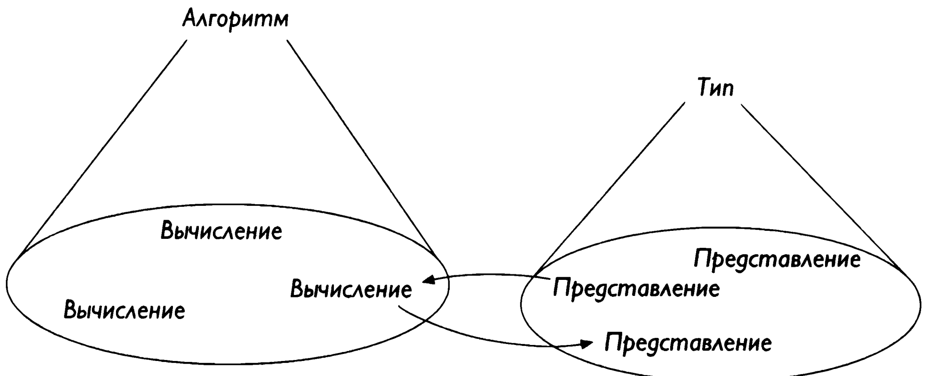
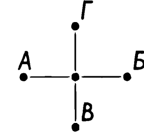


Рис. 15.2. Алгоритм представляет собой абстрагирование от отдельных вычислений. Каждый алгоритм преобразует представления, а тип абстрагирует от отдельных представлений. Если **`Input`** *— это тип представлений, принимаемых алгоритмом, а* **`Output`** *— тип производимых им представлений, то данный алгоритм имеет тип* **`Input`** $\rightarrow$ **`Output`**

Абстракция `Story` представляется апостериорным обобщением, т.е. приведение описания осуществляется *после* рассмотрения многих существующих историй. Нечто подобное может иногда происходить и с алгоритмами. Например, после того как вы неоднократно готовили какую-нибудь еду, меняя ингредиенты и видоизменяя способ приготовления для улучшения конечного результата, вы можете, наконец, решиться составить рецепт, чтобы приобретенный вами кулинарный опыт можно было повторить в будущем. Но во многих других случаях алгоритм представляет собой решение нерешенной задачи и создается еще до всяких вычислений. Именно это и придает абстракции алгоритма особую силу. Помимо описания вычислений, которые уже произведены, алгоритм позволяет формировать по мере надобности совершенно новые вычисления. Алгоритм способен решать новые, не встречавшиеся прежде задачи. Стоит вам рассмотреть некоторых произвольных главных героев и конкретную задачу в контексте абстракции `Story`, как вы получите начало новой истории.

Этот момент дополнительно иллюстрируется следующей аналогией. Рассмотрим простую дорожную сеть и допустим, что были построены две дороги для



сообщения между городами А и Б и городами В и Г соответственно. При этом обе дороги пересекаются, и поэтому построен также перекресток. Но в результате неожиданно становятся возможными новые сообщения между городами, позволяющие переезжать из города А в город В, из города Б в город Г и т.д. Такая дорожная сеть упрощает не только те поездки, для которых она была построена, но потенциально и многие другие поездки, которые не были предусмотрены.

Наблюдение, что алгоритмы являются абстракциями, можно пояснить двумя способами. Во-первых, разработка и применение алгоритмов не только получает все преимущества абстракций, но и несет бремя всех затрат. В частности, задача поиска подходящего уровня абстракции связана с разработкой алгоритмов, поскольку обобщенность нередко оказывает влияние на эффективность алгоритма. Например, для сортировки слиянием требуется линейно-логарифмическое время. Сортировка слиянием обрабатывает любой список элементов и требует лишь возможности сравнивать элементы. Следовательно, это наиболее общий метод сортировки, который можно вообразить и широко применять. Но если элементы сортируемого списка взяты из некоторой конкретной предметной области, то к ним, может быть, можно применить блочную сортировку (см. главу 6, "Сортировка алгоритмов сортировки"), которая выполняется быстрее — в течение линейного времени. Таким образом, помимо потенциального компромисса между обобщенностью и точностью, алгоритмам присущ компромисс между обобщенностью и эффективностью.

И во-вторых, при разработке абстракций можно использовать алгоритмические элементы. Абстракция `Story` служит характерным тому примером. Единственное ее назначение — производить краткие описания историй, не подразумевая никаких вычислений. Но поскольку грамотно разработанная абстракция выявляет главные особенности отдельных историй через применение параметров, обнаруживается потребность в более удобном обращении со списками главных героев и трудностей. В частности, различение списков с разным количеством элементов оказывается удобным для производства специализированных описаний историй.

Выполнение алгоритмов равносильно функциональному поведению, и поэтому алгоритмы называются также *функциональными абстракциями*. Но алгоритмы являются не единственными примерами функциональных абстракций. Так, заклинания в книгах о Гарри Поттере являются функциональными абстракциями волшебства. Когда заклинание исполняется магом, оно вызывает волшебство. Всякий раз при выполнении заклинания оно вызывает разный эффект в зависимости от того, на кого или на что оно направлено и насколько хорошо оно исполнено. Подобно алгоритму, который выражается на каком-то языке и может быть выполнен только тем компьютером, который понимает этот язык, заклинание

выражается на языке волшебства, включающем в себя магические формулы, движения волшебной палочкой и пр. Заклинание может быть исполнено только опытным магом, который знает, как его исполнять. Еще одной абстракцией волшебства являются зелья. Они отличаются от заклинаний тем, что исполнять их намного легче. Чтобы привести зелье в действие, не требуется обращаться к чародею. Это может сделать кто угодно — даже магглы.

Многие машины также являются функциональными абстракциями. Например, карманный калькулятор служит абстракцией арифметических операций. Подобно тому, как зелье распространяет доступ к волшебству за пределы круга магов, калькулятор распространяет доступ к арифметике на тех людей, которые не обладают необходимыми навыками для выполнения арифметических расчетов. Этот доступ может быть также расширен за счет тех людей, у которых эти навыки есть — благодаря ускорению выполнения расчетов. Другими примерами служат кофеварки или будильники — машины, которые могут быть специально настроены для надежного выполнения конкретных функций. История средств передвижения наглядно показывает, насколько машины повышают эффективность абстрагированного метода (в данном случае — передвижения), а иногда и упрощают интерфейс для распространения доступа на многих людей. Так, повозки и кареты требовали лошадей и поэтому были относительно медленными средствами передвижения. Автомашины стали значительно более совершенными средствами передвижения, но и они требуют навыков вождения. Автоматические коробки передач, ремни безопасности, навигационные системы — все это делает автомашины более доступным (в смысле необходимых умений, не в смысле стоимости) и безопасным средством передвижения. В ближайшие годы можно ожидать еще большего распространения доступа к средствам передвижения с появлением самоуправляемых автомашин.

Один тип для всего

Алгоритмы и машины (заклинания и зелья) служат примерами функциональных абстракций, поскольку включают в себя некоторую форму функциональных возможностей. Пассивные представления, преобразуемые в вычислениях, также подлежат абстракции, которая называется *абстракцией данных*. Концепция представления по своей природе является абстракцией, поскольку представление определяет некоторые свойства знаков, обозначающих нечто (см. главу 3, “Тайна знаков”), активно пренебрегая при этом другими свойствами, т.е. абстрагируясь от них.

Когда Гензель и Гретель ищут свой путь домой по камешкам, размер и цвет камешков не имеет особого значения. При использовании камешков в качестве представления мест расположения создается абстракция, игнорирующая отличия

в размере и цвете, но вместо этого сосредоточенная на их свойстве отражать лунный свет. А когда мы говорим, что Гарри Поттер — маг, то подчеркиваем то обстоятельство, что он может творить волшебство. Нас не особенно интересует его возраст или то, что он носит очки, или же какие-нибудь другие любопытные сведения о нем. Называя Гарри Поттера магом, мы абстрагируемся от всех этих подробностей. То же самое происходит, когда мы указываем на то, что Шерлок Холмс — сыщик, а доктор Браун — ученый. При этом мы выделяем свойства, обычно связанные с этими понятиями, временно пренебрегая всем остальным в описываемом человеке.

Такие понятия, как *маг*, *детектив* и *ученый*, безусловно, являются типами. Они несут в себе дополнительные значения свойств, которые обычно считаются истинными для любого отдельного члена подобного типа. Понятия главного героя и трудности в абстракции `Story` также являются типами, поскольку вызывают в уме определенные образы, которые требуются абстракции `Story` для передачи ее смыслового значения. По-видимому, тип *главного героя* является более общим, чем, скажем, тип *мага* или *сыщика*, поскольку предоставляет меньше подробностей. Такое представление подтверждает то обстоятельство, что два последних типа могут быть подставлены вместо первого типа. Но это еще не все. Возьмем в качестве примера Волан-де-Морта. Он также маг, но не главный герой. Напротив, в книгах о Гарри Поттере он главный антигерой. А поскольку и главный их герой Гарри Поттер, и главный антигерой Волан-де-Морт являются магами, то тип *мага*, по-видимому, оказывается более общим, поскольку в нем пренебрегаются подробности, по которым можно отличить главного героя от антигероя. Следовательно, ни *главный герой*, ни *маг* в общем случае не могут рассматриваться как абстрактный тип (что не особенно удивляет, поскольку эти типы происходят из разных предметных областей, а именно — из историй и волшебства).

В пределах одной предметной области типы нередко имеют более ясную организацию, располагаясь по иерархиям. Гарри Поттер, Драко Малфой и Северус Снейп — все это члены сообщества из школы магов в Хогвартсе, но только Гарри Поттер и Драко Малфой являются ее учениками. Если вы учитесь в школе магов в Хогвартсе, то, очевидно, являетесь *членом сообщества магов Хогвартса*. А поскольку Гарри Поттер, но не Драко Малфой, учится на факультете Гриффиндора, то тип *ученик школы в Хогвартсе* является более общим, чем *студент факультета Гриффиндора*. Аналогично волшебство является более абстрактным, чем заклинание, которое, в свою очередь, более абстрактно, чем заклинание для телепортации или чары заступника.

Типы, обнаруживаемые в языках программирования, являются, вероятно, наиболее очевидной формой абстракции данных. Так, 2 и 6 являются разными числами, но у них немало общего. В частности, их можно разделить на два, сложить с другими числами и т.д. Таким образом, можно пренебречь их различиями и

сгруппировать их с другими числам в тип `Number`. Этот тип абстрагируется от свойств отдельных чисел и раскрывает общие свойства всех своих членов. В частности, с помощью типа можно охарактеризовать параметры алгоритма, а затем воспользоваться этим для проверки согласованности алгоритма с помощью контроля типов (см. главу 14, "Волшебный тип"). Таким образом, абстракция данных неразрывно связана с функциональной абстракцией, поскольку алгоритм абстрагируется от отдельных значений с помощью параметров. Но во многих случаях вместо параметров нельзя подставить ничего вообразимого, но требуется, чтобы алгоритм мог манипулировать представлениями аргументов. Например, параметр, умножаемый на 2, должен быть числовым, и здесь вступают в игру типы как абстракция данных. Тип `Number` может также считаться более абстрактным, чем много других специализированных числовых типов, например тип всех четных чисел.

И наконец, помимо таких (простых) типов, как `Number`, абстракция данных распространяется и на типы данных (см. главу 4, "Записная книжка сыщика"). Тип данных определяется исключительно теми операциями, которые он предоставляет, а также их свойствами. Подробности представления при этом игнорируются, т.е. от них абстрагируются, а это означает, что тип данных является более абстрактным, чем структура данных, которая его реализует. Например, стек может быть реализован с помощью списка или массива, но подробности этих структур данных, а следовательно, их различия не видны, когда они применяются для реализации стеков.

Любая удачно выбранная абстракция данных выделяет те свойства представления, с помощью которых поддерживается вычисление. Более того, подобная абстракция игнорирует и скрывает свойства, которые могли бы помешать вычислению.

Время абстракции

Как пояснялось в главе 2, "От слов к делу: когда действительно происходит вычисление", о времени выполнения алгоритма сообщается иначе, чем это делает фитнес-трекер, отслеживающий физическую активность, показывая время вашего последнего забега на 10 км. Сообщать о времени выполнения алгоритмов в секундах (минутах или часах) было бы не очень полезно, поскольку это время зависит от конкретного компьютера. Если один и тот же алгоритм выполняется на медленном и быстром компьютерах, то время его выполнения окажется разным. Сравнивать свое время пробега 10 км со временем товарища имеет смысл, поскольку это дает сведения об относительной эффективности двух компьютеров, т.е. бегунов. Но время не сообщает ничего значительного об эффективности выполнения самого алгоритма, поскольку вы и ваш товарищ по забегу выполняете один и тот же алгоритм.

Следовательно, целесообразно абстрагироваться от конкретного времени, чтобы измерять сложность алгоритма количеством сделанных шагов. Такая мера не зависит от быстродействия компьютера, а следовательно, и от уровня развития техники. Допустим, что вы бежите более-менее постоянным шагом. В таком случае вы будете всегда совершать одно и то же количество шагов в своем забеге на 10 км, независимо от конкретных обстоятельств. Употребление количества движений, проделанных фиксированным шагом, в качестве меры позволяет абстрагироваться от конкретных характеристик бегуна, а следовательно, более устойчиво охарактеризовать бег. В действительности задание количества шагов — это всего лишь другой способ выразить забег длиной 10 км. Длина забега, безусловно, служит лучшей мерой его сложности, чем его продолжительность, поскольку она абстрагируется от разных скоростей движения разных бегунов или даже одного и того же бегуна в разное время.

