ДЖ. ГЛЕННН БРУКШИР • ДЕННИС БРИЛОВ

КОМПЬЮТЕРНЫЕ НАУКИ

БАЗОВЫЙ КУРС

13-е издание



ДИАЛЕКТИКА

Pearson

СКАЧАНО С WWW.SHAREWOOD.BIZ - ПРИСОЕДИНЯЙСЯ!

Компьютерные науки

БАЗОВЫЙ КУРС

Computer Science

AN OVERVIEW



13th Edition

J. Glenn Brookshear
(Author Emeritus)

Dennis Brylow
Marquette University



Pearson Education, Inc.

Компьютерные науки

БАЗОВЫЙ КУРС

13-е издание

Дж. Гленн Брукшир
Деннис Брилов

Москва • Санкт-Петербург
2019

ББК 32.81
Б89
УДК 004.9

ООО "Диалектика"
Зав. редакцией *А.В. Слепцов*
По общим вопросам обращайтесь в издательство "Диалектика"
по адресу: info@dialektika.com, http://www.dialektika.com

Брукшир, Дж. Гленн, **Брилов**, Деннис.
Б89 Компьютерные науки. Базовый курс, 13-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 992 с. : ил. — Парал. тит. англ.
ISBN 978-5-907144-63-7 (рус.)

ББК 32.81

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 Pearson Education, Inc.

Copyright © 2019 by Dialektika Computer Publishing.
Original English edition Copyright © 2019 by Pearson Education, Inc.

All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with Pearson Education, Inc.

Научно-популярное издание

Дж. Гленн Брукшир, Деннис Брилов

Компьютерные науки. Базовый курс, 13-е издание

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

ISBN 978-5-907144-63-7 (рус.)
ISBN 978-0-13-487546-0 (англ.)

© ООО "Диалектика", 2019
© Pearson Education, Inc., 2019

Оглавление

Предисловие 10

Глава 0. Введение 23

Глава 1. Хранение данных 51

Глава 2. Обработка данных 147

Глава 3. Операционные системы 219

Глава 4. Компьютерные сети и Интернет 267

Глава 5. Алгоритмы 355

Глава 6. Языки программирования 433

Глава 7. Технология разработки программного обеспечения 525

Глава 8. Структуры данных 591

Глава 9. Системы баз данных 653

Глава 10. Компьютерная графика 719

Глава 11. Искусственный интеллект 775

Глава 12. Теория вычислений 847

Приложение А. Код ASCII 902

Приложение Б. Электронные схемы обработки чисел в двоичном дополнительном коде 904

Приложение В. Vole: пример простого машинного языка 908

Приложение Г. Высокоуровневые языки программирования 912

Приложение Д. Эквивалентность итеративных и рекурсивных структур 916

Приложение Е. Ответы к разделам “Вопросы и упражнения” 919

Предметный указатель 978

Содержание

Предисловие 10
Аудитория 10
Что нового в 13-м издании 11
Структура книги 13
Преподавателю 14
Студенту 17
Благодарности 18

Глава 0. Введение 23
0.1. Знакомство с алгоритмами 24
0.2. Происхождение вычислительных машин 27
0.3. Обзор этого курса 34
0.4. Дополнительные темы компьютерных наук 37

Глава 1. Хранение данных 51
1.1. Биты и их хранение 52
1.2. Основная память 62
1.3. Массовая память 66
1.4. Представление информации в виде комбинации двоичных разрядов 74
1.5. Двоичная система счисления 85
1.6. Представление целых чисел 91
1.7. Представление дробных значений 101
1.8. Данные и программирование 108
1.9. Сжатие данных 119
1.10. Ошибки при передаче информации 128

Глава 2. Обработка данных 147
2.1. Архитектура компьютера 148
2.2. Машинный язык 152
2.3. Выполнение программы 162
2.4. Арифметические и логические команды 171
2.5. Взаимодействие с другими устройствами 177
2.6. Манипулирование данными в программе 185
2.7. Другие типы архитектуры компьютеров 199

Глава 3. Операционные системы 219
3.1. Эволюция операционных систем 220
3.2. Архитектура операционных систем 227
3.3. Координация действий машины 238
3.4. Организация конкуренции между процессами 243
3.5. Безопасность 250

Глава 4. Компьютерные сети и Интернет 267
4.1. Основы компьютерных сетей 268
4.2. Интернет 282
4.3. Всемирная паутина: World Wide Web 299

4.4. Протоколы Интернета 311
4.5. Простой пример модели "клиент/сервер" 322
4.6. Кибербезопасность 327

Глава 5. Алгоритмы 355
5.1. Понятие алгоритма 356
5.2. Представление алгоритма 361
5.3. Создание алгоритма 372
5.4. Итерационные структуры 380
5.5. Рекурсивные структуры 394
5.6. Эффективность и правильность 405

Глава 6. Языки программирования 433
6.1. Исторический обзор 434
6.2. Концепции традиционного программирования 448
6.3. Процедурные элементы программ 468
6.4. Реализация языка 479
6.5. Объектно-ориентированное программирование 491
6.6. Программирование параллельных процессов 500
6.7. Декларативное программирование 504

Глава 7. Технология разработки программного обеспечения 525
7.1. Предмет технологии разработки программного обеспечения 526
7.2. Жизненный цикл программного обеспечения 531
7.3. Методологии разработки программного обеспечения 538
7.4. Модульность 541
7.5. Инструменты и методы проектирования 552
7.6. Обеспечение качества программ 564
7.7. Документирование программного обеспечения 569
7.8. Интерфейс "человек–машина" 571
7.9. Право собственности и ответственность за создаваемое программное обеспечение 576

Глава 8. Структуры данных 591
8.1. Базовые структуры данных 592
8.2. Связанные концепции 598
8.3. Реализация структур данных 602
8.4. Небольшой практический пример: реализация бинарного дерева 622
8.5. Специализированные типы данных 628
8.6. Классы и объекты 633
8.7. Указатели в машинном языке 635

Глава 9. Системы баз данных 653
9.1. Общие понятия 654
9.2. Реляционная модель 663
9.3. Объектно-ориентированные базы данных 677
9.4. Обеспечение целостности баз данных 681
9.5. Традиционные файловые структуры 686
9.6. Интеллектуальный анализ данных 697
9.7. Влияние технологий баз данных на общество 701

Глава 10. Компьютерная графика 719
10.1. Предмет компьютерной графики 720
10.2. Что такое 3D-графика 724
10.3. Моделирование 726
10.4. Рендеринг 739
10.5. Моделирование глобального освещения 754
10.6. Анимация 759

Глава 11. Искусственный интеллект 775
11.1. Интеллект и машины 776
11.2. Способность к восприятию 783
11.3. Способность к рассуждению 792
11.4. Дополнительные области исследований 808
11.5. Искусственные нейронные сети 816
11.6. Робототехника 823
11.7. Осмысливание последствий 828

Глава 12. Теория вычислений 847
12.1. Функции и их вычисление 848
12.2. Машины Тьюринга 852
12.3. Универсальные языки программирования 857
12.4. Невычислимые функции 865
12.5. Сложность задач 872
12.6. Криптография с открытым ключом 886

Приложение А. Код ASCII 902

Приложение Б. Электронные схемы обработки чисел в двоичном дополнительном коде 904

Приложение В. Vole: пример простого машинного языка 908
Архитектура машины Vole 908
Машинный язык Vole 909

Приложение Г. Высокоуровневые языки программирования 912
Язык Ada 912
Язык C 913
Язык C++ 913
Язык C# 914
Язык FORTRAN 914
Язык Java 915

Приложение Д. Эквивалентность итеративных и рекурсивных структур 916

Приложение Е. Ответы к разделам "Вопросы и упражнения" 919

Предметный указатель 978

Посвящается Декстеру,
который, я знаю, увлеченно
прочтет эту книгу от корки до корки,
прежде чем ему исполнится восемь.
О, какие места ты найдешь!

Предисловие

Данная книга представляет собой введение в область компьютерных наук. В ней необходимая широта обзора предмета сочетается с достаточно глубоким проникновением в сущность излагаемого материала.

Аудитория

Мы писали эту книгу для двух категорий читателей: для будущих специалистов по компьютерным наукам и для студентов всех других дисциплин. Первая категория читателей, включающая студентов начальных курсов, которые по окончании обучения станут специалистами в области компьютерных наук, как правило, на этом этапе обучения склонна отождествлять весь спектр компьютерных наук с программированием, просмотром веб-страниц и обменом файлами в Интернете, поскольку это, в сущности, именно то, что они видели раньше и с чем им приходилось сталкиваться. Однако область компьютерных наук — это нечто существенно большее. Поэтому студентам-компьютерщикам необходимо продемонстрировать всю глубину и обширность той области знаний, к изучению которой они приступают и в которой планируют специализироваться. В предоставлении этой столь необходимой им информации и состоит назначение данной книги. Она познакомит студентов с обзором всего спектра компьютерных наук, что даст основу для правильной оценки различных курсов, которые им предстоит изучать впоследствии. Такой исследовательский подход в действительности является типовой моделью построения вводных курсов в любой области естественных наук.

Столь широкий охват темы — это также именно то, в чем нуждаются студенты других специальностей, если они намерены стать полноправной частью нынешнего техногенного общества, в котором живут. Курс компьютерных наук для этой аудитории должен помочь ей достичь практического, реалистичного понимания того, что представляет собой все обширное поле этой области знаний, а не быть элементарным ознакомлением с тем, как пользоваться Интернетом, или приобретением минимальных практических навыков по

использованию некоторых пакетов программного обеспечения, наиболее популярных в той или иной области. Разумеется, свое место есть и для подобного обучения, однако данная книга предназначена для предоставления учащимся прочных основ компьютерного образования в целом, которые будут работать на них и в будущем.

При подготовке всех предыдущих изданий этой книги сохранение доступности материала для студентов не технических дисциплин было одной из важнейших наших целей. В результате этот учебник уже не раз успешно использовался как базовый в соответствующих курсах для студентов самого широкого круга дисциплин и уровня образования от высшей школы до выпускников обычных школ. Данное, 13-е, издание книги было подготовлено с соблюдением этой традиции.

Что нового в 13-м издании

В 13-м издании продолжена практика использования для записи примеров программного кода на языке Python и псевдокода, выдержанного в том же стиле, впервые принятая в 12-м издании. Это изменение было введено по нескольким причинам. Прежде всего, эта книга уже содержит немало примеров программного кода на различных языках, включая детализированный псевдокод в нескольких главах. Принимая во внимание, что при работе с книгой читателям потребуется усвоить достаточное количество программного синтаксиса, разумно сделать так, чтобы этот синтаксис отвечал тому языку, с которым им с большой вероятностью придется встретиться в последующих курсах. Что еще более важно, все большее число преподавателей, использующих этот учебник, приходят к заключению, что даже в простейших вводных курсах по использованию компьютеров их слушателям сложно усвоить многие темы при отсутствии программного инструмента для проведения самостоятельных исследований и экспериментов.

Почему именно Python? Выбор языка — это всегда спорный вопрос; любое ваше решение вызовет возмущение у одних и восхищение у других, причем примерно в равной пропорции. Язык Python — это отличный выбор среднего уровня, обладающий такими достоинствами, как:

- четкий, понятный и простой в изучении синтаксис;
- простые примитивы ввода-вывода;
- типы данных и управляющие структуры, очень близкие к примитивам псевдокода, использовавшегося в предыдущих изданиях книги;
- поддержка многих парадигм программирования.

Это зрелый язык с активным сообществом разработчиков и обширным полем онлайн-ресурсов для дальнейшего изучения. Язык Python долгое время сохраняет свои позиции в пятерке наиболее широко используемых языков в индустрии программного обеспечения по многим показателям, и в последнее время наблюдается быстрый рост его использования во вводных курсах компьютерных наук. Он особенно популярен в начальных курсах для непрофильных специалистов в самых разных областях естественных наук, таких как физика и биология. Следует добавить, что язык Python сейчас очень широко используется при создании программного обеспечения для проведения различных научных расчетов в этих областях.

Хотя главный акцент в данной книге сделан на основных концепциях самого широкого диапазона компьютерных наук, использование языка Python также позволяет читателю лучше ощутить вкус программирования, чем это было возможно в предыдущих изданиях, и в то же время эта книга вовсе не является полномасштабным введением в программирование на этом языке. Обсуждение отдельных тем по языку Python организовано в соответствии с существующей структурой книги. Так, в главе 1 обсуждается синтаксис представления в Python данных — целых чисел, чисел с плавающей запятой, символов в коде ASCII и строк в кодах Unicode. В главе 2 представлены операции языка Python, которые соответствуют машинным примитивам, обсуждаемым в остальных частях этой главы. Условные операторы, операторы циклов и функции рассматриваются в главе 5, поскольку именно здесь эти конструкции необходимы для реализации достаточно полного псевдокода, предназначенного для записи алгоритмов. Короче говоря, в книге конструкции языка Python используются лишь для подкрепления рассматриваемых концепций компьютерных наук и вовсе не являются предметом обсуждения.

Как и в предыдущих изданиях, в книгу включены некоторые "изюминки": в ее тексте размещены врезки, содержание которых позволяет лучше осознать связь излагаемого материала с реальным миром. Многие из этих врезок содержат ссылки на источники в Интернете, предоставляющие дополнительную информацию по обсуждаемой теме.

И наконец, при подготовке нового издания содержимое каждой главы было подвержено тщательному пересмотру, были внесены необходимые обновления и изменения, а также исправлены ошибки, выявленные в предыдущем издании.

Структура книги

Материал книги упорядочен в соответствии с восходящим (по уровню сложности) подходом, предусматривающим переход от конкретного к абстрактному — именно такой способ изложения обеспечивает ясную и доступную подачу материала, когда одна тема плавно переходит в другую.

Книга начинается с фундаментальных положений в отношении кодирования информации, хранения данных и архитектуры вычислительных машин (главы 1 и 2); далее рассматриваются построение и функционирование операционных систем (глава 3) и компьютерных сетей (глава 4); изучаются такие темы, как алгоритмы, языки программирования и различные аспекты разработки программного обеспечения (главы 5–7); исследуются технологии повышения доступности информации (главы 8 и 9); рассматриваются некоторые важнейшие приложения и технологии компьютерной графики (глава 10) и искусственного интеллекта (глава 11); а в завершение предлагается введение в абстрактную теорию вычислений (глава 12).

Хотя книга в целом выдержана с соблюдением выбранного восходящего стиля изложения, отдельные ее главы и разделы на удивление независимы и вполне могут изучаться как самостоятельные единицы или произвольным образом переупорядочиваться в выбранной альтернативной последовательности изложения материала. В действительности книга часто использовалась как базовый учебник в курсах, материал в которых подавался в ином порядке. В одном случае изложение материала начиналось с глав 5 и 6, после чего мог выполняться переход к главам 1–4. Мне известен и такой курс, который начинался с абстрактной теории вычислений (глава 12). В иных случаях этот учебник использовался в качестве "отправной точки" и служил лишь своеобразной основой, от которой отходили ответвления в углубленное рассмотрение различных тем в тех или иных областях. С другой стороны, курсы для менее технически ориентированной аудитории могли ограничиваться лишь материалом глав 4 (сети и Интернет), 9 (системы баз данных), 10 (компьютерная графика) и 11 (искусственный интеллект).

На первой странице каждой главы приводится перечень рассматриваемых в ней тем — в этом списке звездочками отмечены такие темы, которые можно считать не обязательными для изучения, а лишь дополнительным, факультативным материалом. В каждой главе это разделы, в которых либо рассматриваются более специфические темы, либо предлагается более углубленное изложение уже рассмотренных тем. В наши намерения входило лишь предоставить вам свои предложения в отношении альтернативной последовательности изучения материала курса. Безусловно, могут иметь место и иные подходы к сокращению его объема. В частности, для тех, кто ищет варианты более быстрого прочтения, мы можем предложить такую последовательность изложения материала.

Раздел	Тема
1.1–1.4	Основы кодирования и хранения данных
2.1–2.3	Архитектура компьютера и машинные языки
3.1–3.3	Операционные системы
4.1–4.3	Сети и Интернет
5.1–5.4	Алгоритмы и их создание
6.1–6.4	Языки программирования
7.1–7.2	Разработка программного обеспечения
8.1–8.3	Представление данных
9.1–9.2	Системы баз данных
10.1–10.2	Компьютерная графика
11.1–11.3	Искусственный интеллект
12.1–12.2	Теория вычислений

Существует ряд тем, которые красной нитью проходят через всю книгу. Одна из них — это то, что компьютерные науки являются весьма динамичной областью знания. Все рассматриваемые темы подаются в исторической перспективе, обсуждается достигнутый на данный момент уровень развития и указываются основные направления текущих исследований. Другая тема состоит в пояснении значения абстрактных методов и способов применения различных абстрактных инструментов для управления уровнем сложности. Эта тема вводится в главе 0, а затем вновь и вновь поднимается в контексте архитектуры операционных систем, построения компьютерных сетей, создания новых алгортмов, разработки языков программирования, создания программного обеспечения, представления данных и компьютерной графики.

Преподавателю

Объем материала этой книги превосходит тот, который может быть изучен за один семестр, поэтому не бойтесь пропускать темы, которые не соответствуют задачам вашего курса. Мы написали книгу для того, чтобы она служила основой для проведения курса обучения, а не определяла его содержание. Несмотря на то что изложение материала следует определенной схеме, каждая из тем подается независимо, что позволяет вам делать выбор в соответствии

с вашими вкусами. Мы также предлагаем рассматривать некоторые темы как задания для домашнего чтения. Нам кажется, что зачастую мы недооцениваем студентов, когда считаем необходимым объяснять абсолютно все непосредственно на занятиях. Я часто задаю моим студентам целую главу для чтения на дом, а затем использую время занятий для того, чтобы разъяснить определенные вопросы или подробно осветить некоторые части текста, исходя из собственного опыта. Мы прежде всего должны помочь студентам научиться учиться самостоятельно.

Я уже указывал, что книга построена по восходящему принципу "от конкретного к абстрактному", однако позвольте мне остановиться на этом подробнее. Как ученые мы слишком часто полагаем, что студенты непременно оценят наш подход к предмету, который вырабатывался нами на протяжении многих лет работы в этой области. Однако как преподаватели мы поступим лучше, если будем подавать материал, ориентируясь на точку зрения студента. Именно поэтому книга начинается с освещения методов представления и хранения данных, архитектуры компьютеров, операционных систем и компьютерных сетей. Современные студенты наверняка уже слышали такие термины как "JPEG" и "MP3", они знакомы с компакт-дисками и флеш-памятью, умеют работать с разными операционными системами, ежедневно пользуются смартфонами и Интернетом. Как я обнаружил на опыте, они проявляют немалый интерес к тому, что это такое и как все это работает. Начиная свой курс с этих тем, я часто наблюдал, как они находят ответы на многие свои "почему?", после чего начинают воспринимать этот курс скорее как практический, нежели как теоретический. После такого начала вполне естественно перейти к более абстрактным темам — алгоритмам, алгоритмическим структурам, языкам программирования, методам разработки программного обеспечения, вычислимости и сложности, которые, по нашему мнению, и являются основными темами курса. Как говорилось выше, отдельные темы в книге излагаются в такой манере, которая не обязывает вас следовать выбранной нами последовательности, но мы рекомендуем вам все же попробовать применить ее на практике.

Всем нам известно, что студенты познают гораздо больше того, чему мы их собственно обучаем, и те знания, которые они получают окольным путем, зачастую воспринимаются лучше, чем те, которые подаются им непосредственно. Это становится особенно важным, когда приходит время "учить" решать задачи. Студенты не учатся решать проблемы как отдельную дисциплину путем изучения методологий по решению задач. Они учатся справляться с проблемами, решая их, а не просто заучивая типовое решение тщательно отобранных задач, приведенное в учебниках. Поэтому в эту книгу мы включили многочисленные задания и упражнения, причем некоторые из них преднамеренно сформулированы довольно расплывчато — это означает, что для них необязательно

существует единственный правильный подход или возможно только одно правильное решение. Я настоятельно рекомендую вам использовать такие упражнения и подробно пояснять методы их решения.

Другие темы, которые были отнесены к категории "неявного обучения", — это профессионализм, этика и социальная ответственность. Мы не считаем, что подобный материал может быть представлен как отдельный предмет обсуждения, привязанный к данному курсу. Напротив, он должен выходить на поверхность только там, где это уместно, и именно такой подход выбран в данной книге. В частности, в разделы 4.5, 5.5, 7.9, 9.7 и 11.7 включены такие темы, как безопасность, конфиденциальность, ответственность, социальные аспекты, обсуждаемые в контексте операционных систем, работы в сети, разработки программного обеспечения, использования баз данных и применения искусственного интеллекта. Кроме того, каждая глава включает ряд вопросов (раздел "Общественные и социальные вопросы"), побуждающих студентов к размышлениям об отношении представленного в книге материала к жизни того общества, частью которого они являются.

Мы благодарим вас за то, что вы обратились к этой книге при подготовке собственных курсов. Независимо от того, остановите ли вы на ней свой выбор, мы надеемся, что вы расцените ее как определенный вклад в литературу, посвященную обучению компьютерным наукам.

Педагогические аспекты

Эта книга является плодом многолетней практики преподавания, благодаря чему она богата разнообразным педагогическим материалом. В частности, весьма существенным фактором является обилие предлагаемых задач, решение которых требует активного участия обучающихся — более 1000 в этом издании. Они распределены по разделам "Вопросы и упражнения", "Задания по материалу главы" и "Общественные и социальные вопросы". Подразделы "Вопросы и упражнения" присутствуют в конце каждого раздела каждой главы (за исключением вводной), их назначение — стимулировать обучающихся к самостоятельному мышлению. С помощью предлагаемых задач закрепляется пройденный материал, приведенное выше обсуждение расширяется дополнительными аспектами и даются ссылки на связанный материал, который будет рассматриваться позднее. Ответы на предлагаемые здесь вопросы вынесены в приложение Е.

Разделы "Задания по материалу главы" находятся в конце каждой главы (за исключением вступительной). Они включают подборки задач, разработанных для использования в качестве домашнего задания, поскольку они относятся к содержанию всей главы. По этой же причине в книге ответы на них не приводятся.

За этими разделами следуют разделы под заголовком "Общественные и социальные вопросы", которые содержат вопросы для обдумывания и обсуждения. Многие из этих вопросов могут быть использованы в качестве небольших заданий на проведение исследований, результаты которых должны быть представлены в виде письменных или устных отчетов.

Каждая глава завершается списком рекомендуемой литературы, включающим ссылки на материал, имеющий отношение к теме данной главы. Кроме того, хорошим источником дополнительной информации могут быть веб-сайты, упоминаемые в этом введении, в основном тексте книги и во врезках.

Дополнительные ресурсы

Большое количество дополнительного материала для этой книги можно найти на ее веб-сайте: `www.pearsonhighered.com/brookshear`. На нем любой желающий может получить доступ к следующему.

- Дополнительная информация по каждой главе, дополняющая материал книги и предоставляющая возможность познакомиться со связанными и близкими темами.
- Задания для самопроверки по каждой главе, которые помогают заново обдумать тот материал, который был предложен по этой теме в книге.
- Дополнительные сведения по основам программирования на языке Python, предлагаемые в последовательности, соответствующей изложению материала в книге.

В дополнение к этому на сайте ресурсов для квалифицированных преподавателей издательства Pearson Education (www.pearsonhighered.com) вы найдете следующие материалы для этой книги.

- Руководство для преподавателей с ответами к заданиям из разделов "Задания по материалу главы" всех глав книги.
- Слайды и презентации PowerPoint для лекций.
- Банк тестов.

Студенту

Гленн Брукшир по своему характеру является слегка инакомыслящим (некоторые его друзья считают, что более чем слегка), поэтому, приступив к написанию этой книги, он далеко не всегда следовал тем советам, которые ему давали. В частности, многие утверждали, что определенный материал был представлен на слишком высоком уровне для студентов, только начинающих обучение.

Однако мы уверены, что если тема актуальна, то она будет актуальной, даже если все академическое сообщество сочтет ее "продвинутой". Вы заслуживаете иметь такой учебник, в котором картина компьютерных наук представлена во всей своей полноте, а не "разбавленную" его версию, содержащую сознательно упрощенное представление только тех тем, которые принято считать подходящими для изучающих вводный курс. А это значит, что авторам нельзя было избегать тех или иных тем. Вместо исключения сложной темы нам требовалось найти наилучшие способы ее разъяснения. Мы всегда стремились придать изложению необходимую глубину, предоставить вам истинную картину того, что компьютерные науки представляют собой на самом деле. Как и в случае набора специй, приведенных в рецепте, у вас всегда есть право пропустить некоторые темы, представленные на последующих страницах этой книги, но все они здесь для того, чтобы вы при желании могли попробовать их на вкус, и мы настойчиво рекомендуем вам сделать это.

Мы также должны указать на то, что, как и в случае любого другого учебника, имеющего отношение к технологиям, те сведения, которые вы почерпнете здесь сегодня, могут оказаться не совсем теми сведениями, которые потребуются вам завтра. Эта область знаний весьма динамична, и в этом состоит немалая часть ее притягательности. В нашей книге вы найдете полное отражение *текущей* картины излагаемого предмета вместе с подробной исторической перспективой. И это обеспечит вам надежное основание и подготовит к последующему самостоятельному росту по мере развития технологий. Мы советуем вам начать этот рост немедленно, прямо сейчас, изучая, осваивая и анализируя все, что осталось за рамками этого учебника. Учитесь учиться.

Спасибо за то доверие, которое вы нам оказали, выбрав для чтения эту книгу. Как ее авторы мы были обязаны предоставить вам рукопись, которая будет стоить потраченного на нее временени. И мы надеемся, что ваше решение будет положительным, — мы справились со своими обязательствами.

Благодарности

Прежде всего я хочу поблагодарить Гленна Брукшира, который выносил и выняньчил эту книгу, его кровное детище, поскольку 11 предыдущих изданий охватывают период более четверти века стремительного роста и бурных изменений в области компьютерных наук. Поскольку это всего лишь второе издание, в подготовке которого мне было разрешено участвовать как соавтору и контролировать все вносимые изменения, страницы этого, 13-го, издания по-прежнему в большей части являются голосом самого Гленна и, как я надеюсь, следуют его видению предмета. Любые новые огрехи я принимаю на свой счет, а вся остальная элегантная базовая структура — его.

Я присоединяюсь к благодарности Гленна всем, кто поддерживал эту книгу, читая и используя в своей деятельности ее предыдущие издания. Для нас это большая честь. Тринадцатое издание учебника по компьютерным наукам? Мы, наверное, уже близки к тому, чтобы поставить своего рода рекорд.

Дэвид Т. Смит (David T. Smith) из Университета штата Пенсильвания в Индиане сыграл очень важную роль в моем соавторстве в отношении ревизии 11-издания этой книги, и это все еще заметно и в 13-м издании. Внимательное прочтение Дэвидом предыдущих изданий и тщательная проверка им дополнительных материалов явились очень важным вкладом в подготовительные работы. Эндрю Кюммель (Andrew Kuemmel), Джордж Корлисс (George Corliss) и Крис Мейфилд (Chris Mayfield) — каждый из них предоставил ценные замечания, углубленный анализ и/или поощрительный отзыв в отношении черновиков этого или предыдущих изданий, тогда как Джеймс Е. Эймес (James E. Ames), Стефани Е. Оугуст (Stephanie E. August), Йонсук Чое (Yoonsuck Choe), Мелани Фейнберг (Melanie Feinberg), Эрик Д. Хэнли (Eric D. Hanley), Судхарсан Р. Йенгар (Sudharsan R. Iyengar), Рави Муккамала (Ravi Mukkamala) и Эдвард Приор (Edward Pryor) предложили нам свои ценные замечания в отношении записи программных конструкций в стиле Python в 12-м издании книги.

С каждым новым изданием пополняется список тех, кто внес свой вклад в создание этой книги. На сегодняшний день в этот список входят Дж.М. Адамс (J.M. Adams), К.М. Аллен (C.M. Allen), Д.К.С. Эллисон (D.C.S. Allison), И. Энджел (E. Angel), Р. Ашмор (R. Ashmore), Б. Ауернхеймер (B. Auernheimer), П. Бэнкстон (P. Bankston), М. Бернард (M. Barnard), П. Бендер (P. Bender), К. Боуер (K. Bowyer), П.У. Брешер (P. Brashear), К.М. Браун (C.M. Brown), Х.М. Браун (H.M. Brown), Б. Каллони (B. Calloni), Дж. Карпинелли (J. Carpinelli), М. Клэнси (M. Clancy), Р.Т. Клоуз (R.T. Close), Д.Х. Кули (D.H. Cooley), Л.Д. Корнелл (L.D. Cornell), М.Дж. Кроули (M.J. Crowley), Ф. Дик (F. Deek), М. Дикерсон (M. Dickerson), М.Дж. Дункан (M.J. Duncan), С. Езекиль (S. Ezekiel), К. Фокс (C. Fox), С. Фокс (S. Fox), Н.Е. Гиббс (N.E. Gibbs), Дж.Д. Гаррис (J.D. Harris), Д. Гэском (D. Hascom), Л. Хит (L. Heath), П.Б. Хендерсон (P.B. Henderson), Л. Хант (L. Hunt), М. Хатченрутер (M. Hutchenreuther), Л.А. Джен (L.A. Jehn), К.К. Колберг (K.K. Kolberg), К. Корб (K. Korb), Дж. Кренц (G. Krenz), Дж. Курозе (J. Kurose), Дж. Лью (J. Liu), Т. Дж. Лонг (T.J. Long), К. Мэй (C. May), Дж.Дж. МкКоннел (J.J. McConnell), У. МкКоун (W. McCown), С.Дж. Меррил (S.J. Merrill), К. Мессершмит (K. Messersmith), Дж.К. Мойер (J.C. Moyer), М. Мэрфи (M. Murphy), Дж.П. Майерс (J.P. Myers), Дж.Д.С. Нунен (J.D.S. Noonan), Г. Натт (G. Nutt), У.У. Облити (W.W. Oblitey), С. Оларью (S. Olariu), Г. Риккарди (G. Riccardi), Дж. Райс (G. Rice), Н. Риккерт (N. Rickert), К. Ридесел (C. Riedesel), Дж.Б. Роджерс (J.B. Rodgers), Дж. Сайто (G. Saito), У. Савитч (W. Savitch), Р. Шлефтли (R. Schlafly), Дж.К. Шлиммер (J.C. Schlimmer), С. Селлс (S. Sells), З. Шен

(Z. Shen), Г. Шепперд (G. Sheppard), Дж.К. Симмз (J.C. Simms), М.К. Слэттери (M.C. Slatterry), Дж. Слимик (J. Slimick), Дж.А. Сломка (Slomka), Дж. Солдеритч (J. Solderitsch), Р. Стейгервальд (R. Steigerwald), Л. Стайнберг (L. Steinberg), К.А. Штрубль (C.A. Struble), К.Л. Штрубль (C.L. Struble), У.Дж. Тэффе (W.J. Taffe), Дж. Толбарт (J. Talburt), П. Тонеллато (P. Tonellato), П. Тромович (P. Tromovitch), П.Г. Уинстон (P.H. Winston), И.Д. Уинтер (E.D. Winter), Э. Райт (E. Wright), М. Зиглер (M. Ziegler) и один неизвестный. Всем этим людям мы выражем свою самую искреннюю признательность.