Но несмотря на то что количество шагов или операций более абстрактно, чем время, оно все-таки служит слишком конкретной мерой сложности алгоритма, поскольку это количество изменяется в зависимости от входных данных алгоритма. Например, чем длиннее список, тем дольше он сортируется или обнаруживается его минимальный элемент. Аналогично для забега на 10 км требуется больше шагов, чем для забега на 8 км. Напомним, что цель состоит в том, чтобы охарактеризовать сложность алгоритма в общем, а не его производительность при конкретных входных данных. Поэтому и не ясно, для каких именно входных данных следует сообщать о количестве проделанных шагов. Можно, конечно, представить себе создание таблицы, в которой показано количество шагов для нескольких конкретных случаев, выбранных в качестве примеров, но и тогда не будет ясно, какие именно примеры следует выбрать.

Таким образом, абстракция времени работы алгоритмов продолжается, пренебрегая конкретным количеством проделанных шагов. Вместо этого она сообщает, насколько возрастает количество шагов при увеличении объема входных данных. Так, если алгоритму потребуется в два раза больше шагов при увеличении объема входных данных вдвое, то до такой же степени возрастет и время выполнения алгоритма. Как пояснялось в главе 2, “От слов к делу: когда действительно происходит вычисление”, такое поведение времени выполнения алгоритма называется *линейным.* Именно так и происходит при поиске минимального элемента в списке или в забеге [5]. Даже если время выполнения возрастает на множитель, больший, чем два, сложность алгоритма по-прежнему считается линейной, поскольку зависимость количества проделанных шагов от объема входных данных выражается умножением на постоянный множитель. Именно так и происходит с количеством шагов, совершаемых Гензелем и Гретель в поисках своего пути домой. Время выполнения такого алгоритма возрастает более чем в два раза, поскольку камешки разбросаны один от другого за несколько шагов. Категория времени

выполнения линейных алгоритмов абстрагируется и от этого множителя, и поэтому алгоритм Гензеля и Гретель по-прежнему считается линейным.

К самым важным преимуществам абстракции времени выполнения относится ее способность сообщать, какие задачи легко разрешимы и какой алгоритм следует выбрать для решения конкретной задачи. Например, алгоритмы с экспоненциальным временем выполнения пригодны для обработки только небольших объемов входных данных, а задачи, которые можно решить только по алгоритмам с экспоненциальным временем выполнения, известны как трудноразрешимые (см. главу 7, "Трудноразрешимые задачи"). А если имеется несколько алгоритмов для решения одной и той же задачи, то в общем случае следует выбрать тот из них, который обладает лучшим показателем временн*о́*й сложности. Например, линейно-логарифмической сортировке слиянием обычно отдается предпочтение перед квадратичной сортировкой вставками (см. главу 6, "Сортировка алгоритмов сортировки"). Абстракция времени наглядно подытожена на рис. 15.3.

Рис. 15.3. Абстракция времени. Чтобы абстрагироваться от разного быстродействия компьютеров, здесь используется количество шагов, которые требуются алгоритму для выполнения, в качестве меры его времени выполнения. А для того чтобы абстрагироваться от разного количества шагов, требующихся для обработки разных входных данных, время выполнения алгоритма измеряется тем, как быстро оно возрастает при увеличении объема входных данных

Машинный язык

Сам алгоритм не способен произвести вычисление. Как пояснялось в главе 2, “От слов к делу: когда действительно происходит вычисление”, алгоритм должен быть выполнен компьютером, понимающим язык, на котором написан алгоритм. Любая инструкция, применяемая в алгоритме, должна входить в набор инструкций, которые способен обрабатывать компьютер.

Привязать алгоритм к конкретному компьютеру посредством языка непросто по ряду причин. Во-первых, независимо спроектированные компьютеры, скорее всего, будут понимать разные языки, а это означает, что алгоритм, написанный на языке, понятном для выполнения на одном компьютере, может оказаться непонятным для выполнения на другом компьютере. Так, если бы Гензель или Гретель написали свой алгоритм на немецком языке для поиска обратного пути домой по камешкам, то ребенок, выросший во Франции или Англии, не смог бы его выполнить, не изучив предварительно немецкий язык. И во-вторых, со временем языки, применяемые компьютерами, изменяются. И хотя это может не представлять особых трудностей для людей, надежно знающих прежние, устаревшие формы языка, для машин это, безусловно, может стать серьезным препятствием, из-за которого они вообще не смогут выполнить алгоритм, даже если в нем слегка изменена лишь одна инструкция. Непрочная языковая связь между алгоритмом и компьютером, по-видимому, затрудняет обмен алгоритмами. Правда, программное обеспечение совсем не обязательно переписывать всякий раз, когда на рынке появляется новая модель компьютера. И это оказывается возможным благодаря следующим двум формам абстракции: *трансляции языка* и *абстрактной машине.*

Чтобы наглядно продемонстрировать принцип действия абстрактной машины, рассмотрим алгоритм вождения автомашины. Вероятно, вас учили водить конкретную модель автомашины, хотя вы сможете водить и другие автомобили. Приобретенные навыки вождения не привязаны к конкретной модели автомашины, а более абстрактны и могут быть описаны с помощью таких понятий, как руль, педаль газа и тормоза. Абстрактная модель автомашины реализуется в разнообразных видах конкретных автомашин, различающихся деталями, но предоставляющих доступ к функциональным возможностям с помощью общего языка вождения.

Абстракция применяется ко всем видам машин. Например, чтобы абстрагироваться от подробностей конструкции конкретной кофеварки для приготовления кофе по-турецки, по-французски или по-итальянски, можно сказать, что для приготовления кофе от такой машины требуется способность смешивать горячую воду с молотым кофе в течение определенного периода времени и затем отделять твердые частицы от жидкости. Алгоритм приготовления кофе может быть описан в терминах такой абстрактной кофеварки, которая все равно остается достаточно конкретной, чтобы быть реализованной в разных машинах для приготовления кофе. Безусловно, у абстрактной машины имеются свои ограничения.

В частности, кофеваркой нельзя воспользоваться для выполнения алгоритма вождения, а с помощью автомашины нельзя приготовить кофе. Тем не менее абстрактные машины являются важным средством для разъединения языков и конкретных компьютерных архитектур (рис. 15.4).

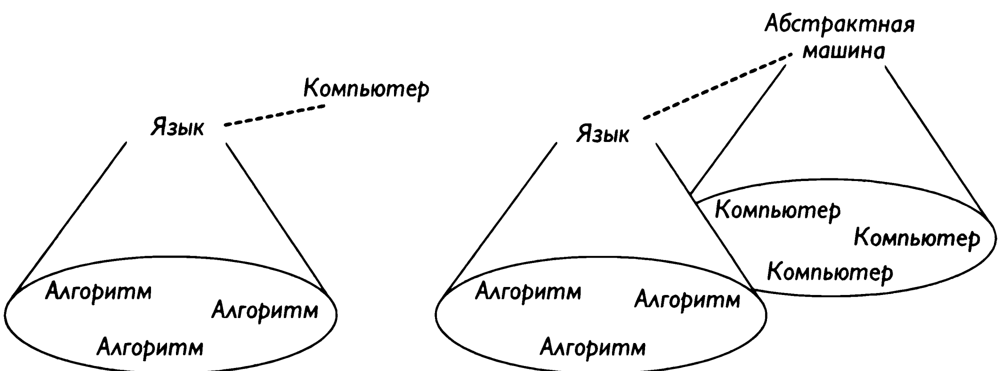


Рис. 15.4. Абстрактная машина служит абстракцией для конкретных компьютеров. Предоставляя упрощенный и общий интерфейс, абстрактная машина обеспечивает независимость алгоритмических языков от архитектуры конкретных компьютеров и расширяет диапазон компьютеров, способных выполнять алгоритмы на этих языках

Самой известной абстракцией машины для вычислений является *машина Тьюринга*, названная в честь известного британского математика и первопроходца в области информатики Алана Тьюринга, который изобрел эту машину в 1936 году и применил ее для формализации понятий вычисления и алгоритма. Машина Тьюринга состоит из ленты, разделенной на ячейки, каждая из которых содержит символ. Доступ к этой ленте осуществляется через головку чтения/записи, которая может также перемещать ленту вперед и назад. Эта машина всегда находится в каком-то конкретном состоянии и управляется программой, задаваемой рядом правил, которые определяют, какой именно символ следует записать в текущую ячейку на ленте, в каком направлении перемещать ленту и в какое новое состояние перейти в зависимости от того, какой символ доступен в настоящий момент и каково текущее состояние машины. Машина Тьюринга была использована для доказательства неразрешимости задачи останова (см. главу 11, "Счастливый конец не гарантируется"). Любую программу можно транслировать в программу для машины Тьюринга, а это означает, что машина Тьюринга служит абстракцией для всех ныне существующих электронных вычислительных машин или просто компьютеров. Важность такого вывода состоит в том, что любое общее свойство, которое можно доказать для машины Тьюринга, присуще и любому другому существующему компьютеру.

Другая стратегия абстрагирования от конкретных компьютеров состоит в том, чтобы воспользоваться трансляцией языка, т.е. переводом с одного языка на

другой. Например, алгоритм прослеживания пути по камешкам можно перевести с немецкого языка на английский или французский, тем самым сняв языковой барьер и сделав данный алгоритм доступным для более широкого круга людей. Этот же подход применим и для языков программирования компьютеров. В действительности почти все написанные в настоящее время программы так или иначе транслируются перед выполнением на вычислительной машине. И это означает, что ни один из современных языков программирования не понятен компьютеру непосредственно и что каждый алгоритм должен быть транслирован на понятный вычислительной машине язык. Язык программирования абстрагируется от подробностей архитектуры конкретного компьютера и обеспечивает единообразный доступ для программирования обширного ряда компьютеров на одном языке. Следовательно, язык программирования служит абстракцией для компьютеров и делает разработку алгоритмов независимой от конкретных компьютеров. Если произведен новый компьютер, то для выполнения на нем всех существующих алгоритмов достаточно приспособить транслятор, чтобы получить измененный код для этого компьютера. Благодаря абстракции трансляции сама разработка языков программирования стала в значительной степени независимой от тех компьютеров, на которых будут выполняться программы, написанные на этих языках.

Абстракцию трансляции *Translate* можно определить разными способами. Как и при любой абстракции, в данном случае возникает следующий вопрос: от каких подробностей следует абстрагироваться и какие из них стоит раскрыть в качестве параметров в интерфейсе? Следующее определение абстрагируется от транслируемой программы (*program*), исходного языка, на котором она написана (*source*), а также целевого языка, на который она транслируется (*target*):

```
Translate(program; source; target) =
                    "Транслировать program из source в target"
```

Обратите внимание на то, что правая часть "Транслировать..." данного определения заключена в кавычки, поскольку трансляция (или перевод) — слишком сложный и длинный алгоритм, чтобы представить его здесь. Например, задача автоматического перевода с одного естественного языка на другой по-прежнему остается нерешенной. Трансляция же языков программирования — вполне понятная и решенная задача. Тем не менее трансляторы являются длинными и сложными алгоритмами, и по этой причине их подробности в приведенном выше определении не указаны.

В качестве примера ниже показано, как воспользоваться абстракцией *Translate*, чтобы перевести инструкцию по поиску камешков с немецкого языка на английский. Инструкция *Finde Kieselstein* — это элемент исходного (немецкого) языка, а результат перевода — инструкция *Find pebble* (Найти камешек), которая является элементом целевого (английского) языка.

```
Translate(Finde Kieselstein; немецкий; английский)
```

В книгах о Гарри Поттере для исполнения заклинаний и чар чародеи пользуются специальным языком. Некоторые заклинания могут быть переведены в соответствующее зелье, которое могут затем исполнить и магглы. Например, результат преобразования некоторых заклинаний может быть зафиксирован в оборотном зелье, выпив которое, можно изменить свою внешность. И если приготовить оборотное зелье, очевидно, совсем не просто даже опытным магам, то перевести ряд некоторых других заклинаний совсем нетрудно. Например, убийственное проклятие Авада Кедавра транслируется в любой обыкновенный смертоносный яд.

Каждая абстракция `Translate` сама является алгоритмом, что дает возможность абстрагироваться от всех трансляций (или переводов) на тот язык, на котором они могут быть выражены, подобно абстрагированию от всех алгоритмов с помощью языка. Именно этот случай показан на рис. 15.5, *слева вверху*. Поскольку каждый язык соответствует компьютеру или абстрактной машине, способной выполнять написанные на нем программы, язык может абстрагироваться от компьютеров, как показано на рис. 15.5, *справа вверху*.