Я также хочу поблагодарить своих друзей из издательства Pearson, оказавших поддержку этому проекту. Трейси Джонсон (Tracy Johnson), Лора Фриденталь (Lora Friedenthal), Эрин Олт (Erin Ault), Кароль Снидер (Carole Snyder) и Скотт Дизанно (Scott Disanno) в особенности приложили свои знания и усилия, внеся множество улучшений в эту книгу в процессе ее подготовки.

И наконец, я благодарю мою жену, Петру, которая целиком брала на себя заботу о наших троих детях на протяжении многих долгих дней и вечеров, пока я работал над этим изданием. Она — моя главная опора.

D.W.B.
Университет Маркетт
1 января 2018 года

Ждем ваших отзывов!

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

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

Отправляя письмо или оставляя сообщение, не забудьте указать название книги и ее авторов, а также свой электронный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

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

Глава 0

В этой, вводной, главе мы познакомимся с областью компьютерных наук, рассмотрим ее в исторической перспективе и заложим необходимые основы, которые позволят приступить непосредственно к изучению материала данного курса.

Введение

0.1 ЗНАКОМСТВО С АЛГОРИТМАМИ

0.2 ПРОИСХОЖДЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

0.3 ОБЗОР ЭТОГО КУРСА

0.4 ДОПОЛНИТЕЛЬНЫЕ ТЕМЫ КОМПЬЮТЕРНЫХ НАУК
- Алгоритмы
- Роль абстракции
- Способность к творчеству
- Данные
- Программирование
- Интернет
- Влияние на общество

Компьютерные науки — это дисциплина, назначение которой состоит в создании научной основы для таких предметов, как проектирование электронно-вычислительных машин, разработка программного обеспечения, обработка информации, алгоритмическое решение задач, а также алгоритмический процесс сам по себе. Следовательно, она представляет собой фундамент для успешного применения современных компьютерных приложений, а также закладывает основы для разработки завтрашней компьютерной инфраструктуры.

Эта книга создана в целях предоставления всеобъемлющего введения в эти науки. В ней рассматривается широкий диапазон тем, включая большинство тех, которые составляют типичную университетскую учебную программу для специалистов по компьютерным наукам. Авторы стремились не только охватить весь спектр, но и показать динамику развития этой области знаний. Вот почему, помимо собственно изложения материала каждой темы, здесь вы найдете исторические сведения о ее развитии, информацию о текущем состоянии проводимых исследований и оценку перспектив ее развития в будущем. Нашей целью было обеспечить конструктивное, деятельное понимание компьютерных наук — такое, которое составит прочную основу для тех, кто желает продолжить свою профессиональную специализацию в этих науках, а также такое, которое позволит тем, кто выбрал иную специализацию, успешно достигать своих целей в современном обществе, которое становится все более и более технократическим.

0.1. Знакомство с алгоритмами

Мы начнем с самого фундаментального понятия в области компьютерных наук — **алгоритма**. Говоря неформально, алгоритм — это последовательность действий, которая определяет способ решения некоторой задачи.[1] Например, существуют алгоритмы для приготовления блюд (в этом случае их называют *рецептами*), поиска пути к нужному месту в неизвестном городе (в этом случае их называют *маршрутами*), для управления стиральными машинами (обычно помещаемые на внутреннюю сторону крышки машины), для воспроизведения музыки (изображаемые в виде *музыкальных нот*), а также для выполнения различных фокусов. Пример алгоритма последнего типа приведен на рис. 0.1.

Прежде чем машина, такая как компьютер, сможет выполнить некоторое задание, необходимо определить алгоритм выполнения этого задания и предоставить его машине в том виде, который будет с ней совместим. Представление алгоритма в таком виде называется **программой**. Для удобства человека текст компьютерных программ обычно печатают на бумаге или отображают на

[1] Если быть более точными, алгоритм — это упорядоченное множество однозначных, выполнимых манипуляций, определяющих некоторое конечное действие. Подробнее эта тема обсуждается в главе 5.

дисплее компьютера. Для удобства машины программы должны быть закодированы в соответствии с технологическими принципами построения машины. Процесс разработки программы, включая ее кодирование в совместимую с выбранной машиной форму, а также процедуру ввода программы в эту машину, принято называть **программированием** или, как еще говорят, **кодированием**. Программы и алгоритмы, которые они представляют, в совокупности принято называть **программным обеспечением**, в противоположность самой машине, которую принято называть **аппаратным обеспечением**.

Описание фокуса. Исполнитель фокуса выкладывает на стол несколько карт из обычной игральной колоды, помещая их лицевой стороной вниз. При этом колода карт постоянно тщательно перетасовывается. Затем зрителям предоставляется право выбора, какого цвета должна быть масть очередной открываемой карты — черного или красного, после чего исполнитель переворачивает на столе требуемую карту.

Секрет фокуса и последовательность его выполнения.

Этап 1. Из обычной колоды карт выберите десять карт красной масти и десять черной. Уложите их в две стопки, согласно цвету масти, лицевой стороной вверх.

Этап 2. Объявите, что вы выбрали несколько карт черной масти и несколько красной.

Этап 3. Выберите карты красной масти. Под видом упорядочения в одну маленькую стопку возьмите карты в левую руку лицевой стороной вниз, после чего большим и указательным пальцами правой руки отогните один из концов этой стопки вниз так, чтобы каждая карта стала слегка выпуклой. Затем со словами: "В этой стопке карты красной масти" положите колоду карт на стол лицевой стороной вниз.

Этап 4. Возьмите карты черной масти. Таким же образом, как и в предыдущем случае, отогните уголок стопки вверх, что сделает карты слегка вогнутыми. Затем положите карты на стол лицевой стороной вниз со словами: "А в этой стопке карты черной масти".

Этап 5. Сразу после того, как карты черной масти будут выложены на стол, перетасуйте обе стопки и приступайте раскладывать их на столе (опять же, лицевой стороной вниз). Раскладывая карты, демонстрируйте зрителям, как тщательно вы их перетасовываете.

Этап 6. Когда все карты будут разложены, выполните следующие действия.

6.1. Попросите зрителей выбрать карту красной или черной масти.

6.2. Если запрашиваемый цвет масти красный и есть карта с выпуклой поверхностью, переверните ее со словами: "Вот карта красной масти".

6.3. Если запрашиваемый цвет черный и есть карта с вогнутой поверхностью, переверните ее со словами: "Вот карта черной масти".

6.4. В противном случае заявите, что больше нет карт требуемого цвета, и переверните все оставшиеся карты, чтобы доказать это.

Рис. 0.1. Алгоритм для выполнения карточного фокуса

Изучение алгоритмов первоначально составляло один из разделов математики. Действительно, поиск алгоритмов привлекал пристальное внимание математиков задолго до того, как появились современные вычислительные машины. Основной целью этих поисков было определение общего набора указаний, описывающих способ решения задачи некоторого типа. Одним из наиболее известных результатов ранних поисков является алгоритм деления столбиком для определения частного двух многозначных чисел. В качестве еще одного примера можно привести алгоритм Евклида, предложенный этим древнегреческим математиком для определения общего наибольшего делителя двух положительных целых чисел. Описание этого алгоритма представлено на рис. 0.2.

Описание. В этом алгоритме предполагается, что входные данные представляют собой два целых положительных числа, для которых требуется определить наибольший общий делитель.

Порядок выполнения.

Этап 1. Присвойте переменным *M* и *N* значения двух введенных чисел (большего и меньшего соответственно).

Этап 2. Разделите *M* на *N* и присвойте значение остатка переменной *R*.

Этап 3. Если значение *R* не равняется 0, присвойте переменной *M* значение переменной *N*, затем переменной *N* присвойте значение остатка *R* и вернитесь к этапу 2. В противном случае наибольшим общим делителем заданной пары чисел является значение, присвоенное в данный момент переменной *N*.

Рис. 0.2. Алгоритм Евклида для поиска наибольшего общего делителя двух положительных целых чисел

Как только алгоритм решения задачи будет найден, само выполнение предусмотренных этим алгоритмом действий уже не потребует понимания законов, по которым данный алгоритм был построен. Напротив, решение задачи сужается до простого выполнения установленной последовательности инструкций. (Мы можем применять алгоритм деления столбиком для поиска частного двух многозначных чисел или Евклидов алгоритм определения наибольшего общего делителя, даже не понимая принципов, на основании которых эти алгоритмы работают.) По сути, в алгоритме закодированы все сведения, необходимые для решения поставленной задачи.

Фиксация и передача интеллектуальной информации (или по крайней мере правил интеллектуального поведения) с помощью алгоритмов позволяет создавать машины, способные выполнять полезные задания. Следовательно, уровень интеллекта, проявляемый машинами, ограничен той информацией, которая может быть передана им через алгоритмы. Только после того, как будет найден алгоритм, позволяющий решить поставленную задачу, может быть сконструировано некоторое устройство, предназначенное для ее решения. В свою

очередь, если не существует алгоритма выполнения определенного задания, то его выполнение оказывается за пределами возможностей машин.

Выявление ограничений алгоритмических возможностей сформировалось в математике как самостоятельная тема в 1930-х годах после публикации Куртом Гёделем теоремы о неполноте. Эта теорема по существу утверждает, что в любой математической теории, охватывающей нашу традиционную арифметическую систему, существуют утверждения, истинность или ложность которых алгоритмически невозможно ни доказать, ни опровергнуть. Осознание этого факта потрясло самые основы математики, и исследования в области изучения алгоритмических возможностей, которые последовали вслед за этим, собственно и положили начало новой области знаний, известной теперь как компьютерные науки. Как видите, именно изучение алгоритмов сформировало ядро компьютерных наук.

0.2. Происхождение вычислительных машин

Современные компьютеры имеют весьма обширную генеалогию. Одним из первых вычислительных устройств является абак, т.е. *счеты*. Историки говорят, что, вероятно, впервые подобные устройства появились в Древнем Китае, но достоверно известно, что счеты уже использовались в ранний период древнегреческой и древнеримской цивилизаций. Само это устройство довольно простое и состоит из бусин, нанизанных на прутья, которые вставлены в прямоугольную рамку (рис. 0.3). Перемещение бусин взад и вперед по прутьям позволяет представлять сохраняемые значения. Именно расположение бусин этот "компьютер" использует для представления и суммирования данных. Управление выполнением требуемого алгоритма с помощью этой машины возлагается на человека-оператора. Таким образом, сами счеты являются просто системой хранения данных, и только сочетание человека и счет образует полную вычислительную машину.

В период с начала эпохи Возрождения и до конца Нового времени неоднократно предпринимались попытки построения более сложных вычислительных машин. В этот период технология создания вычислительных машин строилась на использовании зубчатых колес. Среди создателей подобных механизмов были француз Блез Паскаль (1623–1662), немец Готфрид Вильгельм Лейбниц (1646–1716) и англичанин Чарльз Бэббидж (1792–1871). В их устройствах данные представлялись с помощью определенного расположения зубчатых колес, а исходные значения вводились механически, вручную, посредством приведения колес в необходимое положение. Результаты вычислений в машинах Паскаля и Лейбница определялись путем считывания конечного положения колес, примерно так, как мы сегодня определяем суммарный пробег автомобиля по

показаниям спидометра. Однако Бэббидж предвидел создание машин, которые будут печатать результаты вычислений на бумаге, что позволит устранить возможность ошибок при считывании.

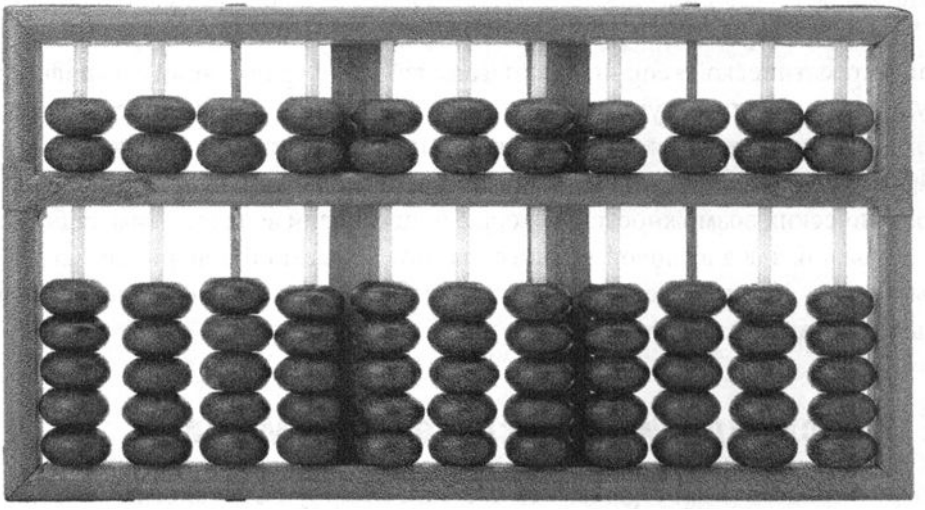


Рис. 0.3. Китайский вариант деревянных счет

Что касается способности следовать алгоритму, то в этих машинах уже явно виден определенный прогресс. Машина Паскаля могла выполнять только алгоритм суммирования, поэтому средства для выполнения соответствующей последовательности действий были встроены в саму машину. Аналогичным образом в архитектуру машины Лейбница был встроен набор неизменных алгоритмов, позволяющих выполнять множество арифметических действий по выбору оператора. Машина Бэббиджа, в отличие от двух предыдущих машин, была сконструирована таким образом, что последовательность выполняемых действий могла быть передана с помощью отверстий в бумажных картах. Таким образом, машина Бэббиджа была уже *программируемой*. Именно по этой причине ассистентка Бэббиджа, Августа Ада Байрон, опубликовавшая статью о том, как именно можно было программировать работу аналитической машины Бэббиджа для выполнения различных вычислений, теперь считается первым в мире программистом.

Идея передачи алгоритма с помощью отверстий в бумажных картах не является собственным открытием Бэббиджа. В 1801 году француз Джозеф Жаккард (1752–1834) применил подобную технологию для управления ткацкими станками. В частности, он разработал ткацкий станок, процесс плетения которого определялся узором из отверстий, нанесенных на последовательность больших карт из толстой бумаги. Благодаря этому алгоритм, по которому работала машина, можно было легко изменить, что позволяло на одном и том же

станке производить множество различных типов тканей. Позднее Герман Холлерит (1860–1929) использовал идею представления информации с помощью отверстий в бумажных картах для ускорения составления таблиц статистических сводок при переписи населения США в 1890 году. Фактически именно эта разработка Холлерита привела к созданию корпорации IBM. Такие карты в конечном итоге стали известны как перфокарты и сохранились как популярный способ общения с компьютерами вплоть до 1990-х годов.

Технологии тех времен не обеспечивали необходимого уровня точности, который позволил бы сделать сложные шестеренчатые калькуляторы Паскаля, Лейбница и Бэббиджа достаточно эффективными и популярными. И до тех пор, пока в начале 1900-х годов электроника не расширила возможности механических устройств, технология не позволяла поддерживать те теоретические разработки, которые появлялись в зарождающейся компьютерной науке того времени. Примерами такого прогресса могут служить электромеханическая машина Джорджа Стибица, созданная в 1940 году в лабораториях компании Bell, и машина Mark I, созданная в 1944 году в Гарвардском университете Говардом Айкеном совместно с группой инженеров корпорации IBM. В этих машинах широко использовались механические реле, работой которых управляла электроника. В этом смысле они устарели практически сразу же после создания, так как другие исследователи в это же время уже использовали технологию электровакуумных приборов для конструирования полностью электронных цифровых вычислительных машин. Первым таким устройством была машина Атанасова-Берри, создававшаяся с 1937 по 1941 год в колледже шт. Айова (сегодня — Университет шт. Айова). Создателями машины были Джон Атанасов и его ассистент Клиффорд Берри. Другим аналогичным устройством является машина COLLOSSUS, созданная в Англии под руководством Томми Флаурса в конце второй мировой войны для расшифровки перехватываемых немецких шифрованных сообщений. (В действительности было создано, вероятно, не менее 10 таких машин, однако режим военной секретности и вопросы национальной безопасности требовали сохранения их существования в тайне, из-за чего эти машины так и не стали общеизвестной частью "генеалогического древа" компьютеров.) За ними вскоре последовали другие, более универсальные машины, например ENIAK (электронный цифровой интегратор и калькулятор), разработанный Джоном Мочли и Дж. Преспером Экертом в электротехнической школе Мура, Университет шт. Пенсильвания (рис. 0.4).

С этого момента дальнейшая история вычислительных машин в большей степени определялась прогрессом в области передовых технологий, включая изобретение транзисторов (за что физики Вильям Шокли, Джон Бардин и Уолтер Братайн получили Нобелевскую премию) и последующее развитие технологии интегральных схем (за что Нобелевскую премию в области физики

получил уже Джек Кибли). Благодаря этим разработкам громоздкие, размером с комнату, машины 1940-х годов за пару десятилетий уменьшились до размеров обычного шкафа. В то же время вычислительная мощность этих компьютеров удваивалась примерно каждые два года (эта тенденция сохраняется и до наших дней). По мере развития технологии создания интегральных схем многие компоненты компьютеров были выпущены на рынок в виде отдельных интегрированных элементов, заключенных в небольшие пластиковые корпуса, получившие название "чипы".

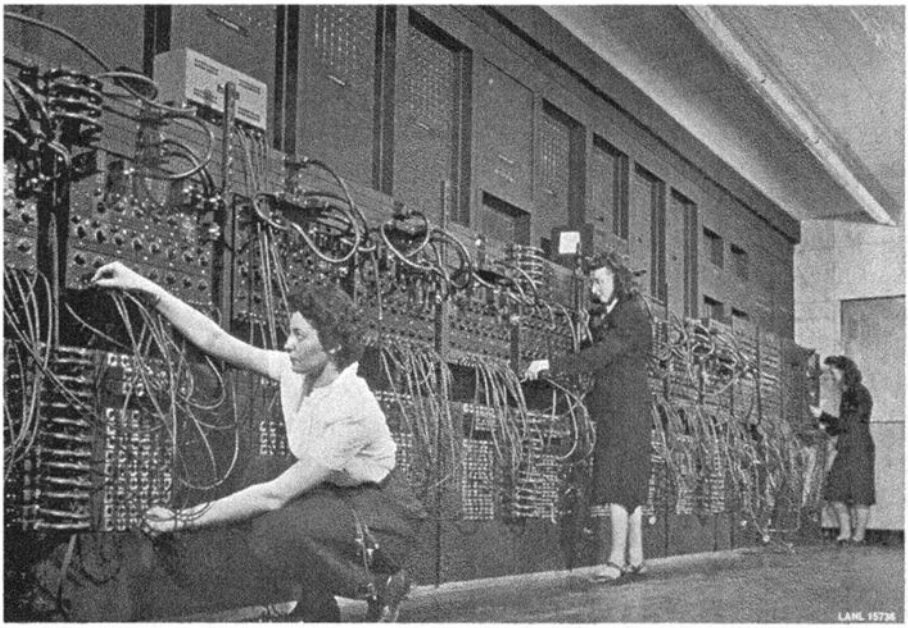
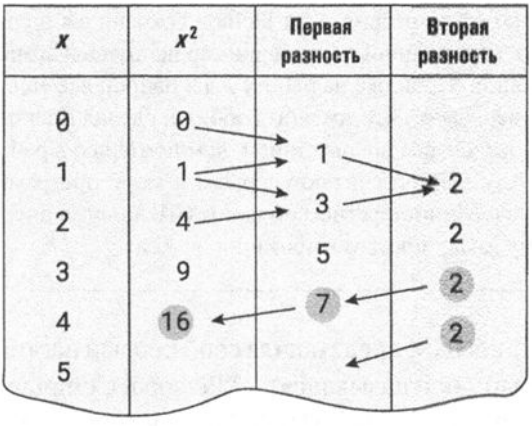


Рис. 0.4. Три женщины контролировали работу машины ENIAC, пользуясь ее главной панелью управления, пока эта машина находилась в школе Мура. Позднее этот компьютер был перевезен в лабораторию баллистических исследований армии США

Огромным шагом вперед в популяризации вычислительных машин было создание настольных компьютеров. Начало созданию таких малогабаритных машин положили те люди, для которых компьютеры были предметом увлечения. Именно они начали экспериментировать с машинами, сконструированными в домашних условиях в виде комбинации чипов. Благодаря этой "подпольной" любительской деятельности Стив Джобс (Steve Jobs) и Стефен Возняк (Stephen Wozniak) впервые построили коммерчески жизнеспособный домашний компьютер и в 1976 году основали компанию Apple Computer, Inc., специализировавшуюся на изготовлении и продаже подобных изделий. Другими

Разностная машина Бэббиджа

Машины, сконструированные Чарльзом Бэббиджем, по своей структуре были настоящими предшественниками современных компьютеров. Если бы технологии тех времен позволяли выпускать эти машины экономически выгодным образом, а требования к обработке данных со стороны коммерческих и правительственных кругов были сопоставимы с современными, идеи Бэббиджа могли бы привести к компьютерной революции еще в 1800-х годах. В действительности во время его жизни была создана только демонстрационная модель его разностной машины. Эта машина определяла числовые значения посредством вычисления "конечных разностей". Получить представление об этом методе можно, например, рассмотрев проблему вычисления квадратов целых чисел. Начнем с известного нам факта, что квадрат нуля равен нулю, квадрат 1 равен 1, квадрат 2 равен 4, а квадрат 3 равен 9. Приняв это во внимание, можно вычислить квадрат 4 следующим образом (см. рисунок ниже). Сначала находим разность уже известных нам квадратов: $1^2 - 0^2 = 1$, $2^2 - 1^2 = 3$ и $3^2 - 2^2 = 5$. Затем вычисляем разности этих результатов: $3 - 1 = 2$ и $5 - 3 = 2$. Обратите внимание, что обе эти разности равны 2. Предположив, что это постоянство значения разностей сохраняется и далее (математики могут доказать, что это именно так), мы можем заключить, что разность между значениями $(4^2 - 3^2)$ и $(3^2 - 2^2)$ также должна быть равна 2. Тогда разность $(4^2 - 3^2)$ должна быть на 2 больше, чем разность $(3^2 - 2^2)$, а значит, $4^2 - 3^2 = 7$, а отсюда $4^2 = 3^2 + 7 = 16$. Теперь, когда мы знаем значение квадрата 4, мы можем повторить эту процедуру для вычисления квадрата 5, воспользовавшись уже известными значениями 1^2, 2^2, 3^2 и 4^2. (Хотя более подробное обсуждение метода конечных разностей выходит за рамки этого курса, студенты, знающие дифференциальное исчисление, могут заметить, что предыдущий пример построен на том факте, что производной функции $y = x^2$ является прямая линия с наклоном 2.)



компаниями, которые вслед за Apple начали выпускать подобную продукцию, были Commodore, Heathkit и Radio Shack. Несмотря на то что продукция этих компаний была популярна среди многочисленных любителей, она не получила широкого признания в деловых кругах, которые продолжали рассматривать респектабельную корпорацию IBM как главный источник удовлетворения своих потребностей в вычислительной технике.

Августа Ада Байрон

Августа Ада Байрон, графиня Лавлейс, многие годы является объектом полемики и восторженных комментариев в компьютерных кругах. Она прожила относительно недолгую, 37 лет (1815–1852), и в чем-то трагическую жизнь, осложненную слабым здоровьем и тем фактом, что была инакомыслящей в обществе, которое жестко ограничивало профессиональные возможности женщин. Хотя Ада проявляла интерес к широкому спектру наук, главные силы она направила на изучение математики. Интерес к "вычислительной науке" появился у нее после того, как она была очарована машиной Чарльза Бэббиджа на демонстрации прототипа его разностной машины в 1833 году. Ее вклад в компьютерные науки связан с выполненным ею переводом с французского на английский статьи, в которой описывался проект аналитической машины Бэббиджа. Бэббидж предложил ей добавить к этому переводу свои комментарии, в которых описывались бы возможные применения машины и приводились бы примеры того, как эту машину можно запрограммировать для выполнения различных заданий. Энтузиазм Бэббиджа по отношению к работе Ады Байрон был, по всей видимости, вызван его надеждами получить в результате этой публикации финансовую поддержку, необходимую для построения его аналитической машины. (Как дочь лорда Байрона Ада Байрон имела статус знаменитости с потенциально значимыми финансовыми связями.) Эти надежды так никогда и не оправдались, однако комментарий Ады Байрон сохранился и со временем был признан как документ, содержащий первый пример настоящей компьютерной программы. Степень влияния Бэббиджа на работу Ады Байрон все еще дискутируется историками. Одни утверждают, что именно Бэббидж сделал главный вклад, тогда как другие — что он был скорее препятствием, чем помощью в работе Ады. В любом случае сегодня Аду Байрон считают первым в мире программистом, и этот статус был подтвержден Министерством обороны США, когда оно в ее честь дало название популярному языку программирования — Ada.

В 1981 году корпорация IBM представила свой первый настольный персональный компьютер, который так и назывался — "Personal Computer" (персональный компьютер), для краткости — "PC" (ПК). Базовое программное обеспечение для этого компьютера было разработано молодой энергичной компанией, ныне известной как Microsoft. Эта модель персонального компьютера очень быстро получила признание и возвела настольный компьютер в ранг общепризнанного

предмета потребления для деловых кругов общества. Сегодня термин "ПК" широко используется для обозначения всех тех устройств (изготовленных различными производителями), базовые модели которых ведут свое начало от первоначального настольного компьютера корпорации IBM. Большинство таких машин по-прежнему выпускается на рынок с программным обеспечением от фирмы Microsoft. Со временем термин "ПК" и исходный термин "настольный компьютер" (desktop) стали практически взаимозаменяемыми.

К концу XX столетия возможность соединить отдельные компьютеры в единую систему мирового масштаба, получившую название **Интернет** (Internet), привела к настоящей революции в средствах коммуникации. В этом контексте Тим Бернерс-Ли (Tim Berners-Lee), британский ученый, предложил развернуть систему, посредством которой документы, хранящиеся на компьютерах, подключенных к Интернету, можно было связать между собой, создав обширную сеть связанной информации, получившей название **веб** (Web) — сокращение от полного названия "World Wide Web" (Всемирная паутина). Чтобы сделать информацию в вебе доступной тем, кто в ней нуждается, были разработаны программные системы, получившие название **поисковые машины** (search engines). Их назначение — "просеивать" содержимое веба, "классифицировать" найденное и использовать полученные результаты для оказания помощи тем, кто ищет информацию по определенным темам. Главными игроками на этом поприще сейчас являются Google, Яндекс, Yahoo и Microsoft. Эти компании продолжают расширять свою активность в вебе, часто в таких направлениях, которые бросают вызов нашему традиционному образу мышления.

Пока настольные, а затем и переносные (лэптопы и ноутбуки) персональные компьютеры получали все большее распространение и популярность в частных домах, миниатюризация вычислительных машин продолжалась во все возрастающей степени. Сегодня крошечные компьютеры непосредственно встраиваются в широчайший спектр электронных устройств и приборов. Обыкновенный автомобиль сейчас может содержать десятки маленьких компьютеров, обеспечивающих работу службы GPS (Global Positioning Systems — глобальная система позиционирования), контролирующих работу двигателя и даже обеспечивающих обработку голосовых команд для управления аудиосистемой автомобиля и его системой телефонной связи.

Вероятно, наиболее революционным применением компьютерной миниатюризации явилась разработка и последующее расширение возможностей **смартфонов**, наручных универсальных компьютеров, в которых собственно телефония является лишь одним из множества доступных приложений. Более мощные, чем суперкомпьютеры предыдущих десятилетий, эти устройства карманных размеров оборудованы множеством датчиков и интерфейсов, включая

Google

Основанная в 1998 году, корпорация Google LLC (ранее — Google, Inc.) в настоящее время стала одной из наиболее известных и влиятельных технологических компаний. Ее основной сервис, поисковая машина Google, ежедневно используется многими миллионами людей для поиска документов в World Wide Web. Кроме того, корпорация Google предоставляет всем желающим бесплатную службу электронной почты (под названием "Gmail"), развернутую в Интернете чрезвычайно популярную в мире службу хранения и просмотра видеоклипов (под названием "YouTube"), а также обеспечивает работу многих других популярных интернет-сервисов, таких как Google Maps, Google Calendar, Google Earth, Google Books или Google Translate.