Подобно тому, как машина Тьюринга является предельной абстракцией любой конкретной вычислительной машины, *лямбда-исчисление* служит абстракцией любого языка программирования. Лямбда-исчисление было изобретено американским математиком Алонсо Чёрчем (Alonzo Church). Оно состоит всего лишь из трех конструкций для определения абстракций, обращения к параметрам в определениях и получения экземпляров абстракций через подстановку аргументов вместо параметров, что очень похоже на то, что показано на рис. 15.1. Всякая программа на любом алгоритмическом языке может быть транслирована в программу лямбда-исчисления. Похоже, что теперь у нас имеются две разные предельные абстракции компьютеров: лямбда-исчисление и машина Тьюринга. Как же такое может быть? Оказывается, что эти две абстракции равнозначны, т.е. любую программу для машины Тьюринга можно транслировать в равнозначную программу лямбда-исчисления и обратно. Более того, было показано, что всякая известная формализация для выражения алгоритмов [6] оказывается не более выразительной, чем машина Тьюринга или лямбда-исчисление. Следовательно, любой алгоритм, по-видимому, может быть выражен в виде программы для машины Тьюринга или лямбда-исчисления. Это наблюдение известно как тезис *Чёрча-Тьюринга*, названный так в честь двух первопроходцев в информатике. Тезис Чёрча-Тьюринга имеет отношение к выразительности и охвату алгоритмов. Поскольку определение алгоритма основывается на принципе исполнительной команды, что интуитивно связано с возможностями человека, то алгоритм нельзя формализовать математически. Тезис Чёрча-Тьюринга является не теоремой, которую можно доказать, а скорее высказыванием об интуитивной концепции алгоритма. Тезис Чёрча-Тьюринга важен потому, что подразумевает следующее:

все, что можно знать об алгоритмах, можно выяснить, изучая машины Тьюринга и лямбда-исчисление. Этот тезис принят большинством специалистов в области информатики.

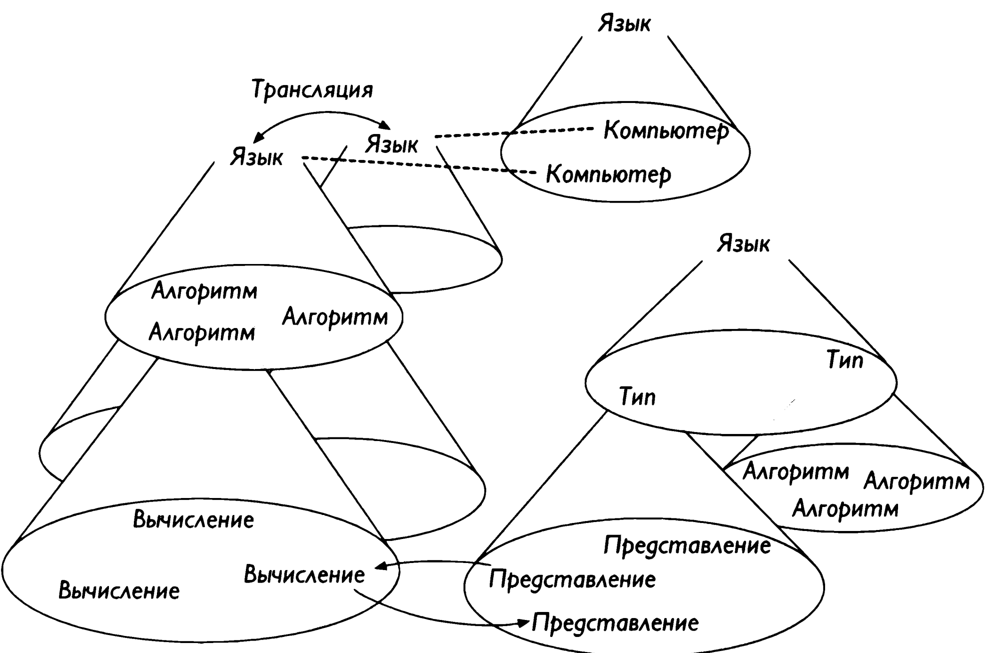


Рис. 15.5. Башня абстракций. Алгоритм является (функциональной) абстракцией от вычислений. Алгоритм преобразует представления, абстракциями (данных) которых являются типы. Приемлемые входные и выходные данные алгоритма также выражаются в виде типов. Каждый алгоритм выражается на языке, который абстрагируется от алгоритмов. Алгоритм трансляции позволяет переводить другие алгоритмы с одного языка на другой, а следовательно, делает их независимыми от конкретного компьютера или абстрактной машины, понимающей язык, на котором этот алгоритм выражен. Язык также служит абстракцией компьютеров, поскольку трансляция, по существу, устраняет различия в архитектурах компьютеров. Типы также выражаются как часть языка. Приведенная здесь иерархия абстракций наглядно показывает, что все абстракции в информатике выражаются на каком-нибудь языке

* * *

Вычисление применяется для систематического решения задач. И хотя электронные вычислительные машины позволяют справляться с беспрецедентным ростом объема и доступности вычислений, они служат лишь одним из инструментальных средств для вычисления. Но понятие вычисления является более общим и более широко употребляемым. Как было показано ранее, Гензель и Гретель уже знали, как выполнять алгоритм и успешно применять абстракции для

представления путей по камешкам. Шерлок Холмс был мастером знаков и представлений, умело манипулируя структурами данных для раскрытия преступлений. А Индиана Джонс не пользовался электронной вычислительной машиной в своих увлекательных поисках. Язык музыки не способен непосредственно решить ни одной задачи, но включает в себя все элементы синтаксиса и семантики, какие только можно вообразить. Филу Коннорсу, возможно, и не были известны теоретические основы информатики, тем не менее он столкнулся с той же проблемой, которая наглядно иллюстрирует фундаментальные ограничения, присущие вычислениям: неразрешимостью задачи останова. Марти Макфлай и доктор Браун жили рекурсивно, а Гарри Поттер открыл нам волшебную силу типов и абстракции.

Возможно, герои всех упомянутых в этой книге историй и не являются героями вычислений, но их истории немало рассказали нам о том, что же такое вычисление. Под конец книги рассмотрим еще одну историю — *историю информатики*. Это история об абстракциях, соперничающих за понятие вычисления. Главным ее героем является *алгоритм*, который систематически решает *задачи*, преобразуя *представления*. Он делает это, умело применяя свои базовые инструменты, такие как *управляющие структуры* и *рекурсия*. И хотя задачи можно решать разными способами, алгоритм стремится не предоставить некоторое конкретное решение, а быть как можно более общим, применяя секретное оружие, именуемое *параметрами*. Тем не менее, какие бы конкретно задания он ни выполнял, ему всегда противостоит заклятый враг — *сложность*:

```
Вычисление = Story(алгоритм; решение задач)
```

Данная история разворачивается по мере того как новые, все более крупные задачи доводят алгоритм до предела его возможностей. Но в своей борьбе он пользуется дружеской поддержкой своих родственников из семейства *абстракций*: его сестры *эффективности*, дающей ему дельные советы по поводу благоразумного использования драгоценных ресурсов; его брата *типа*, неустанно защищающего его от программных ошибок и неверных входных данных; а также его мудрого дедушки *языка*, одаряющего его выразительностью и гарантирующего, что он будет понят его верным товарищем *компьютером*, на которого он полагается, осуществляя любые свои планы:

```
Информатика = Story(абстракции; вычисления)
```

Алгоритм не всесилен, не может решить все задачи и к тому же очень уязвим в смысле неэффективности и ошибок. Но он вполне осознает этот факт и, зная свои ограничения, становится сильным и уверенно ожидающим предстоящих приключений.

Дальнейшее исследование

Волшебство в книгах о Гарри Поттере наглядно показывает типы, правила типизации и способы, которыми они помогают предсказывать будущее. Строгую систему магии можно найти в серии фантастических романов *Отшельничий остров* Л.Э. Модезитта-младшего (L.E. Modesitt Jr.), в которых магия представляет собой способность человека управлять хаосом и порядком, присущими всей материи. А главным героем серии книг *Досье Дрездена* Джима Батчера (Jim Butcher) является частный сыщик и маг, расследующий дела, связанные со сверхъестественными явлениями. Детективные истории в этих книгах содержат разные типы, законы и средства магии.

Многие примеры существ с особыми способностями, которые могут быть описаны как типы, можно обнаружить в мире, созданном Дж.Р.Р. Толкиеном (J.R.R. Tolkien) в его романах *Властелин колец* и *Хоббит.* Супергерои из серии комиксов *Люди Икс* и соответствующих сериалов обладают особыми способностями, которые вполне определены и взаимодействуют с естественным миром строго определенным способом. Одни из супергероев определяют тип, состоящий ровно из одного элемента, т.е. из них самих; другие типы состоят из нескольких элементов.

Неправильное пользование устройством нередко вызывает ошибку несоответствия типов и может привести к нарушению его нормальной работы. Обычно это обстоятельство используется в качестве драматического эффекта, как, например, в фильме *Муха*, в котором неправильное использование устройства телпортации приводит к смешению ДНК человека и мухи. Ошибки несоответствия типов возникают и в тех случаях, когда люди сменяют свои роли, играя по стереотипам, связанным с конкретными ролями. Например, в фильме *Чумовая пятница* души юной девицы и ее матери меняются местами своего пребывания в телах, что приводит к поведению, нарушающему все, что можно ожидать от ролей подростка и взрослого[1]. В романе *Принц и нищий* Марка Твена (Mark Twain) своими ролями меняются принц и бедный мальчик.

---

[1] Здесь можно вспомнить такие фильмы, как *Кара небесная* или *Их поменяли телами*, в которых внезапно меняется пол главных героев. — *Примеч. ред.*

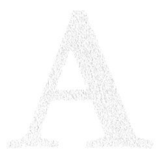
Абстракции принимают самые разные формы. Так, диорамы служат абстракциями, нередко применяемыми для представления разных исторических событий. Некоторые их примеры можно найти в фильме *Ужин с придурками*. Кукла вуду служит абстракцией человека и применяется для наведения порчи на конкретного человека на расстоянии. Примеры тому можно найти в фильмах *Индиана Джонс и Храм Судьбы* и *Пираты Карибского моря: на странных берегах*. В упоминавшейся выше серии книг *Досье Дрездена* куклы вуду также употребляются несколько раз. Аналогично аватар представляет удаленную личность, способную действовать на расстоянии, что составляет основу сюжета фильма *Аватар*. А в мультфильме *Головоломка* в сознании одного человека обитают пять личностей, олицетворяющих основные эмоции.

Деньги служат абстракцией ценности и основным средством для обмена товарами в экономике. Тот факт, что деньги являются социально сложившейся абстракцией и действуют лишь в том случае, если все участники обмена соглашаются признать в них определенную ценность, наглядно показан в историях, в которых применяются нетрадиционные виды валюты, как, например, в фильме *Безумный Макс-2: Воин дороги* (основной валютой здесь является бензин) или *Время* (валютой служит собственный срок жизни). В мини-сериале *Дюна* Франка Герберта (Frank Herbert) валютой служат вода и специи, а в серии юмористических научно-фантастических романов *Автостопом по галактике* английского писателя Дугласа Адамса (Douglas Adams) приводится целый ряд примеров необычных валют.

Абстракции правил поведения можно обнаружить в сборнике коротких научно-фантастических рассказов *Я, робот* Айзека Азимова (Isaac Asimov), в которых они представлены в форме трех законов робототехники, направленных на то, чтобы роботы служили людям, не нанося им вреда. Эти законы представляют собой абстракцию нравственности, а отдельные истории иллюстрируют применение этих законов и их ограничения, временами приводящие к их видоизменению и расширению. В научно-фантастическом романе *Лексикон* Макса Барри (Max Barry) тайное общество выработало новый язык, предназначенный не для передачи мыслей, а для контроля над действиями людей. Если естественные языки, как правило, предоставляют абстракции для описания окружающего мира, то специальный язык в романе *Лексикон* основан на абстракциях нейрохимических реакций, управляющих действиями человека подобно тому, как волшебные чары способны оказывать непосредственное воздействие на поведение человека[2].

---

[2] *Здесь можно также упомянуть фантастическую повесть Вавилон-17* Сэмюэля Дилени (Samuel R. Delany), в которой поднимается ряд языковых вопросов. — *Примеч. пер.*



Словарь специальных терминов

В этом словаре сведены все самые важные термины, употребляемые в данной книге. Статьи этого словаря сгруппированы в разделы по отдельным историям и основным понятиям. Но некоторые статьи присутствуют в нескольких разделах.

Наиболее важные термины, употребляемые в определениях понятий, обозначены буквой того раздела, в котором они определяются. Например, *алгоритм*[А] означает термин *алгоритм* из раздела А.

А. Вычисления и алгоритмы

Алгоритм. Метод решения задачи[А]. Алгоритм применим к разным примерам задач и должен быть задан конечным определением на том языке[Г], который понятен компьютеру[А]. Все шаги метода должны быть эффективны. Выполнение[А] алгоритма над примером конкретной задачи формирует вычисление[А]. Алгоритм должен завершиться[Д] и произвести правильный[А] результат для любой задачи, над которой он выполняется, хотя это происходит не всегда и эти его свойства трудно гарантировать. Алгоритм, понятный машине, называется программой[А].

Время выполнения. Мера, определяющая предполагаемое количество шагов выполнения[А] отдельного алгоритма[А]. Выражается в виде правила, фиксирующего взаимосвязь времени выполнения с объемом входных данных[А]. Обычными мерами времени выполнения являются линейная[В], квадратичная[В], логарифмическая[В] и линейно-логарифмическая[В] (линарифмическая).