Однако, помимо того что ее можно считать истинным образцом духа предпринимательства, корпорация Google является еще и ярким примером того, как стремительное развитие и повсеместное распространение технологий может оказывать влияние на жизнь и состояние общества. Например, всемирная экспансия поисковой машины Google привела к появлению вопросов о том, в какой степени международная компания должна соблюдать пожелания отдельных правительств; в связи с ростом популярности сервиса YouTube были подняты вопросы в отношении того, в какой степени компания должна нести ответственность за информацию, которую сторонние пользователи распространяют через ее сервисы, а также о той степени, в которой компания может заявить о своем праве собственности на эту информацию; сервис Google Books вызвал озабоченность в отношении объема и ограничений прав на интеллектуальную собственность, а сервис Google Maps обвиняют в нарушении прав на неприкосновенность частной жизни.

многокомпонентные фото- и видеокамеры, микрофоны, компасы, сенсорные экраны, датчики GPS, акселерометры (для определения ориентации и перемещения устройства), а также поддерживающие разнообразные беспроводные технологии, обеспечивающие возможность коммуникации с другими смартфонами и компьютерами. Многие считают, что появление смартфонов оказало на глобальное сообщество даже большее влияние, чем революция, вызванная появлением персональных компьютеров.

0.3. Обзор этого курса

В этой книге материал упорядочен в соответствии с подходом к изучению компьютерных наук, предусматривающим переход от конкретного к абстрактному. Она начинается с таких практических тем, как устройство и оборудование компьютеров, а завершается рассмотрением таких абстрактных аспектов, как сложность алгоритмов и вычислимость функций. В результате изучение

предмета следует схеме построения все более и более абстрактных инструментов, поскольку наше понимание предмета расширяется.

В начале рассматриваются темы, относящиеся к проектированию и конструированию машин для выполнения алгоритмов. В главе 1, "Хранение данных", речь идет о том, как информация кодируется и сохраняется в современных компьютерах, а в главе 2, "Обработка данных", рассмотрены основные внутренние операции в простом компьютере. Хотя определенная часть материала этой книги связана с технологиями, общая ее тема не зависит от конкретных технологий. То есть такие темы, как структура цифровых электронных схем, кодирование данных, способы их сжатия, а также архитектура компьютеров, фактически относятся к широкому спектру технологий и, скорее всего, будут оставаться актуальными независимо от направления развития технологий в будущем.

В главе 3, "Операционные системы", рассказывается о программном обеспечении, управляющем работой компьютера в целом. Такое программное обеспечение принято называть *операционной системой*. Именно операционная система компьютера управляет работой интерфейса между этой машиной и внешним миром, защищая машину и хранящиеся в ней данные от несанкционированного доступа, позволяя пользователю компьютера запускать на выполнение различные программы и координируя внутреннее взаимодействие элементов компьютера, необходимое для выполнения запросов пользователя.

В главе 4, "Компьютерные сети и Интернет", объясняется, как компьютеры могут быть соединены между собой для образования компьютерных сетей и как сети подключаются одна к другой с целью образования их объединений — *интернетов*. В этой главе также обсуждаются такие темы, как сетевые протоколы, структура и принципы управления Всемирной сети — Интернета и развернутой на ее основе распределенной системы World Wide Web, а также даются необходимые сведения по вопросу защиты.

В главе 5, "Алгоритмы", изучение алгоритмов проводится с более формальной точки зрения. В ней поясняется как алгоритмы создаются, приводится несколько фундаментальных алгоритмических структур и обсуждаются простейшие способы представления алгоритмов. В заключение вводятся такие важные понятия, как эффективность и правильность алгоритмов.

В главе 6, "Языки программирования", рассматриваются представление алгоритмов и разработка программ. Вы узнаете, что поиск лучших методов программирования в конечном счете привел к выработке различных методологий программирования или *парадигм*, для каждой из которых был разработан собственный набор языков программирования. Помимо парадигм и отдельных языков, в главе обсуждаются вопросы построения грамматики различных языков и их трансляции.

В главе 7, "Технология разработки программного обеспечения", обсуждается та ветвь компьютерных наук, которая связана с проблемами, возникающими при создании больших программных систем. Причина их возникновения состоит в том, что проектирование и создание крупных программных систем является сложной, комплексной задачей, порождающей проблемы, выходящие за рамки традиционной инженерии. В результате задача разработки новых технологий создания программного обеспечения становится важным полем исследований в рамках компьютерных наук, затрагивающим такие аспекты, как методы проектирования, управление проектами, управление персоналом, разработка языка программирования и даже архитектура систем.

В следующих двух главах рассматриваются способы организации хранения данных и доступа к ним в компьютерных системах. В главе 8, "Структуры данных", речь идет о методах, традиционно используемых для организации данных в основной памяти компьютера, и прослеживается эволюция абстракции данных от концепций примитивов до современных объектно-ориентированных методологий. В главе 9, "Системы баз данных", рассматриваются методы, традиционно используемые для организации данных в массовой памяти компьютеров и объясняется, как могут быть реализованы чрезвычайно большие и сложные системы современных баз данных.

В главе 10, "Компьютерная графика", речь идет о компьютерной графике и анимации — области компьютерных наук, которая имеет дело с созданием и отображением виртуальных миров. Основываясь на достижениях в более традиционных областях компьютерных наук, таких как архитектура машин, разработка алгоритмов, структуры данных и технология разработки программного обеспечения, результаты исследований в области компьютерной графики и анимации продемонстрировали колоссальный прогресс, превратив это направление в динамичную, захватывающую и процветающую область применения компьютеров. Более того, здесь можно найти яркие иллюстрации тому, как различные дисциплины компьютерных наук успешно сочетаются с дисциплинами других областей науки и культуры, такими как физика, искусство и фотография, давая поразительные результаты.

В главе 11, "Искусственный интеллект", рассказывается, что с целью разработки более полезных человеку машин исследователи в области компьютерных наук обратились к изучению человеческого интеллекта, стремясь углубить свое понимание этого феномена. Главная цель — понять, как в нашем сознании осуществляются рассуждение и восприятие, что позволит исследователям разработать алгоритмы, копирующие эти процессы, а значит, передать сравнимые возможности машинам. Результатом явилось появление еще одной области компьютерных наук, получившей название *искусственный интеллект*, в значительной степени опирающейся на исследования в таких областях знаний, как психология, биология и лингвистика.

Завершается эта книга главой 12, "Теория вычислений", в которой исследуются теоретические основы компьютерных наук — предмет, который позволяет понять ограничения, существующие в отношении алгоритмов (а значит, и алгоритмических машин). Здесь приводятся примеры некоторых проблем, которые невозможно решить алгоритмически (а значит, решение которых лежит и за пределами возможностей машин), наравне с демонстрацией того, что решение многих других проблем может потребовать таких невероятных затрат времени или пространства, что их также можно считать неразрешимыми с практической точки зрения. Таким образом, именно благодаря подобным исследованиям мы можем понять область применения и существующие ограничения возможностей алгоритмических систем.

В каждой главе нашей целью было исследовать тему с глубиной, достаточной для достижения истинного ее понимания. Мы стремились предоставить практические, действующие знания в области компьютерных наук — такие, которые позволят читателю лучше понимать то технократическое общество, в котором он живет, и создадут прочную основу, на которую он сможет уверенно опереться в своем дальнейшем продвижении в науке и технике.

0.4. Дополнительные темы компьютерных наук

В каждой главе, помимо основных тем, перечисленных выше, мы также надеемся расширить ваше понимание компьютерных наук посредством включения нескольких всеобъемлющих тем. Миниатюризация компьютеров и постоянное расширение их возможностей вывело компьютерные технологии на передний край в современном обществе, и эти технологии настолько превалируют, что близкое знакомство с ними является совершенно необходимым условием для того, чтобы ощущать себя частью современного мира. Компьютерные технологии изменили возможности правительств осуществлять контроль; оказали колоссальное влияние на мировую экономику; привели к поразительным достижениям в области научных исследований; внесли революционные изменения в сбор, хранение и использование данных; предоставили людям новые средства общения и взаимодействия и неоднократно бросили вызов текущему состоянию общества, вынуждая его изменяться. Результатом стало бурное развитие областей знаний, прилежащих к компьютерным наукам, каждая из которых сейчас сама по себе является важной областью изучения и исследований. Более того, как и в случае машиностроения и физики, часто трудно провести разграничительную линию между этими научными областями и компьютерными науками. Таким образом, чтобы дать правильные перспективы, в этой книге следовало не только рассматривать основные темы ядра компьютерных наук, но и проводить экскурсы в различные дисциплины, связанные как с

практическим приложением, так и с последствиями развития науки. Действительно, любое введение в компьютерные науки должно пониматься как задача междисциплинарная.

Приступая к изучению области компьютерных наук во всей ее широте, полезно всегда держать в уме те главные темы, которые объединяют компьютерные науки в единое целое. Их всего семь — "Большая семерка" основных концепций включает: алгоритмы, абстракция, творчество, данные, программирование, Интернет и воздействие. В следующие главы мы включили ряд тем, каждый раз представляя их центральные положения, текущую область исследований и некоторые из методов, используемых для расширения знаний в этой области. Помните о Большой семерке. потому что мы будем возвращаться к ней снова и снова.

Алгоритмы

Ограниченные возможности хранения данных и использование детального, требующего больших затрат времени, программирования ограничивали сложность алгоритмов, которые могли выполнять первые вычислительные машины. Однако по мере того как эти ограничения преодолевались, компьютеры стали применяться к решению все более и более сложных задач. Когда попытки выразить структуру этих задач в алгоритмической форме стали требовать чрезмерных умственных усилий, все больше и больше исследований было направлено на изучение самих алгоритмов и процесса программирования.

Именно тогда теоретическая работа математиков начала приносить свои плоды. В результате появления теоремы Геделя о неполноте к моменту создания первых вычислительных машин математики уже в достаточной степени исследовали те аспекты алгоритмических процессов, которые были необходимы развивающейся технологии. Этим была заложена основа для появления новой дисциплины, ныне известной как *компьютерные науки*.

Сегодня эта дисциплина зарекомендовала себя как наука об алгоритмах. Как мы уже убедились, ее границы достаточно широки, поскольку она использует знания из таких дисциплин, как математика, инженерное искусство, психология, биология, менеджмент и языкознание. На самом деле исследователи в различных ответвлениях компьютерных наук могут иметь очень отличающееся определение для этой области науки. Например, тот, кто работает в области архитектуры компьютеров, может сфокусироваться на задаче миниатюризации компьютерных схем, а значит, видит компьютерные науки как развитие и применение технологии. Однако специалист, работающий в области систем баз данных, может представлять компьютерные науки как поиск возможностей сделать информационные системы более полезными. Наконец, исследователь в

области искусственного интеллекта может понимать компьютерные науки как изучение интеллекта и интеллектуального поведения.

Но как бы там ни было, все эти специалисты в той или иной степени связаны с определенными аспектами науки об алгоритмах.

Принимая во внимание ту центральную роль, которую алгоритмы играют в компьютерных науках (рис. 0.5), будет целесообразно определить некоторые вопросы, которые помогут сосредоточиться в нашем изучении на этой важнейшей идее.

- Какие проблемы могут быть решены с помощью алгоритмических процессов?
- Как можно упростить задачу поиска требуемого алгоритма?
- Каким образом можно усовершенствовать технологию представления и передачи алгоритмов?
- Как можно анализировать и сравнивать свойства различных алгоритмов?
- Как алгоритмы можно использовать для манипулирования информацией?
- Как можно применить алгоритмы к демонстрации интеллектуального поведения?
- Какое воздействие алгоритмы оказывают на общество?



Рис. 0.5. Основная роль алгоритмов в компьютерных науках

Роль абстракции

Термин **абстракция** в том понимании, которое мы ему здесь придаем, ссылается на различие между внешними свойствами объекта и деталями внутреннего устройства этого объекта. Именно абстракция позволяет игнорировать внутренние детали устройства сложных объектов, таких как компьютер, автомобиль

или микроволновая печь, и использовать их как целостную, доступную пониманию вещь. Более того, именно посредством абстракции такие сложные системы могут быть разработаны и изготовлены. Компьютеры, автомобили и микроволновые печи сконструированы из компонентов, каждый из которых представляет некоторый уровень абстракции, на котором использование этих компонентов рассматривается изолировано от их внутреннего устройства.

Именно благодаря использованию абстракции мы получаем возможность конструировать, анализировать и управлять большими, сложными компьютерными системами, что было бы совершенно невыполнимым, если бы они рассматривались на их внутреннем, детальном уровне. На каждом уровне абстракции мы рассматриваем систему в терминах компонентов, называемых **абстрактными инструментами**, внутреннее устройство которых игнорируется. Это позволяет сконцентрироваться на том, как каждый компонент взаимодействует с другими компонентами на том же уровне и как вся их совокупность образует компонент более высокого уровня. Действуя таким образом, мы можем понять работу той части системы, которая имеет отношение к задаче, и не потеряться в море деталей.

Следует подчеркнуть, что использование абстракции не ограничивается лишь наукой и технологией. Абстракция является важным методом упрощения, с помощью которого наше общество создало тот образ жизни, который иначе создать было бы просто невозможно. Например, немногие из нас понимают, как на самом деле реализуются различные удобства, которыми мы пользуемся в повседневной жизни. Мы употребляем пищу и носим одежду, которую не способны производить самостоятельно. Мы используем электроприборы, не понимая принципов, положенных в основу их функционирования. Мы пользуемся услугами разных людей, не вникая в подробности их деятельности. С каждым новым достижением лишь небольшая часть общества стремится профессионально специализироваться в этой области, в то время как остальные лишь учатся пользоваться достигнутыми результатами, воспринимаемыми как абстрактные инструменты, внутреннее устройство которых нам понимать не требуется. В результате накопленный объем абстрактных средств расширяется, повышая способность общества к дальнейшему продвижению вперед.

В этой книге абстракция встречается вновь и вновь. Вы узнаете, что компьютерное оборудование конструируется на уровне абстрактных инструментов, а разработка больших систем программного обеспечения осуществляется по модульному принципу, в котором каждый модуль воспринимается как абстрактный инструмент построения более крупных модулей. Более того, абстракция играет важную роль в задаче развития самих компьютерных наук, позволяя исследователям сосредоточить свое внимание на определенных аспектах сложной задачи. В действительности даже сама организация этой книги отражает

данное свойство науки. Каждая глава, фокусируясь на отдельном аспекте компьютерных наук, часто оказывается на удивление независимой от остальных, тогда как все вместе эти главы образуют исчерпывающий обзор всей этой области знаний.

Способность к творчеству

Хотя компьютеры могут быть просто сложными машинами, механически выполняющими алгоритмические инструкции, вы увидите, что область компьютерных наук является, по сути, несомненно, творческой. Создание и применение новых алгоритмов является сугубо человеческой деятельностью, которая определяется нашим врожденным желанием применить свои инструменты для решения различных задач в окружающем мире. Компьютерные науки не только расширяют формы выражения, охватывающие изобразительное, литературное и музыкальное искусство, но также предоставляют новые способы их цифрового представления, пронизывающие весь современный мир.

Разработка крупной программной системы мало напоминает приготовление блюда по рецепту, найденному в поваренной книге, — гораздо больше она напоминает создание новой изысканной скульптурной группы. Обдумывание и компоновка ее общей структуры и функций требуют тщательного планирования. Создание ее компонентов требует времени, внимания к деталям и практических умений. Конечный продукт воплощает дизайнерскую эстетику и эмоциональные качества ее создателей.

Данные

Компьютеры способны представить любую информацию, которую можно представить дискретными значениями и оцифровать. Алгоритмы позволяют обрабатывать или преобразовывать такую представленную в цифровом виде информацию бесчисленным количеством способов. Результатом этих действий является не просто перенос цифровых данных из одной части компьютера в другую; компьютерные алгоритмы позволяют находить шаблоны, создавать имитации и сопоставлять связи такими способами, которые генерируют новые знания и понимание. Громадные объемы сохраняемых данных, высокоскоростные компьютерные сети и мощные вычислительные инструменты позволяют делать важнейшие открытия во многих других областях естественных наук, техники и гуманитарных наук. Независимо от того, предсказывается ли эффект от воздействия нового лекарственного препарата путем моделирования сложной объемной структуры белка, статистически анализируется эволюция языка на протяжении столетий на основе оцифрованных книг или визуализируется

трехмерные изображения внутренних органов при неинвазивном медицинском сканировании, именно данные ведут к современным открытиям по всему фронту исследовательской деятельности человека.

Вот некоторые из вопросов о данных, которые обсуждаются в этой книге.

- Как в компьютерах сохраняются данные обычных цифровых образов таких типов информации, как числа, текст, изображения, звук и видео?
- Как в компьютерах аппроксимируются данные различных аналоговых образов, полученных из реального мира?
- Как компьютеры обнаруживают и предотвращают появление ошибок в данных?
- Какими могут быть последствия от наличия постоянно растущей в объеме и взаимосвязанной цифровой совокупности данных, имеющихся в нашем распоряжении?

Программирование

Перевод намерений человека в такой алгоритм, который сможет выполнить машина, сейчас чаще всего называют *программированием*, хотя при изобилии языков и прочих инструментов, доступных в наше время, этот процесс сейчас мало напоминает программирование компьютеров, каким оно было в 1950- и ранние 1960-е годы. Хотя компьютерные науки включают намного больше, чем просто написание компьютерных программ, способность решать задачи посредством разработки выполнимых алгоритмов (программ) остается фундаментальным умением для каждого специалиста по компьютерным наукам.

Аппаратные средства компьютеров способны выполнять только относительно простые алгоритмические шаги, но абстракция, обеспечиваемая компьютерными языками программирования, позволяет человеку обдумывать, находить и кодировать решения гораздо более сложных проблем. Несколько основных вопросов, которые очерчивают рамки обсуждения этой темы в книге, можно сформулировать следующим образом.

- Как создаются программы?
- Ошибки каких типов могут иметь место в программах?
- Как ошибки в программах обнаруживаются и устраняются?
- Каким может быть эффект от наличия ошибок в современных программах?
- Как программы документируются и оцениваются?

Интернет

Глобальная сеть Интернет соединила между собой компьютеры и электронные устройства по всему миру, и это оказало громадное влияние на то, как современное технологическое общество хранит, извлекает информацию и делится ею. Коммерция, новости, развлечения и коммуникации теперь во все большей степени зависят от этой паутины соединений между компьютерными сетями меньших размеров. Наше обсуждение этой темы предусматривает не только рассмотрение методов построения и функционирования самой этой системы, но и анализ многих других аспектов жизни человеческого общества, которое теперь накрепко переплетено с глобальной сетью.

Доступность Интернета также оказала глубокое влияние на степень конфиденциальности и уровень защищенности личной информации. Киберпространство несет в себе множество опасностей. Соответственно, криптография и кибербезопасность имеют все возрастающую важность для нашего тесно связанного мира.

Влияние на общество

Компьютерные науки оказали глубокое влияние не только на технологии, которые мы используем для коммуникации, работы и развлечений; их бурное развитие имело огромные социальные последствия. Прогресс в компьютерных науках размывает множество ограничений, на основании которых наше общество принимало решения в прошлом, и ставит под сомнение многие из давних принципов общества. В правоведении возникают вопросы относительно того, в какой степени возможно владение программным обеспечением, а также относительно прав и обязанностей, накладываемых этим правом собственности. В этике люди сталкиваются со множеством аспектов, бросающих вызов традиционным принципам, на которых основано поведение человека. Что касается правительств, то здесь возникают вопросы относительно допустимой степени регулирования компьютерной технологии и ее применения. В философии этот прогресс породил противоречия между наличием интеллектуального поведения и наличием самого интеллекта. А во всем обществе он вызвал острые споры о том, что собой представляют новые приложения: новые свободы или новые средства контроля.

Такие темы очень важны для тех, кто выбрал для себя карьеру в компьютерных науках или связанных с ними областях. Открытия в науке иногда вызывают противоречивые применения, вызывая серьезное недовольство в отношении сделавших их исследователей. Более того, вместо успешной карьеры иной раз можно быстро сойти с рельс и оказаться на обочине из-за совершенной этической ошибки.

Способность справляться с дилеммами, создаваемыми продвижением компьютерных технологий, также важна и для тех, кто находится вне их непосредственной сферы. И действительно, эти технологии настолько быстро пронизывают общество, что лишь очень немногие, если таковые вообще имеются, способны остаться независимыми от их воздействия.

В этой книге вам будет предоставлено техническое обоснование, необходимое для рационального подхода к дилеммам, генерируемым компьютерными науками. Однако одних технических знаний в этой области науки будет недостаточно для решения всех возникающих вопросов. Принимая это во внимание, мы включили в материал книги несколько разделов, посвященных социальным, этическим и правовым аспектам влияния компьютерных наук на общество. В них обсуждаются, например, проблемы безопасности, вопросы права собственности и ответственности за программное обеспечение, социальное воздействие технологии баз данных и последствия достижений в области искусственного интеллекта.

Более того, часто не существует окончательного правильного решения проблемы, а многие имеющие силу решения являются компромиссами между противоположными (и возможно, одинаково правомерными) представлениями. Поиск решений в подобных случаях часто требует умения выслушивать и признавать иные точки зрения, проводить рациональное обсуждение и менять собственное мнение по мере приобретения новых знаний. По этой причине в конце каждой главы этой книги приводится подборка вопросов, объединенных под заголовком "Общественные и социальные вопросы", в которых исследуется связь между компьютерными науками и обществом. Эти вопросы не предполагают немедленных и однозначных ответов, они предназначены, прежде всего, для обдумывания и обсуждения. Во многих случаях ответ, который кажется очевидным на первый взгляд, перестанет вас удовлетворять после ознакомления с альтернативами. Если говорить коротко, назначение этих вопросов не в том, чтобы вы немедленно предоставили "правильный" ответ, а скорее в том, чтобы повысить вашу осведомленность, в том числе ваше понимание позиций различных заинтересованных сторон в той или иной проблеме, ваше понимание альтернатив и ваше понимание как краткосрочных, так и долгосрочных последствий принятия этих альтернатив.

Философы предоставили множество подходов к этическим аспектам в поисках фундаментальных теорий, которые позволяют вывести принципы принятия руководящих решений и правил поведения.

Этика характера, иначе называемая этикой добродетели, была разработана Платоном и Аристотелем, утверждавшими, что "хорошее поведение" не является результатом применения идентифицируемых правил, но является лишь естественным следствием "хорошего характера". В противоположность этому другие этические учения, такие как этика последствий, этика служебная, и

этика соблюдения обязательств, предполагают, что человек для решения этической дилеммы должен дать ответ на вопрос "Какими будут последствия?", "Каков мой служебный долг?", "Какие у меня обязательства?" соответственно, тогда как этика добродетели предлагает решать дилеммы поиском ответа на вопрос "Кем я хотел бы стать?" Следовательно, хорошее поведение достигается посредством развития хорошего характера, что обычно является результатом здорового воспитания и развития добродетельных привычек.

Именно этика добродетели положена в основу того подхода, который обычно принимается при "обучении" этике профессионалов в различных областях. Вместо представления определенных этических теорий, этот подход заключается в рассмотрении конкретных примеров, в которых разбирается определенное множество этических вопросов в той области, в которой эти лица являются экспертами. В результате обсуждения всех конкретных "за" и "против" в этих примерах, обучающиеся становятся более осведомленными, проницательными и восприимчивыми к опасностям, которые могут скрываться в их профессиональной жизни, и, таким образом, укрепляют свой характер. Это как раз тот стиль, в котором представлены вопросы в отношении социальных аспектов компьютерных наук в конце каждой главы этой книги.

СОЦИАЛЬНЫЕ И ОБЩЕСТВЕННЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники. Задача не сводится к тому, чтобы просто дать ответ на предложенные вопросы. Вы должны также понять, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются друг с другом.

1. Предположение, что нынешнее общество *отличается* от того общества, которое могло бы быть без компьютерной революции, как правило, принимается большинством. Является ли наше общество *лучше* или хуже некомпьютеризованного? Могли бы вы дать иной ответ на этот вопрос, если бы занимали другое положение в обществе?
2. Возможно ли участие в жизни современного техногенного общества без понимания основ этой технологии? Например, обязаны ли члены демократического государства, чье решение на выборах часто определяет, как технология будет поддерживаться и использоваться, осмыслить эту технологию? Зависит ли ваш ответ от того, какая именно технология имеется в виду? Например, будет ли ваш ответ на вопрос о ядерной технологии таким же, как и при рассмотрении компьютерной технологии?

3. Используя наличные деньги, люди обычно имеют право проводить финансовые операции без дополнительной платы за обслуживание. Однако в процессе автоматизации экономики финансовые учреждения сделали платными услуги за доступ к своим автоматизированным системам. Верно ли утверждение, что эти доплаты несправедливо ограничивают доступ частных лиц к экономике? Предположим, что работодатель платит своим работникам только чеками, а все финансовые учреждения ввели оплату за выдачу наличных денег по чеку или за помещение их на счет. Справедливо ли это по отношению к работникам? А как быть в том случае, если работодатель настаивает на оплате труда только посредством помещения суммы на счет в банке?
4. Если говорить об интерактивном телевидении, то в каких пределах можно разрешить компании получать от детей информацию относительно состояния домашних дел (возможно, через интерактивные игры). Например, можно ли разрешать компаниям получать от ребенка информацию о покупках, сделанных им или его родителями? А как насчет информации о самом ребенке?
5. В какой мере правительственные органы должны регулировать развитие компьютерных технологий и их применение? Например, можно еще раз вернуться к проблемам, упомянутым в вопросах 3 и 4. Чем можно оправдать введение правительственного регулирования?
6. В какой степени на наших внуков повлияют решения, принятые нами относительно технологии в целом и компьютерной технологии в частности?
7. По мере развития технологии наша образовательная система вынуждена постоянно пересматривать уровень абстракции, установленный в отношении подачи отдельных тем. Многие вопросы касаются того, действительно ли необходимо какое-либо умение или можно позволить студентам пользоваться только абстрактным инструментом? Студентов, изучающих тригонометрию, уже не учат определять значения тригонометрических функций с помощью таблиц. Вместо этого они используют калькуляторы как абстрактное средство определения этих значений. Некоторые утверждают, что письменное деление чисел в столбик также обеспечивает определенную абстракцию. Какие еще есть дисциплины, характеризующиеся подобными разногласиями? Может ли наличие автоматической проверки правописания устранить необходимость в грамотном письме? Может ли случиться так, что с использованием видеотехники когда-нибудь отпадет потребность в чтении?
8. Предполагается, что концепция публичных библиотек построена в основном на утверждении, что все граждане демократического общества

должны иметь доступ к информации. Так как все больше информации сохраняется и распространяется с помощью компьютерных технологий, является ли доступ к таким технологиям правом каждого человека? Если это так, то должны ли публичные библиотеки стать средством, обеспечивающим этот доступ?

9. Какие вопросы этического характера возникают в обществе, которое полагается на использование абстрактных инструментов? Есть ли такие случаи, когда использование товара или услуги без знания того, как это осуществляется, является неэтичным? А без знания того, как это производится? Или без понимания побочных последствий использования этого товара или услуги?
10. По мере того, как наше общество становится все более и более автоматизированным, правительствам становится проще следить за деятельностью граждан. Как вы считаете, хорошо это или плохо?
11. Какие технологии, которые Джордж Оруелл описал в своем романе-антиутопии *1984*, теперь стали реальными? Используются ли они для тех же целей и тем же способом, как это предсказано в романе Оруэллом?
12. Если бы у вас была машина времени, в какой период истории вы хотели бы жить? Какие современные технологии вы хотели бы взять с собой? Можно было бы выбранные вами технологии взять с собой, не взяв прочие? До какой степени одну технологию можно отделить от другой? Согласуется ли протест против глобального потепления с безусловным принятием современного уровня медицинской помощи?
13. Предположим, что работа требует от вас проживания в иной культуре. Должны ли вы сохранить верность этическим нормам своей культуры или принять иную этику той культуры, в которой живете? Будет ли ваш ответ зависть от того, с чем именно связана проблема, скажем, со стилем одежды или с правами человека? Какие этические стандарты должны превалировать, если вы продолжаете жить в своей родной культуре, но ведете свой бизнес через Интернет в странах с иной культурой?
14. Стало ли общество более зависимым от компьютерных приложений в области коммерции, коммуникаций или социального общения? Например, какими могут быть последствия от долговременного прерывания работы Интернета и/или сотовой телефонной связи?
15. Большинство смартфонов способны определить свое местоположение посредством GPS. Это позволяет приложениям предоставлять вам информацию, связанную с вашим текущим местоположением (например, новости, местная погода или наличие учреждений, предприятий или организаций поблизости). Однако подобные возможности GPS могут также

позволить другим приложениям предоставлять сведения о местонахождении смартфона любым другим заинтересованным сторонам. Хорошо это или плохо? Как можно злоупотреблять информацией о местонахождении смартфона (а значит, и о вашем местоположении)?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Goldstine J.J. *The Computer from Pascal to von Neumann.* — Princeton: Princeton University Press, 1972.
2. Haigh T., Priestley M., Rope C. *ENIAC in Action: Making and Remaking the Modern Computer.* — Cambridge, MA: The MIT Press, 2016.
3. Kizza J.M. *Ethical and Social Issues in the Information Age*, 3rd ed. — London: Springer-Verlag, 2007.
4. Mollenhoff C.R. *Atanasoff: Forgotten Father of the Computer.* — Ames, IA: Iowa State University Press, 1988.
5. Neumann P.G. *Computer Related Risks.* — Boston, MA: Addison-Wesley, 1995.
6. Ni L. *Smart Phone and Next Generation Mobile Computing.* — San Francisco: Morgan Kaufmann, 2006.
7. Quinn M.J. *Ethics for the Information Age*, 5th ed. — Boston, MA: AddisonWesley, 2012.
8. Randell B. *The Origins of Digital Computers*, 3rd ed. — New York: SpringerVerlag, 1982.
9. Spinello R.A., Tavani H.T. *Readings in CyberEthics*, 2nd ed. — Sudbury, MA: Jones and Bartlett, 2004.
10. Swade D. *The Difference Engine.* — New York: Viking, 2000.
11. Tavani H.T. *Ethics and Technology: Ethical Issues in an Age of Information and Communication Technology*, 4th ed. — New York: Wiley, 2012.
12. Woolley B. *The Bride of Science: Romance, Reason, and Byron's Daughter.* — New York: McGraw-Hill, 1999.