Входные данные. Переменная часть задачи[А], которой различаются примеры задач. Входные данные являются частью представления[А] задачи и могут, в свою

очередь, состоять из нескольких частей. Алгоритм[А], применяемый к конкретным входным данным, приводит к вычислению[А] экземпляра задачи для этих входных данных. В алгоритме входные данные представлены параметрами[А].

Выполнение. Процесс последовательного решения конкретного примера задачи[А] по шагам алгоритма[А]. Выполнение осуществляется компьютером[А] и формирует вычисление[А].

Вычисление. Процесс, в ходе которого задача[А] решается в течение ряда шагов путем систематического преобразования соответствующего представления[А] задачи. Вычисление является именно тем, что происходит во время вычисления[А] алгоритма[А] компьютером[А].

Задача. Такое представление[А] ситуации, при котором признается потребность в решении, упрощающем отдельные решения через вычисление[А]. Обозначает как отдельные примеры задач, подверженные преобразованию путем вычисления[А], так и класс всех подобных задач, которые могут служить входными данными[А] для алгоритма[А].

Компьютер. Человек, машина или другой исполнитель, способный выполнить[А] алгоритм[А]. Компьютер должен понимать тот язык[Г], на котором задан алгоритм.

Наихудший случай. Самое продолжительное из возможных время выполнения[А] вычисления[А], формируемого алгоритмом[А]. Наихудший случай служит оценкой, по которой можно судить, является ли алгоритм[А] достаточно эффективным, чтобы произвести решение задачи[А].

Параметр. Имя, употребляемое в алгоритме[А] для обозначения его входных данных[А].

Правильность. Свойство алгоритма[А] всегда производить требуемый результат для любых заданных входных данных[А]. Если алгоритм застревает или не завершается[Д] при обработке каких-нибудь входных данных, то он не является правильным решением задачи[А].

Представление. Сущность, обозначающая нечто в реальном мире. Представление должно быть видоизменяемым шагами вычисления[А]. См. также *Представление*[Б].

Программа. Алгоритм[А], который может быть понят и выполнен[А] машиной.

Понятие	*Проявление в истории из сказки "Гензель и Гретель"*
Алгоритм	Метод следования по камешкам
Время выполнения	Шаги, требующиеся для следования по камешкам домой
Входные данные	Расположение камешков
Выполнение	Выполнение Гензелем и Гретель алгоритма следования по камешкам
Вычисление	Поиск пути домой

Окончание таблицы

Понятие	*Проявление в истории из сказки "Гензель и Гретель"*
Задача	Выживание; отыскание пути домой
Компьютер	Гензель и Гретель
Наихудший случай	Гензелю и Гретель приходится посещать каждый камешек не больше одного раза
Параметр	Выражение "сверкающий камешек, который еще не был посещен"
Правильность	Следование по камешкам, приводящее домой, не заводящее в тупик и не зацикливающееся
Представление	Места расположения камешков
Программа	Описание алгоритма следования по камешкам в распознаваемой машиной форме

Б. Представления и структуры данных

Массив. Структура данных[Б] для представления коллекции элементов. Подобна таблице, состоящей из двух строк, в одной из которых содержатся имена или числа (индексы) элементов, хранящихся в массиве. Индекс служит указателем[Б] для поиска элементов в массиве. Массив удобен для реализации множества[Б], потому что упрощает поиск элементов в течение постоянного времени. Но поскольку в массиве приходится резервировать свободное место для хранения всех возможных элементов, им лучше пользоваться лишь в том случае, если количество элементов, которые могут храниться в массиве, невелико. Массив применяется также для представления блоков в блочной сортировке[В].

Графический знак. Это знак[Б], дающий представление на основании сходства с означаемым[Б].

Дерево. Структура[Б] данных для представления коллекции элементов в виде иерархии. Ее элементы называются узлами[Б]. Любой узел, находящийся выше по иерархии, соединен с нулевым или б*о*льшим количеством узлов, находящихся ниже по иерархии, и хотя бы с одним узлом, находящимся выше по иерархии.

Знак. Особая форма представления[Б], состоящая из означающего[Б] (его видимой части) и означаемого[Б] (т.е. того, что означает означающее).

Ключ. Значение, применяемое для поиска информации в словаре[Б]. См. также *Ключ поиска*[В].

Множество. Тип данных[Б] для представления коллекции элементов, предоставляющий операции ввода, удаления и поиска элементов в коллекции. Может быть представлен словарем[Б], элементами которого являются ключи[Б]. Тип данных множества важен потому, что упрощает представление свойств.

Означаемое. Часть знака[Б], представленная означающим[Б]. Это не конкретный объект из окружающего мира, а представление, которое складывается об объектах в умах людей.

Означающее. Часть знака[Б], которая постигается и обозначает означаемое[Б]. Одно означающее может представлять разные понятия.

Очередь. Тип данных[Б] для представления коллекции, из которой элементы удаляются в том порядке, в каком они в нее поступают. Реализует принцип "первым пришел, первым ушел" доступа к элементам коллекции.

Очередь с приоритетами. Тип данных[Б] для представления коллекции, из которой элементы удаляются по задаваемому приоритетом порядку. Реализует принцип "с наивысшим приоритетом пришел, первым ушел" доступа к элементам коллекции.

Представление. Знак[Б], обозначающий нечто в реальном мире. См. *Представление*[А].

Символ. Знак[Б], обозначающий означаемое[Б] на основании произвольного соглашения.

Словарь. Тип данных[Б] для связывания информации с ключами[Б]. Предоставляет операции для ввода, удаления и обновления информации по отдельным ключам, а также операцию поиска информации по заданному ключу. Множество[Б] является особой формой словаря, в котором элементами являются ключи, по которым никакой информации не хранится.

Список. Структура данных[Б] для представления коллекции элементов в определенном порядке. Элементы доступны по очереди, начиная с того элемента, который находится в начале списка. Список можно рассматривать как дерево[Б], у каждого узла[Б] которого (кроме последнего) имеется ровно один потомок. Может также служить для реализации таких типов данных[Б], как стеки[Б], очереди[Б] или множества[Б].

Стек. Тип данных[Б] для представления коллекции, из которой элементы удаляются в порядке, обратном порядку их поступления. Реализует принцип "последним пришел, первым обслужен" доступа к элементам коллекции.

Структура данных. Представление[А], применяемое в алгоритме[А], предоставляющее отдельный доступ к данным и обеспечивающее манипулирование ими. К наиболее употребительным структурам данных относятся массивы[Б], списки[Б] и деревья[Б].

Тип данных. Описание представления[Б], поведение которого задается рядом операций, способных манипулировать этим представлением[Б]. Может быть реализовано структурой данных[Б]. Один тип данных может быть нередко реализован разными структурами данных, которые, как правило, реализуют операции с разным временем выполнения[А]. Зачастую применяются такие структуры данных, как множества[Б], словари[Б], стеки[Б] и очереди[Б].

Узел. Элемент структуры данных[Б], связанный с другими ее элементами. Примерами узлов служат элементы деревьев[Б] и списков[Б]. В дереве[Б] элементы, доступные непосредственно из узла, называются потомками, а узел, из которого

доступен другой узел, называется родителем. У каждого узла имеется хотя бы один родитель, но у первого сверху, корневого узла дерева родитель отсутствует. А узлы без потомков называются листьями.

Указатель. Знак, дающий представление на основании подобной закону взаимосвязи с означаемым[Б]. Служит также в качестве индекса для поиска элементов в массиве[Б].

Понятие	*Проявление в истории из повести “Собака Баскервилей”*
Массив	Представление множества подозреваемых с отметками
Графический знак	Портрет сэра Хьюго Баскервиля; карта местности Девонширского болота
Дерево	Родословное дерево семейства Баскервилей
Знак	Надпись, выгравированная на тросточке доктора Мортимера
Ключ	Имена подозреваемых
Множество	Круг подозреваемых
Означаемое	Госпиталь на Чаринг-Кросс (обозначается означающим CCH)
Означающее	Сокращение CCH, обозначающее Charing Cross Hospital (Госпиталь на Чаринг-Кросс)
Очередь с приоритетами	Порядок наследования для наследников Баскервиля
Очередь	Список дел, которые доктор Ватсон должен сделать в усадьбе Баскервиль-Холл
Представление	Список или массив подозреваемых; карта местности Девонширского болота
Символ	Номер (2704) наемного кеба
Словарь	Записная книжка Шерлока Холмса с заметками о подозреваемых
Список	Список подозреваемых Мортимер→Джек→Берил→Селден→ ...
Стек	Повторяющееся вычисление потомков детей до братьев/сестер
Структура данных	Список подозреваемых; родословное дерево
Тип данных	Круг подозреваемых
Узел	Имя члена семьи в родословном дереве
Указатель	Следы лап собаки и пепла от сигары на месте преступления

В. Решения задач и их ограничения

Бинарное дерево. Это такое дерево[Б], у каждого узла которого имеется не более двух потомков.

Бинарное дерево поиска. Бинарное дерево[В] с тем свойством, что все узлы в левом поддереве каждого узла[Б] дерева[Б] содержат значение, которое меньше значения в корне поддерева, а все узлы в правом поддереве — больше него.

Бинарный поиск. Алгоритм[А] для поиска элемента в коллекции. Сравнивает искомый элемент с находящимся в коллекции элементом, разделяющим коллекцию

на две области. В зависимости от результата сравнения поиск продолжается только в одной из областей. Элементы выбираются таким образом, чтобы происходило разделение коллекции на две приблизительно одинаковые по размеру области, как и в сбалансированном бинарном дереве поиска[В]. В наихудшем случае[А] время выполнения бинарного поиска оказывается логарифмическим[В].

Блочная сортировка. Алгоритм[А] сортировки, просматривающий сортируемый список[Б] и размещающий каждый элемент в одном из целого ряда специально зарезервированных и первоначально пустых блоков. Как только все элементы списка будут размещены в блоках, эти блоки проверяются по порядку, и все элементы из непустых блоков переносятся в результирующий список. Как правило, для представления блоков при блочной сортировке применяется массив[Б]. При этом требуется, чтобы множество элементов, которые могут оказаться в списке, было не слишком большим, чтобы каждому потенциальному блоку можно было назначить лишь несколько элементов или даже один элемент. В таком случае время выполнения блочной сортировки окажется линейным[В].

Быстрая сортировка. Алгоритм[А] сортировки "разделяй и властвуй"[В], сначала разделяющий сортируемый список[Б] на два подсписка, содержащих все элементы, которые соответственно меньше или больше выбранного опорного элемента. Затем оба эти подсписка сортируются, а полученные результаты соединяются, причем опорный элемент оказывается посредине в результирующем отсортированном списке. Быстрая сортировка в худшем случае[А] выполняется в течение квадратичного времени[В], но на практике проявляет себя очень хорошо и в среднем выполняется в течение линейно-логарифмического времени[В].

Генерация и проверка. Схема алгоритма[А], состоящая из двух основных стадий. На первой стадии получаются потенциальные решения, которые затем систематически проверяются на второй стадии. Порождение и проверка является не алгоритмом, а структурой алгоритма. Примером порождения и проверки служит опробование всех комбинаций, например, для открытия секретного замка или для решения задачи о заполнении рюкзака[В].

Жадный алгоритм. Если алгоритм[А] всегда выбирает наилучший вариант из имеющихся в настоящий момент, то он называется жадным. Примером такого алгоритма служит выбор элементов из отсортированного списка для решения задачи о наполнении рюкзака[В].

Задача о наполнении рюкзака. Это пример трудноразрешимой задачи[В]. Данная задача состоит в том, чтобы наполнить рюкзак ограниченной вместимости как можно б*о*льшим количеством предметов, чтобы достичь максимально возможной стоимости всех предметов в рюкзаке.

Квадратичное время выполнения. Если время выполнения алгоритма[А] возрастает пропорционально квадрату размера входных данных[А], то такое время выполнения называется квадратичным. И хотя квадратичное время выполнения

хуже, чем линейно-логарифмическое[В], оно все-таки намного лучше, чем экспоненциальное[В]. В наихудшем случае[А] сортировка вставками[В] и выбором[В] выполняется в течение квадратичного времени[В].

Ключ поиска. Фрагмент информации, обозначающий то, что требуется найти. В одних случаях ключ является производным, и тогда он оказывается указателем[Б], а в других случаях — отдельным, ни с чем не связанным значением, и тогда он оказывается символом[Б]. Обозначает границу, отделяющую подходящие элементы от неподходящих для текущего поиска. См. также *Ключ*[Б].

Линейное время выполнения. Если время выполнения алгоритма[А] возрастает пропорционально размеру входных данных[А], то оно называется линейным. И хотя линейное время выполнения лучше, чем линейно-логарифмическое время выполнения[В], оно все-таки хуже, чем логарифмическое время выполнения[В]. Поиск наименьшего элемента в списке требует в наихудшем случае[А] линейного времени выполнения.

Линейно-логарифмическое время выполнения. Если время выполнения алгоритма[А] пропорционально размеру входных данных[А], умноженному на логарифм этого размера, то такое время выполнения называется линейно-логарифмическим (линарифмическим). И хотя линейно-логарифмическое время выполнения хуже, чем линейное время выполнения[В], оно все-таки лучше, чем квадратичное время выполнения[В]. В худшем случае[А] сортировка слиянием[В] потребует линейно-логарифмического времени выполнения.

Логарифмическое время выполнения. Если время выполнения алгоритма[А] возрастает лишь на константное значение при увеличении объема входных данных[А] в два раза, то оно называется логарифмическим. Логарифмическое время выполнения намного быстрее, чем линейное время выполнения[В]. Например, бинарный поиск[В] в сбалансированном бинарном дереве поиска[В] выполняется в течение логарифмического времени.

Нижняя граница. Сложность задачи[А]. Указывает скорость роста минимального количества шагов, которое требуется любому алгоритму[А] для решения задачи. Любой алгоритм, время выполнения[В] которого в худшем случае[А] оказывается таким же, как и нижняя граница, считается оптимальным алгоритмом[В].

Оптимальный алгоритм. Это такой алгоритм[А], время выполнения[В] которого в худшем случае[А] оказывается таким же, как и нижняя граница[В] решаемой задачи[А].

Префиксное дерево. Структура данных[Б] дерева[Б] для реализации множеств[Б] или словарей[Б], представляющая каждый ключ[Б] как последовательность узлов[Б]. Префиксное дерево особенно эффективно при сортировке ключей с общими частями.

Приближенный алгоритм. Алгоритм[А] для вычисления[А] решений, которые могут оказаться не совсем правильными[А], но вполне пригодными в большинстве случаев.

“Разделяй и властвуй”. Схема алгоритма[А], в которой решение получается разделением входных данных на отдельные части. Сначала задачи решаются для этих частей независимо друг от друга, а полученные для них решения затем объединяются в единое решение исходной задачи[А]. Само разделение является не алгоритмом, а структурой алгоритма. Примерами разделения служат сортировка слиянием[В] и быстрая сортировка[В].

Сбалансированное дерево. Это такое дерево[Б], листья которого находятся на одинаковом расстоянии от корня (их расстояния различаются не больше чем на 1). Сбалансированное дерево имеет наименьшую высоту среди всех деревьев с одинаковым числом узлов[Б], т.е. распространяется как можно шире. Такая форма дерева гарантирует в худшем случае[А] логарифмическое время выполнения[В] поиска элементов, если бинарное дерево поиска[В] сбалансировано.

Сортировка вставками. Алгоритм[А] сортировки, многократно выбирающий следующий элемент из неотсортированного списка[Б] и вставляющий его в нужном месте отсортированного списка. В худшем случае[А] время выполнения сортировки вставками оказывается квадратичным[В].

Сортировка выбором. Алгоритм[А] сортировки, повторно находящий минимальный элемент в неотсортированном списке[Б] и помещающий его в конец отсортированного списка. Сортировка выбором в худшем случае[А] выполняется в течение квадратичного времени[В].

Сортировка слиянием. Алгоритм[А] сортировки “разделяй и властвуй”[В], сначала разделяющий сортируемый список[Б] на два подсписка равной длины, затем сортирующий эти подсписки и, наконец, осуществляющий слияние отсортированных подсписков в результирующий отсортированный список. Сортировка слиянием в худшем случае[А] выполняется в течение линейно-логарифмического времени[В]. А поскольку нижняя граница[В] для сортировки также линейно-логарифмическая, сортировка слиянием считается оптимальным алгоритмом[В] сортировки.

Трудноразрешимая задача. Это такая задача[А], для решения которой известны только алгоритмы[А] с экспоненциальным временем выполнения[В]. Примерами тому служат задача о наполнении рюкзака[В] и задача коммивояжера.

Экспоненциальное время выполнения. Если время выполнения алгоритма[А] возрастает в некоторое число раз (большее единицы) при увеличении размера входных данных[А] на единицу, то такое время выполнения называется экспоненциальным. Алгоритмы с экспоненциальным временем выполнения не являются практическими решениями задач, поскольку они пригодны для обработки входных данных только небольших размеров.

Понятие	*Проявление в историях об Индиане Джонсе*
Бинарное дерево (поиска)	Представление частотности букв в таких словах, как *Iehova*
Бинарный поиск	Поиск плиток на плиточном полу

Окончание таблицы

Понятие	*Проявление в историях об Индиане Джонсе*
Блочная сортировка	Вычисление частотности букв с помощью массива
Быстрая сортировка	Сортировка задач для поиска Кладезя Душ
Генерация и проверка	Систематическое опробование всех сочетаний весов
Жадный алгоритм	Опробование веса объектов в нисходящем порядке
Задача о наполнении рюкзака	Собирание драгоценных предметов в храме хрустального черепа
Квадратичное время выполнения	Применение сортировки выбором
Ключ поиска	Венеция; Iehova
Линейное время выполнения	Выполнение списка задач
Линейно-логарифмическое время выполнения	Определение веса идола методом проб и ошибок
Логарифмическое время выполнения	Поиск частотности букв в бинарном дереве
Нижняя граница	Минимальное время сортировки списка задач
Оптимальный алгоритм	Оптимальный алгоритм
Префиксное дерево	Плиточный пол
Приближенный алгоритм	Применение надувной лодки в качестве парашюта
"Разделяй и властвуй"	Сужение поисков Генри Джонса-старшего в Венеции
Сбалансированное дерево	Применение некоторых деревьев поиска для определения частотности букв
Сортировка вставками	Сортировка задач для поиска Кладезя Душ
Сортировка выбором	
Сортировка слиянием	
Трудноразрешимая задача	Определение точного веса идола с помощью рычажных весов
Экспоненциальное время выполнения	Определение точного веса идола

Г. Язык и смысловое значение

Абстрактный синтаксис. Иерархическая структура предложения[Г], представленная синтаксическим деревом[Г]. Абстрактный синтаксис языка[Г] определяется грамматикой[Г] как множество всех синтаксических деревьев, которые могут быть построены посредством грамматики.

Вывод. Последовательность сентенциальных форм[Г], каждая из которых получается из предыдущей формы подстановкой нетерминала[Г] по грамматическому правилу[Г]. Первым элементом вывода непременно должен быть нетерминал, а его последним элементом — предложение[Г].

Грамматика. Ряд грамматических правил[Г], преобразующих нетерминалы[Г] в сентенциальные формы[Г]. Определяет язык[Г]. Один язык может определяться

разными грамматиками. Язык, определяемый грамматикой, состоит из всех предложений[Г], для которых существует вывод[Г] из стартового символа[Г].

Грамматическое правило. Правило, имеющее форму *nt* → RHS, где *nt* — нетерминал[Г], а RHS — сентенциальная форма[Г]. Служит для подстановки нетерминалов в сентенциальные формы как часть вывода[Г].

Композиционность. Свойство семантического определения[Г] языка[Г]. Семантическое определение оказывается композиционным в том случае, если смысловое значение предложения[Г] получается из смысловых значений его составных частей. Подразумевает, что структура предложения определяет систематическое объединение смысловых значений его составных частей.

Конкретный синтаксис. Внешний вид предложения[Г] в виде его терминалов[Г] (или расстановки визуальных символов).

Неоднозначность. Свойство грамматики[Г]. Обозначает ситуацию, когда у предложения[Г] может быть два или больше синтаксических деревьев[Г].

Нетерминал. Символ в левой части грамматического правила[Г], вместо которого можно подставить сентенциальную форму[Г]. Стартовый символ[Г] грамматики[Г] является нетерминалом.

Область семантики. Множество значений, пригодных для конкретной области применения. Применяется в семантическом определении[Г] языка[Г].

Определение семантики. Преобразование синтаксических деревьев[Г] языка[Г] в элементы области семантики[Г].

Предложение. Элемент языка[Г], задаваемый последовательностью терминалов[Г].

Сентенциальная форма. Последовательность терминалов[Г] и нетерминалов[Г]. Появляется в правой части грамматического правила[Г].

Синтаксис. Определяет, к какому именно языку[Г] относятся конкретные предложения[Г]. Как правило, определяется грамматикой[Г], состоящей из правил для формирования или реорганизации предложения[Г]. Правила определяют как конкретный синтаксис[Г], так и абстрактный синтаксис[Г] языка.

Синтаксический анализ. Процесс выявления структуры предложения[Г] и представления ее синтаксиса синтаксическим деревом[Г].

Синтаксическое дерево. Иерархическое представление структуры предложения[Г] в форме перевернутого дерева[Б]. Листья дерева являются терминалами[Г], тогда как все остальные узлы — нетерминалами[Г].

Стартовый символ. Выделенный нетерминал[Г] грамматики[Г]. Любое предложение[Г] языка[Г], определяемое грамматикой, должно быть доступно через вывод[Г] из стартового символа.

Терминал. Символ, который появляется только в правой части грамматического правила[Г] и не подлежит подстановке. Предложение[Г] языка[Г] состоит из последовательности одних только терминалов.

Язык. Множество предложений[Г], каждое из которых имеет вид последовательности слов или расстановки визуальных символов (конкретного синтаксиса[Г]) и внутренней структуры (абстрактного синтаксиса[Г] и синтаксического дерева[Г]). Определение языка задается грамматикой[Г]. Язык — это множество предложений, которые можно вывести[Г] из стартового символа[Г] грамматики. У некоторых языков имеется также семантическое определение[Г].

Понятие	*Проявление в мелодии песни “Над радугой”*
Абстрактный синтаксис	Дерево тактов и нот
Алгоритм[А]	Партитура в нотном стане или табулатуре
Вывод	Расширение *мелодии* в партитуру с помощью грамматических правил
Выполнение[А]	Пение или игра
Грамматика	Грамматика мелодий
Грамматическое правило	*нота* →
Композиционность	Звучание песни как соединение звуков в ее тактах
Компьютер[А]	Джуди Гарланд как певица или музыкант
Конкретный синтаксис	Нотный стан; табулатура
Неоднозначность	Основная тема песни без тактовой черты
Нетерминал	*такт*
Область семантики	Звуки
Определение семантики	Отображение синтаксического дерева на звуки
Предложение	Песня, заданная в нотном стане
Сентенциальная форма	*нота* *мелодия*
Синтаксис	Правила печати нотного стана
Синтаксический анализ	Выявление структуры, задаваемой темами, припевами, тактами и т.д.
Синтаксическое дерево	Дерево таковых размеров и нот
Стартовый символ	*мелодия*
Терминал	
Язык	Все допустимые нотные станы

Д. Управляющие структуры и циклы

Завершение. Свойство алгоритма[А] или вычисления[А], которое выполняется по завершении вычисления. Завершение алгоритма[А] нельзя определить автоматически с помощью другого алгоритма; это неразрешимая[Д] задача[А].

Задача останова. Ставит вопрос “Всегда ли алгоритм[А] завершается[Е] для заданных входных данных[А]?”

Неразрешимость. Свойство задачи[А]. Задача считается неразрешимой, если ее нельзя решить с помощью алгоритма[А]. Самой известной из неразрешимых задач является задача останова[Д].

Тело (цикла). Группа операций, повторяющихся в цикле[Д]. Должна модифицировать состояние программы, чтобы цикл мог завершиться[Д].

Управляющая структура. Часть алгоритма[А] для организации порядка, применения и повторения операций. Тремя основными управляющими структурами являются циклы[Д], условные конструкции[Д] и рекурсия[Е].

Условие завершения. Это условие, которое определяет, завершается ли цикл[Д] или продолжает свое выполнение.

Условная конструкция. Управляющая структура[Д] для выбора выполнения[А] одной из двух групп операций.

Цикл. Управляющая структура[Д] для повторения тела[Д] цикла. Количество повторений определяется условием завершения[Д], которое проверяется после каждого выполнения[А] тела цикла и зависит от состояния программы, которое может быть изменено операциями, выполняемыми в теле цикла. Как правило, неясно, сколько повторений цикла будет сформировано, а следовательно, и завершится ли цикл вообще. Двумя основными разновидностями циклов являются циклы `repeat`[Д] и `while`[Д]. В условии их завершения определяется, сколько раз будет повторяться тело цикла, что, как правило, неизвестно перед выполнением цикла. И напротив, количество повторений цикла `for`[Д] фиксируется в начале данного цикла, гарантируя тем самым завершение[Д] подобных циклов[Д].

Цикл `for`. Цикл, тело[Д] которого выполняется фиксированное число раз.

Цикл `repeat`. Цикл[Д], условие завершения[Д] которого всегда проверяется после выполнения операций в теле[Д] цикла.

Цикл `while`. Цикл[Д], условие завершения[Д] которого всегда проверяется перед выполнением операций в теле[Д] цикла.

Понятие	*Проявление в истории из фильма "День Сурка"*
Завершение	Счастливый конец
Задача останова	Завершится ли когда-нибудь повторение Дня Сурка?
Неразрешимость	Невозможность заранее судить, завершится ли День Сурка
Тело (цикла)	События, происходящие в течение Дня Сурка
Управляющая структура	Цикл Дня Сурка
Условие завершения	Является ли Фил Коннорс хорошим человеком?
Условная конструкция	Любое решение, принятое Филом Коннорсом
Цикл	Повторение Дня Сурка
Цикл **`for`**	Количество повторений в фильме, зафиксированное в его сценарии
Цикл **`repeat`**	Цикл Дня Сурка
Цикл **`while`**	

Е. Рекурсия

Базовый случай. Нерекурсивная часть рекурсивного определения[Е]. Как только рекурсивный алгоритм[А] достигнет базового случая, рекурсия[Е] на этом остановится. Базовый случай обязателен, но не достаточен для завершения рекурсии[Е].