Глава 1

В этой главе будут рассмотрены темы, связанные с представлением и хранением данных в компьютере. Рассматриваемые типы данных включают текст, числовые значения, изображения, звук и видео. Немало информации, предоставляемой в этой главе, также связано с областями науки и техники, выходящими за пределы традиционного представления о компьютерных науках, таких как цифровая фотография, запись и воспроизведение аудио и видео, а также коммуникации на больших расстояниях.

Хранение данных

1.1. БИТЫ И ИХ ХРАНЕНИЕ
- Булевы операции
- Вентили и триггеры
- Шестнадцатеричная система счисления

1.2. ОСНОВНАЯ ПАМЯТЬ
- Организация основной памяти
- Измерение емкости памяти

1.3. МАССОВАЯ ПАМЯТЬ
- Магнитные системы
- Оптические системы
- Флеш-накопители

1.4. ПРЕДСТАВЛЕНИЕ ИНФОРМАЦИИ В ВИДЕ КОМБИНАЦИИ ДВОИЧНЫХ РАЗРЯДОВ
- Представление текста
- Представление числовых значений
- Представление изображений
- Представление звука

*1.5. ДВОИЧНАЯ СИСТЕМА СЧИСЛЕНИЯ
- Двоичная нотация
- Двоичное сложение
- Представление дробей в двоичных кодах

*1.6. ПРЕДСТАВЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ
- Двоичный дополнительный код
- Сложение чисел в двоичном дополнительном коде
- Проблема переполнения
- Двоичная нотация с избытком

*1.7. ПРЕДСТАВЛЕНИЕ ДРОБНЫХ ЗНАЧЕНИЙ
- Двоичная нотация с плавающей точкой
- Ошибки усечения

*1.8. ДАННЫЕ И ПРОГРАММИРОВАНИЕ
- Знакомимся с языком Python
- Программа Hello Python
- Переменные
- Операторы и выражения
- Программа конвертирования валюты
- Отладка

*1.9. СЖАТИЕ ДАННЫХ
- Универсальные методы сжатия данных
- Сжатие изображений
- Сжатие аудио и видео

*1.10. ОШИБКИ ПРИ ПЕРЕДАЧЕ ИНФОРМАЦИИ
- Биты четности
- Коды с исправлением ошибок

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

Человечество использует компьютеры для создания программного обеспечения, цифровых медиаданных, веб-контента, наборов данных различного назначения, конструкторских и имитационных моделей — все это примеры компьютерных артефактов, которые люди создают с использованием вычислительных средств и абстракции. Изучение компьютерных наук мы начнем с рассмотрения, как эти компьютерные артефакты кодируются и сохраняются внутри компьютеров. Прежде всего мы обсудим принципы построения устройств хранения данных компьютеров, а потом познакомимся с тем, как информация кодируется для сохранения в этих системах. Сама сложность этих систем требует использования мощных абстракций на многих уровнях — от простейших аппаратных вентилей, построенных из электронных переключателей, до программных языков самого высокого уровня, которые используются для кодирования требуемой последовательности выполняемых компьютером команд, предоставляемой ему в виде программного обеспечения.



Основные положения для запоминания

- **Компьютерный артефакт — это что-либо, созданное человеком с использованием компьютера, что может быть программой, изображением, аудио- или видеороликом, презентацией, файлом веб-страницы или чем-то иным.**

1.1. Биты и их хранение

Абстракция различных уровней позволяет нам представлять множество типов данных, но на самом нижнем ее уровне в современных компьютерах вся информация кодируется как комбинация нулей и единиц. Эти цифры называют **битами** (от англ. *binary digits* — двоичные цифры). Хотя у вас может иметь место склонность связывать биты с указанными числовыми значениями, на самом деле они представляют собой просто символы, смысл которых зависит от конкретного приложения. Иногда последовательность битов используется для представления числовых значений, а в других случаях она может представлять букву или знак пунктуации; с тем же успехом последовательность битов может определять цвет элемента изображения либо громкость или высоту звука.



Основные положения для запоминания

- Цифровые данные представляются с использованием абстракции различных уровней.
- На самом нижнем уровне абстракции все цифровые данные представляются с помощью битов.
- На более высоких уровнях абстракции биты группируют для представления чисел, символов, цвета и т.д.
- На одном из самых низких уровней абстракции цифровые данные могут быть представлены числами в двоичной системе счисления (системе счисления с основанием 2), в которой любые числа представляются как комбинация нулей и единиц.

Булевы операции

Чтобы лучше понять, как отдельные биты сохраняются и как ими манипулируют в компьютере, будет удобно представить, что бит `0` представляет значение *ложь*, а бит `1` представляет значение *истина*. В математике операции, которые манипулируют значениями "истина/ложь", называют логическими или **булевыми операциями**, — последнее название присвоено в память о математике Джордже Буле (George Boole, 1815–1864), который положил начало области математики, получившей название *математической логики*. Рассмотрим три булевы операции, `AND` (И), `OR` (ИЛИ) и `XOR` (исключающее ИЛИ), принципы выполнения которых показаны на рис. 1.1. (Здесь и далее названия булевых операций мы будем записывать прописными буквами.) Эти операции подобны арифметическим операциям умножения и сложения, поскольку соответствующим образом комбинируют пару величин (входные данные операции) в целях создания третьей величины (выходные данные операции). Однако в отличие от арифметических операций, логические операции `AND`, `OR` и `XOR` манипулируют значениями "истина/ложь", а не числовыми значениями.

Булева операция `AND` (И) была разработана для отражения истинности или ложности высказываний, образованных в результате соединения двух меньших высказываний с помощью союза "и". В общем виде такие высказывания можно представить следующим образом:

```
P AND Q
```

Здесь `P` представляет одно высказывание, а `Q` — другое. Давайте рассмотрим это на таком примере:

Кермит — лягушка `AND` *Мисс Пигги — актриса*

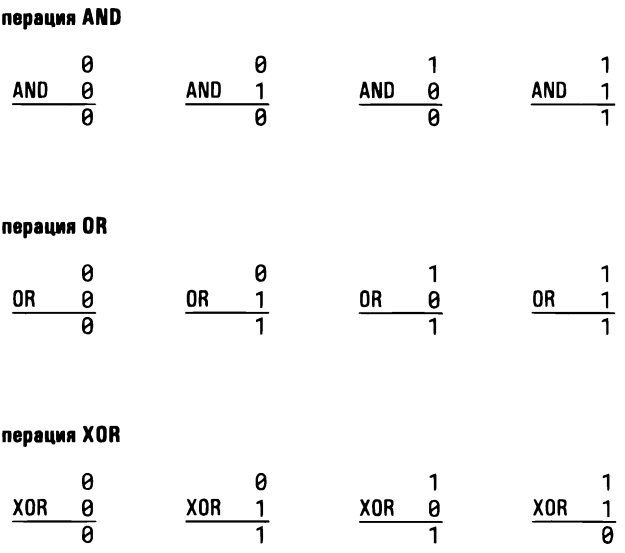


Рис. 1.1. Возможные входные и выходные значения булевых операций AND, OR и XOR

Входные данные для операции AND представляют истинность или ложность компонентов, составляющих высказывание, а результат — истинность или ложность самого высказывания. Поскольку высказывание, представленное в виде `P AND Q`, является истинным только тогда, когда оба его компонента истинны, мы можем сделать заключение, что выражение `1 AND 1` должно иметь значение `1`, тогда как во всех остальных случаях его результатом будет значение `0`, что и показано на рис. 1.1.

Аналогичным образом операция OR (ИЛИ) основана на сложном высказывании, имеющем следующий вид:

```
P OR Q
```

Здесь также операнд `P` представляет одно выражение, а операнд `Q` — другое. Такие высказывания являются истинными в том случае, если хотя бы один из их компонентов истинен, как изображено на рис. 1.1.

В английском языке нет отдельного союза, выражающего смысл операции XOR (исключающее ИЛИ). Операция XOR дает в результате значение `1` (истина) только тогда, когда один из входных операндов имеет значение `1` (истина), а другой — значение `0` (ложь). Таким образом, утверждение, построенное по схеме `P XOR Q`, имеет следующий смысл: "либо `P`, либо `Q`, но не оба одновременно".

Следующая булева операция — NOT (НЕ). Она отличается от операций AND, OR и XOR тем, что имеет только одно входное значение. Результат этой опера-

ции имеет значение, противоположное входному значению. Иначе говоря, если входным значением операции NOT является истина, то результат будет иметь значение "ложь", и наоборот. Например, если входное значение операции NOT представляет истинность или ложность высказывания

Фоззи — медведь,

то результат выполнения этой операции будет представлять истинность или ложность противоположного высказывания:

Фоззи — не медведь

Вентили и триггеры

Устройство, которое выдает результат булевой операции после введения в него входных данных, называется **вентилем**. Существуют различные технологии конструирования вентилей, например с использованием зубчатых колес, реле или оптических устройств. Вентили, встроенные в современный компьютер, — это небольшие электронные схемы, в которых цифры 0 и 1 представляются разными уровнями электрического напряжения. Однако нам совсем необязательно подробно обсуждать эту тему. Вполне достаточно представить вентили в их символической форме, как это показано на рис. 1.2. Обратите внимание, что вентили AND, OR, XOR и NOT изображаются в виде различных схематических элементов, у которых входные данные поступают с одной стороны, а выходной сигнал считывается с другой стороны.

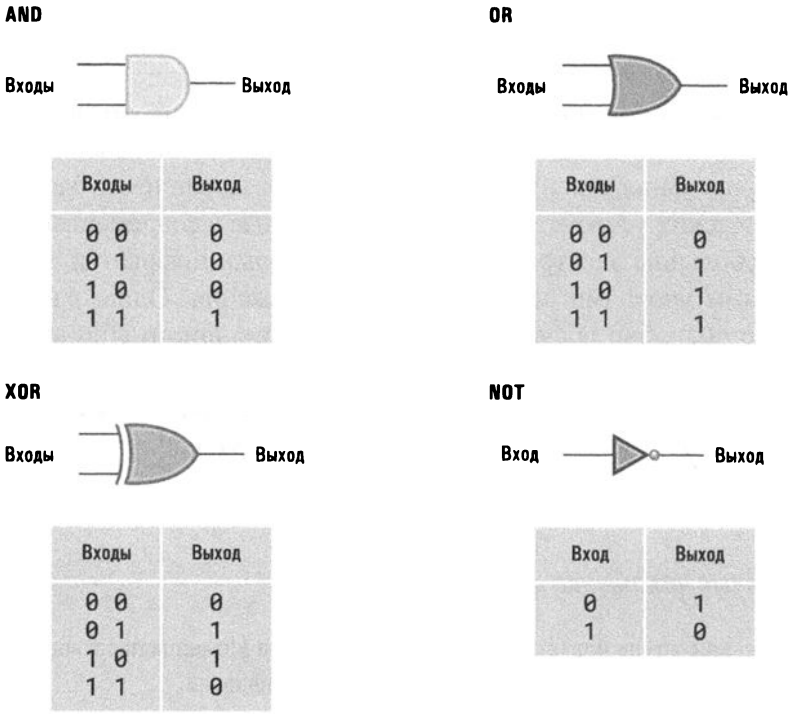


Основные положения для запоминания

- **Логические вентили представляют собой абстракцию уровня аппаратного обеспечения, используемую для моделирования булевых операций.**

Вентили, подобные показанным на рис. 1.2, представляют собой строительные блоки, из которых конструируются компьютеры. Как результат булева логика и операторы являются фундаментальными операциями в существующих языках программирования. Один важный этап этого направления представлен в электрической схеме, показанной на рис. 1.3. Это один из возможных вариантов схем определенного класса, называемых **триггерами**. Триггер является фундаментальным элементом построения компьютерной памяти. Это схема, которая постоянно выдает выходное значение 0 или 1; которое не меняется до тех пор, пока одиночный импульс (например, временное изменение значения

от 0 до 1 с последующим возвращением к нулю) от другой схемы не переведет ее в противоположное состояние. Другими словами, выходное значение будет "запомнено" и может быть переведено из одного состояния в другое только под воздействием внешних стимулов. Пока оба входных значения в схеме, представленной на рис. 1.3, равны нулю, выходное значение (0 или 1) будет неизменным. Однако даже кратковременное появление значения 1 на верхнем входе схемы вызовет установку на ее выходе значения 1, тогда как кратковременное появление значения 1 на нижнем входе вызовет установку на выходе значения 0.



Входы	Выход
0 0	0
0 1	0
1 0	0
1 1	1

Входы	Выход
0 0	0
0 1	1
1 0	1
1 1	1

Входы	Выход
0 0	0
0 1	1
1 0	1
1 1	0

Вход	Выход
0	1
1	0

Рис. 1.2. Схематическое представление вентилей AND, OR, XOR и NOT, а также таблицы их входных и выходных данных

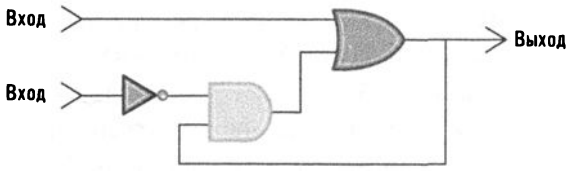


Рис. 1.3. Схема простого триггера

Основные положения для запоминания

- **Логические концепции и булева алгебра являются фундаментальными основами программирования.**

Теперь рассмотрим предыдущее утверждение более подробно. Мы не знаем текущего выходного значения схемы, представленной на рис. 1.3, поэтому предположим, что на верхний вход поступило значение `1`, тогда как на нижнем входе сохраняется значение `0` (рис. 1.4, *а*). Это приведет к тому, что выходное значение вентиля `OR` станет равным `1`, независимо от текущего значения на его втором входе. В свою очередь, на обоих входах вентиля `AND` теперь будут значения `1`, поскольку на другом его входе уже присутствует значение `1` (оно появляется за счет передачи значения `0` на нижнем входе триггера через вентиль `NOT`). В результате выходное значение вентиля `AND` станет равным `1`, а это значит, что на втором входе вентиля `OR` также появится значение `1` (рис. 1.4, *б*). Это гарантирует, что выходное значение вентиля `OR` останется равным `1` даже в том случае, если значение на верхнем входе триггера вновь станет равным `0` (рис. 1.4, *в*). Таким образом, выходное значение триггера теперь равно `1` и будет сохраняться таким даже в том случае, если на верхний вход будет вновь подано значение `0`.