Интерпретатор. Компьютер[А], выполняющий[А] алгоритм[А] с помощью типа данных[Б] стека[Б] для отслеживания рекурсивных[Е] (или нерекурсивных) вызовов и многих копий аргументов, возникающих вследствие рекурсивных вызовов[Е].

Косвенная рекурсия. Происходит в том случае, если определение имени или объекта содержит ссылку не на себя, а на другое имя, содержащее ссылку (непосредственно или косвенно) на исходное имя.

Неограниченная рекурсия. Это рекурсия[Е], которая не завершается[Е]. Происходит в том случае, если рекурсивному определению[Е] недостает базового случая[Е] или если базовый случай недоступен для рекурсивных вызовов[Е]. Примерами тому служат непрекращающиеся песенки, бесконечные списки[Б] чисел или некоторые непрерывающиеся вычисления[А].

Ограниченная рекурсия. Это рекурсия[Е], которая завершается[Е].

Описательная рекурсия. Если определение понятия содержит ссылку на себя, это обычно делается с помощью имени или символа. Иногда еще называется рекурсивной ссылкой.

Парадокс. Логическое противоречие, вызванное описательной рекурсией[Е], которую нельзя выполнить, чтобы получить в итоге ограниченную[Е] развернутую рекурсию[Е]. Рекурсивное уравнение описывает парадокс, если оно не имеет фиксированной точки[Е].

Подстановка. Процесс замены параметров[А] конкретными значениями. При этом вызов алгоритма[А] заменяется его определением, тогда как аргументы подставляются вместо параметров.

Прямая рекурсия. Происходит в том случае, если имя или объект содержит ссылку на себя.

Развернутая рекурсия. Происходит в том случае, если какой-нибудь артефакт (как правило, изображение или текст) содержит свою (иногда масштабированную) копию. Иногда называется также самоподобием. Фракталы и геометрические рисунки наподобие треугольников Серпинского служат примерами развернутой рекурсии в изображениях, а матрешки — примером рекурсивного физического объекта.

Рекурсивное определение. Определение, в котором имя или объект содержит ссылку на себя. В рекурсивном определении алгоритма[А] ссылка называется рекурсивным вызовом[Е].

Рекурсивный вызов. Вызов алгоритма[A] из его собственного определения (прямая рекурсия[E]) или из определения другого, вызываемого им алгоритма (косвенная рекурсия[E]).

Рекурсия. Означает описательную рекурсию[E], развернутую рекурсию[E] или выполнение[A] рекурсивного определения[E].

След. Последовательность, фиксирующая разные состояния, через которые проходило вычисление[A]. Удобен для иллюстрации вычисления[A] алгоритма[A], задаваемого рекурсивным определением[E].

Фиксированная точка. Решение рекурсивного уравнения.

Понятие	*Проявление в истории из кинотрилогии "Назад в будущее"*
Базовый случай	Марти спасает доктора Брауна в 1885 году
Интерпретатор	Вселенная, выполняющая действия по путешествию во времени
Косвенная рекурсия	Разделение алгоритма `Goal` на две функции, например `Easy()` и `Hard()`
Неограниченная рекурсия	Происходит в том случае, если Марти никогда не прекращает путешествия обратно во времени
Ограниченная рекурсия	Путешествие во времени происходит лишь фиксированное число раз
Описательная рекурсия	Описание путешествия во времени в алгоритме `ToDo` или `Goal`
Парадокс	Дженнифер встречается себя старую/молодую
Подстановка	Последовательность действий, полученная для вызова алгоритма `ToDo` или `Goal`
Прямая рекурсия	Марти путешествует во времени в 1885 год после путешествия во времени в 1955 год
Развернутая рекурсия	Последовательность действий, происходящих в результате путешествия во времени
Рекурсивное определение	Определение алгоритма `ToDo` или `Goal`
Рекурсивный вызов	Вызов `ToDo(1885)` в определении `ToDo(1955)`
Рекурсия	Марти путешествует во времени в прошлое
След	Линейная последовательность действий из кинотрилогии
Фиксированная точка	Согласованные действия, предпринимаемые Марти и другими

Ж. Типы и абстракция

Абстрактная машина. Высокоуровневое описание машины, нередко задаваемое в виде математической модели, в котором игнорируются подробности реализации конкретных машин, а следовательно, дается единообразное представление

о целом ряде различных, но сходных машин. Примером тому служит машина Тьюринга[Ж].

Абстракция данных. Абстракция[Ж], применяемая к представлению[А,Б]. Дает общее представление о самых разных данных. Основной формой абстракции данных служат типы[Ж].

Абстракция. Процесс пренебрежения подробностями при описании и сведении, а также результат данного процесса. Чтобы упростить применение абстракции, для нее задается интерфейс[Ж].

Аксиома. Правило, которое всегда истинно. Правило типизации[Ж] является аксиомой, если у него отсутствуют предпосылки[Ж].

Динамический контроль типов. Контроль типов[Ж] во время выполнения[А] алгоритма[А].

Заключение. Часть правила типизации[Ж]. Заключение из правила типизации[Ж] истинно только в том случае, если истинны все его предпосылки[Ж].

Интерфейс. Имя плюс параметры[А], присваиваемые абстракции[Ж], чтобы упростить ее применение.

Лямбда-исчисление. Абстракция[Ж] языка[Г] для алгоритмов[А]. Если тезис Чёрча-Тьюринга истинен, любой алгоритм может быть выражен в виде программы[А] лямбда-исчисления. В своей выразительности для описания алгоритмов и вычисления[А] лямбда-исчисление равнозначно машине Тьюринга[Ж].

Машина Тьюринга. Абстрактная машина[Ж] для компьютеров[А]. Служит в качестве математической модели вычисления[А]. В этом отношении равнозначна лямбда-исчислению[Ж]. Если тезис Чёрча-Тьюринга истинен, любой алгоритм[А] может быть транслирован в программу[А] для машины Тьюринга.

Правило типизации. Правило наложения ограничений для допустимых входных данных[А] и выходных данных алгоритмов[А], позволяющее обнаруживать ошибки в алгоритмах. Может быть проверено автоматически средством контроля типов[Ж].

Предпосылка. Часть правила типизации[Ж], которое должно быть истинным, чтобы вступило в действие заключение[Ж] из правила типизации.

Средство контроля типов. Алгоритм[А], проверяющий, не нарушает ли другой алгоритм любые правила типизации[Ж].

Статический контроль типов. Контроль типов[Ж] перед выполнением[А] алгоритма[А].

Тезис Чёрча-Тьюринга. Гласит, что любой алгоритм[А] можно выразить в виде программы для машины Тьюринга[Ж] или лямбда-исчисления[Ж]. Считается истинным большинством специалистов по информатике.

Тип. Абстракция данных[Ж], предоставляющая единообразное описание группы представлений[А].

Функциональная абстракция. Абстракция[Ж], применяемая к вычислению[А]. Дает общее представление о самых разных вычислениях. Воплощается в понятии алгоритма[А], описывающем класс сходных вычислений.

Экземпляр. Экземпляр абстракции[Ж], получаемый подстановкой[Е] аргументов вместо параметров[А] в определении абстракции.

Понятие	*Проявление в книгах о Гарри Поттере*
Абстрактная машина	—
Абстракция данных	Типы магов и магглов
Абстракция	Зелье, карта Мародера
Аксиома	Магглы не способны творить волшебство
Динамический контроль типов	Попытка налагать чары без проверки требований
Заключение	...то он или она может налагать чары (см. *Предпосылка*)
Интерфейс	У заклинания имеется название и для него требуются волшебная палочка и магическая формула
Лямбда-исчисление	—
Машина Тьюринга	
Правило типизации	Если кто-нибудь является магом, он может налагать чары
Предпосылка	Если кто-нибудь является магом, то... (см. *Заключение*)
Средство контроля типов	Мир волшебства
Статический контроль типов	Предсказание, что магглы не способны творить чудеса
Тезис Чёрча-Тьюринга	—
Тип	Заклинание, маг, маггл
Функциональная абстракция	Чары
Экземпляр	Применение заклинания "Вингардиум Левиоза" к перышку

Примечания

Введение

1. John Dewey, "The Pattern of Inquiry," in *Logic: Theory of Inquiry* (1938).

Глава 1, "Путь к пониманию вычислений"

1. Цитаты взяты из версии издания *Grimms' Fairy Tales* (Сказки братьев Гримм) Якоба и Вильгельма Гримм, свободно доступной в Интернете по адресу `www.gutenberg.org/ebooks/2591`.
2. Представленная здесь терминология может показаться несколько запутанной, поскольку термином *задача* обычно обозначается как конкретная задача, так и класс задач. Например, мы говорим как об общей задаче "поиска пути", так и о конкретной задаче "поиска пути между двумя отдельными местами". Но, как правило, подобная неоднозначность разрешается по контексту.
3. Это справедливо только при определенных обстоятельствах, о которых речь пойдет далее.

Глава 2, "От слов к делу: когда действительно происходит вычисление"

1. Количество молотого кофе определяется в соответствии с рекомендациями Национальной кофейной ассоциации (National Coffee Association; см. по адресу `https://www.ncausa.org/About-Coffee/How-to-Brew-Coffee`).

2. Употребление термина *компьютер* было впервые зафиксировано еще в 1613 году. Идея механического компьютера впервые была реализована в виде разностной машины Чарльзом Беббиджем (Charles Babbage) в 1822 году. А первый программируемый компьютер в виде электромеханического вычислительного устройства Z1 был построен около 1938 года Конрадом Цузе (Konrad Zuse).
3. Здесь предполагается, что Гензель и Гретель никак не сокращают путь и не отступают назад, чтобы выйти из тупика (см. главу 1, "Путь к пониманию вычислений").

По дороге

1. Если вы никогда не водили автомашину в Соединенных Штатах, Канаде или Южной Африке, то вам, вероятно, неизвестно это правило дорожного движения. Для тех, кто учился вождению в Германии, перекрестки четырех улиц просто непостижимы своей организацией и, в частности, служат причиной огромного количества аварий и споров по поводу того, кто приехал на перекресток первым.
2. Это следует из того, что знаки являются представлениями, а вычисление было ранее определено как систематическое преобразование представлений.

Глава 3, "Тайна знаков"

1. Двоичное число состоит из последовательного ряда единиц и нулей. Первые числа натурального ряда и нуль представлены в двоичной форме следующим образом: 0 → 0, 1 → 1, 2 → 10, 3 → 11, 4 → 100, 5 → 101, 6 → 110, 7 → 111, 8 → 1000, 9 → 1001 и т.д.
2. Если это покажется вам сложным для понимания, примите во внимание, что эта глава была написана в начале 2015 года, когда приближался "Супербоул XLIX" (финал первенства национальной лиги американского футбола в сезоне 2014 года). Римское число XLIX соответствует арабскому числу 49, где цифра L означает 50, а предшествующая ей цифра X — вычитание 10. К полученному результату (40) следует прибавить число IX (т.е. 9), что означает вычитание 1 из 10.
3. Чтобы удвоить число в двоичной системе, как и умножить число в десятичной системе на 10, достаточно добавить 0 справа.
4. В общем случае операция пересечения может дать несколько точек (или вообще ни одной точки), если дорога пересекает реку в нескольких местах (или вообще не пересекает ее).
5. В повести *Собака Баскервилей* отсутствуют примеры вычислений с помощью отдельных символов. Но в то же время в ней присутствуют примеры вычислений с помощью коллекций символов (см. главу 4, "Записная книжка сыщика").

Глава 4, "Записная книжка сыщика"

1. Цитаты взяты из версии повести *The Hound of the Baskervilles* (Собака Баскервилей) Артура Конана Дойла, свободно доступной в Интернете по адресу `www.gutenberg.org/files/2852/2852-h/2852-h.htm`.
2. Имеются и более сложные варианты списков, например двусвязные списки, элементы которых связаны в обоих направлениях. Но в этой главе рассматривается только простой односвязный список.
3. В принципе, порядок следования элементов мог бы иметь значение, а их положение в списке — использовано, например, для выражения степени подозрения каждого участника событий. Но в данной истории ничто не указывает на то, что Шерлок Холмс придерживается такого порядка.
4. Следует, однако, иметь в виду, что данное предположение обычно *не* оправдывается, поскольку имена, как правило, нельзя употреблять в качестве идентификаторов элементов массива, чтобы обеспечить эффективный доступ к ним. Впрочем, это ограничение можно отчасти преодолеть, употребив более развитые структуры данных вроде хеш-таблиц или так называемых деревьев, обсуждаемых в главе 5, "Поиск идеальной структуры данных". Этим ограничением можно в дальнейшем пренебречь, поскольку оно никак не влияет на сравнение списков с массивами.
5. См. `wikipedia.org`.
6. Вики (wiki) — программа для коллективного редактирования и обмена информацией через Интернет.
7. Их конечной целью, конечно, было сократить множество до одного конкретного элемента.

Потери и находки

1. Для поиска информации в Интернете можно найти немало инструкций на тот случай, если вы забудете, как это делается, хотя сам поиск подходящих инструкций может сам по себе оказаться затруднительным.

Глава 5, "Поиск идеальной структуры данных"

1. Текущая статистика посещения веб-сайта YouTube приведена по адресу `https://www.youtube.com/yt/about/press/`.
2. Наличие элемента внутри области поиска не гарантируется, поскольку искомый элемент может вообще отсутствовать в этой области.
3. См. `en.wikipedia.org/wiki/Boggle`.
4. Здесь предполагается, что становиться на плитки в определенном порядке и обходить все надежные плитки совсем необязательно.

5. Таким образом, чтобы подсчитать каждую букву только один раз при обработке слова, придется организовать тип данных множества. Такое множество можно проверить на отсутствие буквы, прежде чем обновлять его счетчик, а впоследствии ввести букву в множество, чтобы исключить дальнейшее обновление счетчика, если эта буква еще не раз встретится в том же самом слове.
6. В действительности это отнимет время, пропорциональное количеству элементов, умноженному на логарифм данного количества. Такая временн*ая* сложность еще называется *линейно-логарифмической* (линарифмической — linearithmic).
7. По-английски этот вид деревьев обозначается словом *trie*, которое произносится как "три" в слове re*trie*val (извлечение), откуда оно и происходит, но сегодня многие произносят его как "трай" (try), чтобы подчеркнуть особые свойства этого вида деревьев, которыми они, в общем, отличаются от бинарных деревьев поиска.
8. Половина узлов бинарного дерева поиска содержится в его листьях.

Приведение дел в порядок

1. Время выполнения алгоритма имеет линейно-логарифмический характер, если для этого требуется время, пропорциональное размеру входных данных (в данном случае — количеству учащихся и экзаменов), умноженному на логарифм этого размера.
2. Даже если принять во внимание, что список экзаменов сокращается на каждом шаге.

Глава 6, "Сортировка алгоритмов сортировки"

1. Точнее, $1 + 2 + 3 + \quad + n = \frac{1}{2}n(n+1)$.
2. Поскольку мы удалили опорный элемент.
3. Относится к логарифмам по основанию 2. $log_2 100 < 7$, поскольку $2^7 = 128$, и т.д.
4. Джон фон Нейман (John von Neumann) более всего известен в информатике описанием архитектуры вычислительных машин, на основании которого построены все современные компьютеры.
5. Дело в том, что между любыми двумя именами, которые могут служить в качестве индексов соседних элементов массива, существует бесконечно много других имен. Допустим, что после индекса "aaa" должен следовать индекс "aab". В таком случае комбинации "aaaa," "aaab," "aaaaa" и бесконечное число других индексов не могут служить индексами массива и, соответственно, не могут быть подсчитаны. По той же самой причине массивы нельзя индексировать действительными числами, а следовательно, сортировка подсчетом не годится для сортировки списков действительных чисел.

6. Или по крайней мере один из Священных Граалей, поскольку оптимальных алгоритмов может быть несколько.
7. Все возможные (и содержащие разные элементы) списки длиной n могут быть составлены следующим образом. Любой из n элементов может оказаться на первой позиции в списке. И тогда любой из оставшихся $n-1$ элементов может оказаться на второй позиции в списке и т.д. Общее число возможных вариантов составляет $n \times (n-1) \times \ldots \times 2 \times 1 = n!$, а это означает, что имеется $n!$ разных списков длиной n.
8. Алгоритм, в котором используется не больше k операций сравнения, способен различить 2^k разных случаев. Следовательно, чтобы справиться со всеми возможными вариантами для списка длиной n, необходимо, чтобы выполнялось соотношение $2^k \geq n!$ или $k \geq \log_2(n!)$. Можно показать, что $k \geq n \log_2 n$, что и дает линейно-логарифмическую (линарифмическую) нижнюю границу.

Глава 7, "Трудноразрешимые задачи"

1. Включая и пустое сочетание, которое является решением в том случае, если вес должен быть нулевым. В данном примере этот случай не имеет особого значения, но в общем он вполне возможен. Исключение пустого множества из рассмотрения не меняет тот факт, что число возможных сочетаний изменяется экспоненциально количеству имеющихся объектов.
2. На самом деле действие закона Мура, отметившего свой 50-летний юбилей в 2015 году, подходит к концу, поскольку минимизация электронных схем достигла принципиальных пределов, налагаемых законами физики.
3. Квадратичный алгоритм для обработки входных данных объемом n выполняется за n^2 шагов. Удвоение быстродействия компьютера означает, что он способен выполнить за то же самое время в два раза больше шагов, т.е. $2n^2$ шагов. Это количество шагов, выполняемых алгоритмом для обработки входных данных объемом $\sqrt{2}n$, поскольку $\left(\sqrt{2}n\right)^2 = 2n^2$. Иными словами, алгоритм способен обработать входные данные, более крупные приблизительно в 1,4 раза, поскольку $\sqrt{2} \approx 1,4142$.
4. Чтобы получить энергию, необходимую для свечения лампочки накаливания мощностью 1 Вт в течение 1 с, потребуется расщепить около 30 млрд. атомов.
5. Здесь P означает *полиномиальный*, т.е. класс задач, для которых решение может быть построено по алгоритму со временем выполнения, пропорциональным полиному, например, n^2 или n^3, N — *недетерминированный*. Таким образом, сокращение NP означает класс задач, для которых решение может быть построено по алгоритму с полиномиальным временем выполнения на недетерминированной машине, т.е. гипотетической машине, на которой всегда делается правильное предположение для каждого решения, которое требуется принять в алгоритме. Например, решение задачи взвешивания можно сформировать в

течение линейного времени, если удастся сделать правильное предположение относительно включения каждого веса в данное решение. В равной степени сокращение *NP* означает класс задач, для которых решение может быть проверено с помощью алгоритма с полиномиальным временем выполнения. Например, предлагаемое решение задачи взвешивания может быть проверено в течение линейного времени, поскольку для этого требуется лишь сложить все исходные веса и сравнить их с целевым весом.

6. Но если приходится неоднократно искать и удалять наименьший элемент из списка, то основанный на сортировке алгоритм становится в какой-то момент более эффективным.
7. Рассмотрим первый добавленный объект. Он весит больше половины целевого веса, что доказывает, что такое возможно, или меньше половины, что означает, что можно добавить следующий объект, поскольку сложенный вес, который меньше первого веса, а следовательно, меньше половины целевого веса, не достигает целевого веса.

   Теперь мы можем различить оба случая. Если оба объекта вместе весят больше половины целевого веса, их вес составляет меньше 50% целевого веса. А если они весят меньше половины целевого веса, то каждый из них должен весить меньше одной четвертой целевого веса, как, впрочем, и следующий объект, поскольку он весит еще меньше, а следовательно, может быть добавлен. Следуя этому рассуждению, нетрудно показать, что объекты можно добавлять хотя бы до тех пор, пока не будет достигнута половина целевого веса. Такой аргумент допускает наличие достаточного количества объектов для достижения целевого веса, но подобное допущение уже содержится в допущении, что оптимальное решение может быть найдено.

Предписания врача

1. Ludwig Wittgenstein, *Tractatus Logico-Philosophicus* (1921), para 5.6, trans. K. C. Ogden. `www.gutenberg.org/files/5740/5740-pdf.pdf`.

Глава 8, “Сквозь призму языка”

1. Американская ассоциация звукозаписи (Recording Industry Association of America) и Национальный фонд поддержки искусств (National Endowment for the Arts) поставили эту мелодию первой в своем списке “Песен века” (Songs of the Century).
2. Музыка Харольда Арлена (Harold Arlen), слова Эдгара Харбурга (E.Y. Harburg).
3. Цель игры в испорченный телефон — передать шепотом сообщение или фразу от одного человека к другому. Если в этой игре принимает участие большая группа людей, то после целого ряда передач фраза зачастую искажается самыми забавными способами.

4. К сожалению, табулатура неоднозначна в отношении длительности нот, а следовательно, может быть, по существу, использована лишь в том случае, если исполнитель уже знает музыкальное произведение.
5. Чтобы ясно отличать нетерминальные символы от терминальных, первым всегда присваивается имя в пунктирной рамке подобно пунктирному подчеркиванию, употреблявшемуся в главе 2, “От слов к делу: когда действительно происходит вычисление”. Такое обозначение, в свою очередь, вызывает обозначение подставляемого заполнителя.
6. Для представления n разных высот тона и m разных длительностей звучания нот потребуется $n \times m$ правил. Такое количество правил можно существенно сократить приблизительно до $n+m$ правил, разложив каждую ноту на два нетерминала для обозначения высоты тона и длительности звучания, плюс правила для получения этих двух свойств ноты независимо одно от другого.
7. Идиоматические выражения являются исключением из данного правила (см. главу 9, “Поиск нужного тона: смысл звука”).
8. Ох, я забыл завершить вопрос словами “если таковое имеет место”, что могло помешать вам успешно проанализировать данное предложение. В таком случае вы могли бы исправить предложение, добавив недостающую часть, но те предложения, для которых такой прием не подходит, так и останутся лишенными смысла, вызывая у читателя или слушателя весьма неприятное ощущение досады.

Глава 9, “Поиск нужного тона: смысл звука”

1. Точнее говоря, следовало бы сказать, что каждое предложение имеет *хотя бы* одно смысловое значение, поскольку некоторым синтаксически правильным предложениям может недоставать смыслового значения. Например, каково смысловое значение предложения “Зеленая мысль съела”?

Глава 10, “Намылить, смыть, повторить”

1. Город в Индии, удерживающий мировой рекорд по самому высокому уровню месячных и годовых осадков в виде дождя.
2. Напомним, что *шаг* и *условие* являются нетерминальными символами, обозначающими действия и условия. Если заменить эти нетерминалы действием и условием соответственно, данная схема станет примером составления цикла для конкретной программы.
3. Это высказывание часто приписывается то Альберту Эйнштейну (Albert Einstein), то Бенджамину Франклину (Benjamin Franklin), но его авторство неясно. Первые письменные ссылки на него можно найти в романе *Sudden Death* (Внезапная смерть) Риты Мэй Браун (Rita Mae Brown), увидевшем свет

в 1983 году, а также в брошюре Общества анонимных алкоголиков (Alcoholics Anonymous), изданной в 1980 году.

4. С другой стороны, можно расширить первый *шаг* до *встать*, а второй *шаг* — до *принять душ*; *позавтракать*, что приведет в конечном счете к тому же самому результату.
5. Из этого правила имеется ряд важных исключений. Например, в большинстве электронных таблиц *не* применяют цикл для того, чтобы применить, скажем, формулу к ряду значений или ячеек таблицы. Для этого, как правило, строки и столбцы электронных таблиц копируются, что очень похоже на развертывание цикла **repeat** *сложить три раза* в последовательность *сложить*; *сложить*; *сложить*.
6. Условие в цикле с повторением уместно называется *условием завершения*, тогда как условие в цикле с проверкой условия следовало бы назвать *предусловием* или *начальным условием*.
7. Циклы с фиксированным числом итераций можно обнаружить в большинстве языков программирования, где они имеют приведенную ниже форму, которая является несколько более общей. В этой форме тело цикла снабжается сведениями о количестве пройденных итераций.

   for *имя* := *число* **to** *число* **do** *шаг*

   В этой форме вводится нетерминал *имя*, действующий в качестве счетчика, привязанного к текущему числу итераций. Оно может использоваться и в теле цикла, как, например

   for *n* := 1 **to** 10 **do** *вычислить квадрат числа n*

И не останавливаться

1. Рекурсия может также стать источником незавершенности алгоритма. Но поскольку любой рекурсивный алгоритм может быть преобразован в нерекурсивный алгоритм, в котором применяются циклы, то достаточно рассмотреть только циклы.

Глава 11, “Счастливый конец не гарантируется”

1. Самая важная часть алгоритма на этом рисунке, конечно, опущена. При этом лишь предполагается, что условие для проверки завершения алгоритма может быть каким-то образом определено, чего фактически нельзя сделать, как будет показано далее.
2. См. `en.wikipedia.org/wiki/Barber_paradox`.
3. Если задача требует утвердительного или отрицательного ответа, как в случае с задачей остановки, то она называется *задачей принятия решения*. Задача принятия решения, не имеющая алгоритмического решения, называется

неразрешимой, а иначе — *разрешимой*. Если же задача требует получения конкретного выходного значения для любого заданного входного значения, она называется *функциональной*. Функциональная задача, не имеющая алгоритмического решения, называется *невычислимой*, а иначе — *вычислимой*.

4. Если вам известно, чем понятие "счетного множества" отличается от понятия "несчетного множества", то множество разрешимых задач счетно, тогда как количество множеств неразрешимых задач несчетно.

Глава 12, "Своевременный стежок вычисляется впрок"

1. Между прочим, Карл Саган (Carl Sagan) считал второй фильм *Назад в будущее* едва ли не самой лучшей экранизацией научных воззрений о путешествиях во времени (как следует из комментариев в вопросах и ответах авторов фильма: режиссера Роберта Земекиса (Robert Zemeckis) и сценариста Боба Гейла (Bob Gale), Q&A Commentary with Robert Zemeckis and Bob Gale," *Back to the Future Part II*, Blu-Ray (2010)).
2. Напомним, что точкой с запятой обозначается управляющая структура, соединяющая шаги алгоритма в последовательность.
3. И хотя дело обстоит несколько сложнее, аналогия все же верна.
4. Рекурсивное описание данного алгоритма оказывается более длинным потому, что название алгоритма приходится указывать явно.
5. Различают также *порождающую* и *структурную* рекурсию, но такое различие не столь важно для глубокого понимая самого понятия рекурсии. Кроме того, различают *линейную* и *нелинейную* рекурсию (см. главу 13, "Все дело в интерпретации").
6. См. `en.wikipedia.org/wiki/Drawing_Hands` и `en.wikipedia.org/wiki/Print_Gallery_(M._C._Escher)`.
7. Только в том случае, если эти алгоритмы применяются к неотрицательным числам.
8. Взято из книги *Essentials of Discrete Mathematics* (Основы дискретной математики) Дэвида Хантера (David Hunter), Jones & Bartlett Publishers, 2011.

Состояние дел

1. Следует, однако, иметь в виду, что это не просто совокупность результатов, определяющих алгоритм, поскольку разные алгоритмы для решения одной и той же задачи различаются тем, как они ее решают.

Глава 13, "Все дело в интерпретации"

1. Такой способ передачи аргументов называется *вызовом по значению*.

2. В данном примере значения параметров фактически не требуются, и вычисление по алгоритму `Isort` на этом завершается.
3. Возможно, это происходит потому, что молодой Бифф не осознает, что старый Бифф, который передает ему спортивный альманах, является им же самим, но в старшем возрасте.
4. На практике можно остановить рекурсию, если списки имеют единичную длину. Они уже отсортированы и поэтому могут быть возвращены без изменения.

Глава 14, "Волшебный тип"

1. Если в будильнике используется 12-часовый формат времени, в нем потребуется дополнительно указать время до и после полудня (a.m./p.m соответственно).
2. Типы, которые могут быть описаны подобным образом, называются *полиморфными*, потому что могут принимать самые разные формы. А поскольку все эти разнообразные формы могут быть описаны с помощью одного параметра, то такого рода полиморфизм называется *параметрическим*.
3. Но возможности правил типизации все же ограничены. Они не могут, например, гарантировать завершение алгоритмов.
4. Все это зависит от контекста. Как правило, можно также сказать, что не имеет никакого смысла умножать рост человека на самого себя, хотя квадрат роста человека действительно используется для вычисления так называемого росто-весового показателя, или индекса массы тела, путем деления на его вес.

Глава 15, "С высоты птичьего полета: от абстракции к деталям"

1. Слово *абстракция* происходит от латинского глагола *abstrahere*, означающего "оттаскивать, отвлекать".
2. Данное определение можно было бы дополнительно расширить до списков произвольной длины со многими главными героями. Но такое определение оказалось бы несколько более сложным.
3. Ради краткости правила для расширения нетерминалов *protagonist* и *challenge* были опущены.
4. Конусообразное представление абстракции само является абстракцией, в которой понятие абстракции находится вверху, тогда как абстрагируемые понятия — внизу.
5. Здесь не учитывается то обстоятельство, что бегун в конечном счете устанет, что накладывает естественное ограничение на дальность забега.
6. В данном контексте слово *алгоритм* означает в более узком смысле методы вычисления математических функций, куда не входят, например, рецепты.

Предметный указатель

А

Абстрактная машина, 315; 336
Абстракция, 277; 297; 300; 301
  данных, 310; 337
  функциональная, 309; 338
Азбука Морзе, 197
Аксиома, 286; 337
Алгоритм, 13; 18; 31; 40; 323
  выполнение, 324
  жадный, 157; 328
  завершаемость, 44; 333
  линейный, 84
  недетерминированный, 190
  оптимальный, 140; 329
  приближенный, 146; 148; 329
  примеры, 16
  сложность, 52
  требования, 42
Анаграмма, 161
Анализ
  восходящий, 183
  нисходящий, 183
  синтаксический, 182; 332

Б

Блок-схема, 209

В

Википедия, 90
Входные данные, 55; 155; 323
Вычисление, 12; 32; 33; 324
  отложенное, 143
  планирование, 143
  предварительное, 125; 142
Вычислитель, 50

Г

Гистограмма, 111
Грамматика, 174; 331
Граница
  нижняя, 127; 140

Д

Данные
  входные, 55; 155; 323
Дерево, 93; 325
  абстрактное синтаксическое, 180
  бинарное, 112; 327
    поиска, 112; 327
  генеалогическое, 95; 110
  обход, 96
  поддерево, 112
  префиксное, 118; 329
  сбалансированное, 116; 330
  синтаксического анализа, 180
  синтаксическое, 332

Ж

Жест, 198

З

Завершаемость алгоритма, 44; 333
Задача, 31; 324
  NP-полная, 154
  вычислимая, 347
  невычислимая, 228
  неразрешимая, 228; 334
  останова, 333
  приведение, 155
  принятия решения, 346
  сложность, 140
  трудноразрешимая, 146; 147; 330
Закон
  Мура, 151
Знак, 65
  графический, 72

И

Индекс, 73
Интерпретатор, 196; 259; 335
Интерпретация, 64
Интерфейс, 301; 337
Информатика, 11; 18
Итерация, 204

К

Квадратичная зависимость, 56
Классификация, 281
Ключ, 89; 104; 325
Композиционность, 188; 194; 332
Компьютер, 11; 50; 52; 324
  термин, 340
Конечность, 42

Л

Лабанотация, 198
Линейная зависимость, 55
Логарифмическая зависимость, 116
Лямбда-исчисление, 318; 337

М

Массив, 78; 325
  индекс, 86
  элемент, 86
Машина
  абстрактная, 315; 336
  Тьюринга, 316; 337
Метод
  генерации с проверкой, 150; 328
  разделяй и властвуй, 138; 330
Множество, 85; 325
Модель, 170

Н

Наихудший случай, 54; 324
Неоднозначность, 187; 190; 332
  грамматическая, 188
  лексическая, 188
Нотный стан, 171

О
Обход дерева, 96
Омоним, 70; 188
Оптимальное решение, 127
Очередь, 77; 92; 326
  с приоритетами, 91
Ошибка логическая, 190

П
Парадокс, 335
Параметр, 45; 298; 324
Партитура, 172
Переменная, 205; 221
Пиктограмма, 72
Повторное использование, 132
Поддерево, 112
Подстановка, 259; 335
Поиск бинарный, 102; 327
Предварительное вычисление, 125; 142
Предвычисление. См. Предварительное вычисление
Предложение, 173; 332
Представление, 36; 63; 324; 326
Проблема останова, 219
  неразрешимость, 224
Программа, 51

Р
Рекурсия, 22; 94; 133; 136; 235. См. Рекурсия
  базовый случай, 335
  косвенная, 335
  линейная, 270
  неограниченная, 335
  ограниченная, 335
  описательная, 335
  прямая, 335

С
Семантика, 185; 192
  денотационная, 193
Семиотика, 67
Сентенциальная форма, 176; 332
Символ, 65; 73
Синтаксис, 170; 173; 332
  абстрактный, 180; 331
  конкретный, 180; 332
След, 73; 336
Словарь, 89; 326
Сложность
  алгоритма, 52
  временная, 53
  пространственная, 52
Смысловое значение, 63; 192
Сортировка, 123; 127
  блочная, 137; 328
  быстрая, 133; 328
  вставками, 130; 330
  выбором, 129; 330
  и поиск, 128
  подсчетом, 139
  слиянием, 136; 330
Список, 78; 82; 326
Стек, 78; 81; 92; 98; 326
Структура
  данных, 78; 82; 326
  управляющая, 201; 203; 208; 334
Структурная распечатка, 183

Т
Табулатура, 173
Тезис Чёрча-Тьюринга, 318; 337
Терминал, 175; 332
Тип, 23; 170; 277; 281; 337
  данных, 77; 81; 85; 281; 326
  контроль, 291
    динамический, 291
    статический, 291
  ошибка несоответствия, 278
Трансляция, 317
Требования к алгоритму, 42
Указатель, 49; 73; 327
Управляющая структура, 201; 203; 208; 334

У
Условная конструкция, 209

Ф
Фиксированная точка, 336

Ц
Цикл, 201; 203; 204; 334
  бесконечный, 207
  конечный, 207
  развертывание, 222
  с повторением, 212
  с проверкой условия, 212
  с фиксированным числом итераций, 214
  условие завершения, 206; 334

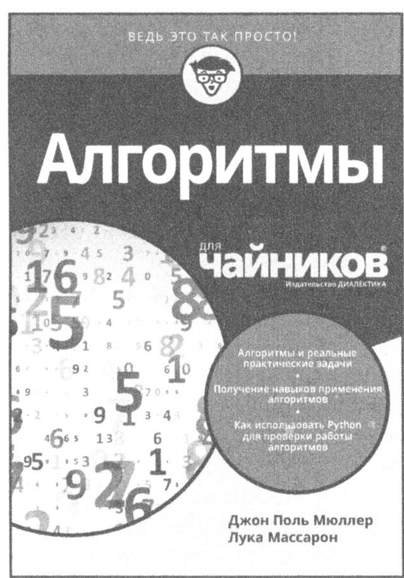
Ш
Шифр, 75; 159

Э
Элемент
  массива, 86
  опорный, 132
Эффективность, 42; 58

Я
Язык, 19; 167; 333
  визуальный, 210
  программирования, 13; 51; 317
  синтаксис, 173
  трансляция, 315

АЛГОРИТМЫ ДЛЯ ЧАЙНИКОВ

Джон Поль Мюллер
Лука Массарон



www.dialektika.com

Эта книга — действительно книга для чайников, поскольку основная ее задача не научить программировать реализации тех или иных давно известных алгоритмов, а познакомить вас с тем, что же такое алгоритмы, как они влияют на нашу повседневную жизнь, и каково состояние дел в этой области человеческих знаний сегодня. В книге рассматривается крайне широкий спектр вопросов, связанных с алгоритмами — это и стандартные сортировка и поиск, и работа с графами (но с уклоном не в стандартные базовые алгоритмы, а в приложении их к таким явлениям сегодняшнего дня, как, например, социальные сети), работа с большими данными и вопросы искусственного интеллекта. При этом материал книги — не просто отвлеченный рассказ о том или ином аспекте современных алгоритмов, но и демонстрация реализаций алгоритмов с конкретными примерами на языке программирования Python. Книга будет полезна всем, кто интересуется современным состоянием дел в области программирования и алгоритмов.

ISBN 978-5-9909446-2-6 **в продаже**

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ ДЛЯ ЧАЙНИКОВ

Джон Пол Мюллер
Лука Массарон

www.dialektika.com

Искусственный интеллект является захватывающим и немного жутковатым. Он вокруг нас. Искусственный интеллект помогает защитить мошенничества, контролировать расписание медицинских процедур, он способен работать в клиентской службе и даже помогает вам в выборе телешоу и приборке вашего дома. Хотите узнать больше? Эта книга восполняет пробелы, знакомя вас с тем, что представляет собой искусственный интеллект и чем он не является, рассматриваются также этические вопросы использования искусственного интеллекта, его современное применение и некоторые из удивительных вещей, на которые он, вероятно, будет способен завтра. Будь вы технофилом или просто любопытны, вы будете очарованы тем, что узнаете!

ISBN 978-5-907114-57-9 **в продаже**

Г-н ЖУРДЕН. Честное слово, я и не подозревал, что вот уже более сорока лет говорю прозой.

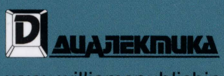
Жан-Батист Мольер, "Мещанин во дворянстве"

Вы всегда считали, что алгоритм — это что-то из мира неопрятных хакеров, освещенных экранами в полутемных помещениях, забитых компьютерами? Что это нечто "не от мира сего", для обычного человека находящееся за гранью понимания?

Вы непростительно ошибаетесь. Вероятно, это одно из наибольших заблуждений нашего времени — особенно непонятное в свете того, что человечество давно и основательно знакомо с алгоритмами. Прочтя книгу, вы убедитесь в этом.

В книге применен необычный подход к изложению алгоритмов — с использованием литературных произведений и фильмов. Вы никогда не задумывались о том, что Гензель и Гретель, возвращаясь домой по цепочке камешков, используют простой алгоритм с линейным временем работы? Что Шерлок Холмс, расследуя очередное дело, имеет дело со структурами данных — списком подозреваемых, родословным древом, очередью с приоритетами наследников жертвы? Что Индиане Джонсу неоднократно приходится иметь дело с сортировкой и поиском? Что события в фильме "День Сурка", по сути, являются циклом, а в фильме "Назад в будущее" — рекурсией?

Перед вами книга по основам информатики и алгоритмов — но книга, в которой для иллюстрации тех или иных концепций используются такие знакомые большинству из нас литературные произведения и фильмы, что делает и без того увлекательную тему информатики, алгоритмов и вычислений еще более интересной и увлекательной.



www.williamspublishing.com

The MIT Press

ISBN 978-5-907144-08-8