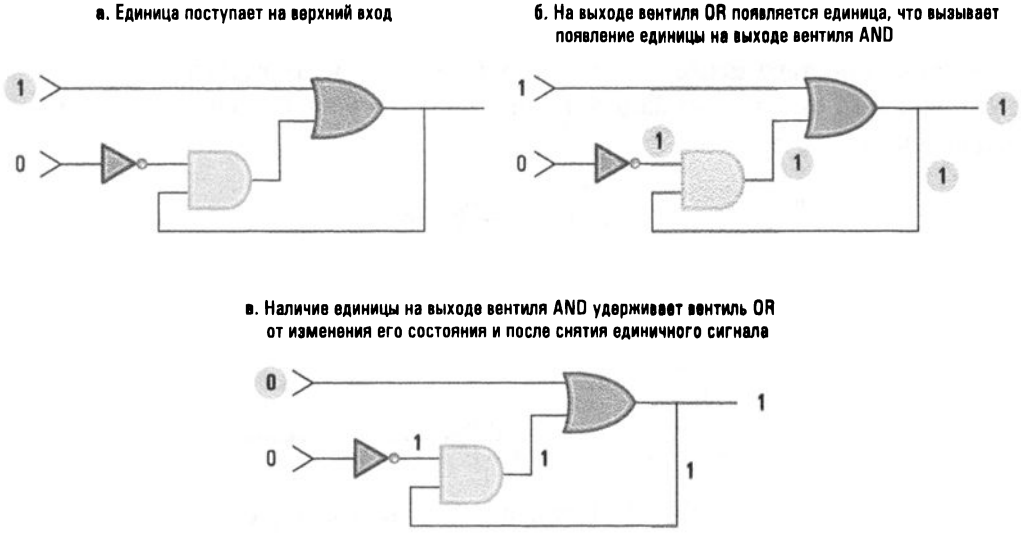


Рис. 1.4. Установка выходного значения триггера равным 1

Точно так же временное появление значения 1 на нижнем входе триггера приведет к тому, что на его выходе установится значение 0, которое будет оставаться неизменным даже после того, как на нижний вход вновь будет подано значение 0.

Наше подробное обсуждение работы триггерной схемы, представленной на рис. 1.3 и 1.4, преследовало три цели. Во-первых, продемонстрировать, как схемы компьютера могут быть сконструированы из вентилей, — этот процесс называется разработкой цифровых схем, что является важной темой в компьютерной инженерии. В действительности триггер — это только один из многих типов электронных схем, которые являются базовыми элементами при конструировании компьютеров.

Во-вторых, рассмотренная концепция триггера являет собой яркий пример абстракции и использования абстрактных инструментов. В действительности существует много разных способов построения триггеров. Одно из альтернативных решений представлено на рис. 1.5. Если вы поэкспериментируете с этой схемой, то обнаружите, что, хотя у нее совсем иная внутренняя структура, ее внешние свойства полностью аналогичны схеме, представленной на рис. 1.3. Инженеру, разрабатывающему компьютер, нет необходимости знать, какой именно тип схемы использован для построения каждого конкретного триггера. Ему вполне достаточно понимания внешних характеристик поведения триггеров, чтобы пользоваться ими как абстрактными инструментами. В этом случае триггер и другие строго определенные схемы в совокупности образуют полный набор строительных блоков, из которых инженер может конструировать более сложные элементы компьютера. В свою очередь, процесс разработки электронных схем компьютера также имеет иерархическую структуру, на каждом уровне которой компоненты, созданные на предыдущем уровне, используются как абстрактные инструменты.

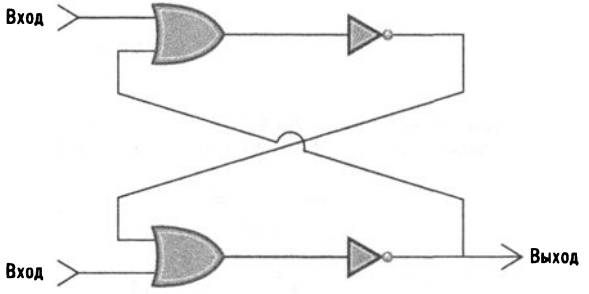


Рис. 1.5. Другой способ построения триггера

В-третьих, еще одна цель нашего детального знакомства с триггером состояла в том, что в действительности это одно из устройств, используемых для хранения значения бита в современных компьютерах. И это понятно, ведь мы

уже знаем, что выходное значение триггера можно установить так, чтобы он хранил требуемое значение бита, `0` или `1`. Тогда другие схемы при необходимости легко смогут изменять это значение, отсылая импульсы на входы триггера. Аналогичным образом другие схемы смогут реагировать на хранимое в триггере значение посредством использования его выходного значения как одного из своих входных значений. В результате множество триггеров, сконструированных как небольшие электронные схемы, можно использовать в компьютере как средство записи информации, которая закодирована в виде последовательности нулей и единиц. И это действительно так: существует технология проектирования *сверхбольших интегральных схем* (very large-scale integration — *VLSI*), позволяющая разместить миллионы электронных компонентов на полупроводниковой пластине (или *чипе*) и широко применяемая для создания миниатюрных устройств, содержащих миллионы триггеров вместе с соответствующей управляющей схемой. Далее эти чипы используются как абстрактные инструменты при конструировании компонентов компьютерных систем более высокого уровня. В действительности данная технология часто используется для создания *всей* компьютерной системы на единственном чипе.



Основные положения для запоминания

- **Двоичные данные обрабатываются на физическом уровне компьютерного оборудования, включающем вентили, чипы и электронные компоненты.**
- **Аппаратное обеспечение компьютеров строится с использованием нескольких уровней абстракции, таких как транзисторы, логические вентили, чипы, материнские платы, специализированные карты и устройства хранения (память).**

Шестнадцатеричная система счисления

При обсуждении внутренних процессов компьютера нам придется иметь дело со строками битов, некоторые из которых могут оказаться достаточно длинными. Длинные строки битов принято называть **потоками** (stream). К сожалению, человек с трудом оперирует подобными величинами. Даже простое воспроизведение комбинации 101101010011 кажется утомительной задачей и подвержено ошибкам. Поэтому для упрощения представления комбинаций двоичных разрядов обычно используется упрощенная система записи, которая называется **шестнадцатеричной нотацией**, поскольку построена на шестнадцатеричной системе счисления. Особенность этой нотации состоит в том, что двоичные комбинации в машине обычно представляются группами из двоичных разрядов, длина которых кратна четырем. Это означает, что, поскольку в

шестнадцатеричной нотации каждый символ используется для представления четырех битов, строку из двенадцати битов можно представить всего лишь тремя шестнадцатеричными символами. В математике для указания используемой системы счисления принято пользоваться нижним индексом, поэтому шестнадцатеричное значения для числа 15_{10} может быть представлено как F_{16}. Специалисты-компьютерщики, привыкшие пользоваться текстовыми языками программирования, не поддерживающими нотации в нижнем индексе, чаще всего для указания, что данное число не является десятичным, используют префиксы. В этой книге, чтобы исключить возможность разночтения, перед шестнадцатеричными числами мы всегда будем ставить префикс "`0x`".



Основные положения для запоминания

- **Шестнадцатеричная система счисления (с основанием 16) широко используется для представления цифровых данных, поскольку шестнадцатеричное представление числа содержит гораздо меньше цифр, чем двоичное.**

На рис. 1.6 изображена схема шестнадцатеричной системы кодирования. В левом столбце показаны все возможные комбинации битов для строк длиной четыре двоичных разряда, а в правом приведен соответствующий символ, используемый в шестнадцатеричной системе счисления для представления комбинации битов из левого столбца. Согласно этой системе кодирования двоичный код `10110101` может быть представлен как `0xB5`. Этот результат был получен путем разделения исходной двоичной комбинации на подгруппы длиной четыре бита и замены каждой такой группы ее шестнадцатеричным эквивалентом. Комбинация `1011` представлена буквой `0xB`, а комбинация `0101` — цифрой `0x5`. Таким же способом 16-битовая строка `1010010011001000` может быть представлена в более удобной форме: `0xA4C8`. Шестнадцатеричная нотация будет широко использоваться в следующей главе. Именно тогда у вас появится возможность оценить ее эффективность.

Комбинация битов	Шестнадцатеричное представление
0000	0x0
0001	0x1
0010	0x2
0011	0x3
0100	0x4
0101	0x5
0110	0x6
0111	0x7
1000	0x8
1001	0x9
1010	0xA
1011	0xB
1100	0xC
1101	0xD
1110	0xE
1111	0xF

Рис. 1.6. Шестнадцатеричная система кодирования

1.1. Вопросы и упражнения

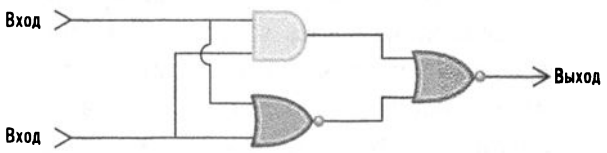
1. При каких значениях на входах представленной ниже схемы на ее выходе появится значение 1?



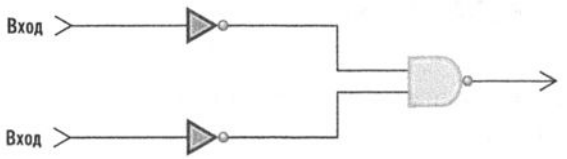
2. Выше утверждалось, что при поступлении значения 1 на нижний вход триггера, показанного на рис. 1.3 (с сохранением при этом значения 0 на верхнем его входе), на его выходе установится значение 0. Опишите последовательность событий, происходящих в этом случае в элементах триггера.

3. Предположим, что на оба входа триггера, показанного на рис 1.5, подается значение 0. Опишите последовательность событий, которые будут происходить в элементах этого триггера при кратковременном поступлении на его верхний вход значения 1.

4. **а.** Если выход вентиля AND подается на вход вентиля NOT, то такая комбинация вентилей вычисляет результат булевой операции, называемой NAND, и на ее выходе значение 0 появится только тогда, когда на оба входа поступают значения 1. Графическое обозначение вентиля NAND точно такое же, как и вентиля AND, за исключением того, что в начало выходной стрелки помещается кружок. Ниже представлена схема, содержащая вентили NAND. Какую булеву операцию реализует эта схема?



б. Если выход вентиля OR подается на вход вентиля NOT, то такая комбинация вентилей вычисляет результат булевой операции, называемой NOR, и на ее выходе значение 1 появится только тогда, когда на оба входа поступят значения 0. Графическое обозначение вентиля NOR точно такое же, как и вентиля OR, за исключением того, что в начало выходной стрелки помещается кружок. Ниже представлена схема, содержащая вентиль AND и два вентиля NOR. Какую булеву операцию реализует эта схема?



5. Используйте шестнадцатеричную систему счисления для представления следующих комбинаций двоичных разрядов.

   а. `0110101011110010` **б.** `111010000101010100010111`

   в. `01001000`

6. Какие комбинации двоичных разрядов представлены следующими шестнадцатеричными кодами?

   а. `0x5FD97` **б.** `0x610A` **в.** `0xABCD` **г.** `0x0100`

1.2. Основная память

В компьютере для хранения данных используется большой набор схем (таких, как триггеры), каждая из которых способна запомнить один двоичный разряд (бит). Это хранилище битов принято называть **основной** (или *оперативной*) **памятью**.

Организация основной памяти

Запоминающие схемы основной памяти машины организованы в небольшие блоки (доступные как единое целое), которые называются **ячейками памяти**, при этом размер ячейки памяти обычно составляет 8 бит. (Строки из 8 бит называют **байтами**. Следовательно, типичная ячейка памяти имеет емкость 1 байт.) Небольшие компьютеры, встроенные в такие бытовые приборы, как холодильники или микроволновые печи, могут иметь основную память, состоящую всего из нескольких сотен ячеек, тогда как персональные компьютеры или смартфоны могут иметь основную память, исчисляемую миллиардами ячеек.

Биты в ячейке памяти можно представить себе размещенными в один ряд. Один конец этого ряда называется **старшим**, а другой — **младшим**. Несмотря на то что в памяти машины нет ни правой, ни левой стороны, в нашем представлении биты всегда выстроены в ряд слева направо, причем старший конец располагается слева. Бит, находящийся на этом конце, обычно называют **старшим** или битом с наибольшим весом, а бит на другом конце, соответственно,

именуют **младшим** или битом с наименьшим весом. Таким образом, содержимое ячейки памяти размером 1 байт можно представить себе так, как показано на рис. 1.7.

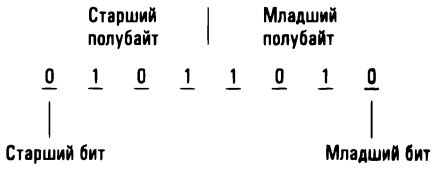


Рис. 1.7. Организация ячейки памяти размером 1 байт

Для идентификации отдельных ячеек основной памяти машины каждой ячейке присваивается уникальное "имя", называемое **адресом**. Эта система аналогична методу, используемому для поиска здания в городе по указанному адресу. Однако в случае с ячейками памяти применяются исключительно цифровые адреса. Точнее говоря, можно просто представить себе все эти ячейки помещенными в один ряд и пронумерованными в восходящем порядке начиная с нуля, т.е. адреса ячеек в памяти машины будут представлены числами 0, 1, 2, ... Следует отметить, что такая система адресации не только позволяет однозначно идентифицировать каждую ячейку памяти (рис. 1.8), но и упорядочивает их, делая правомочными такие выражения, как "следующая ячейка" или "предыдущая ячейка".

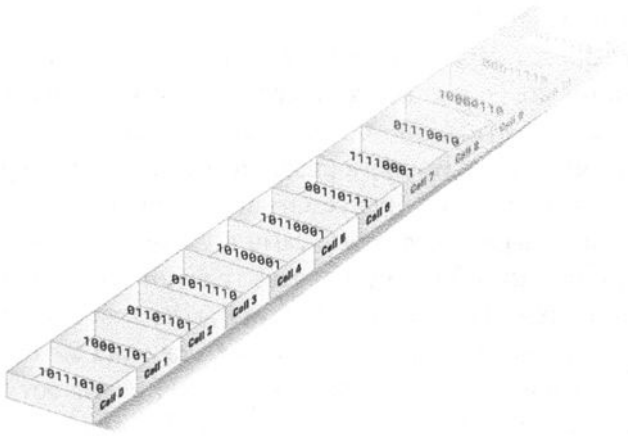


Рис. 1.8. Образное представление ячеек памяти, упорядоченных по адресам

Важным следствием упорядоченности ячеек в основной памяти и отдельных битов в пределах каждой такой ячейки является то, что вся совокупность битов памяти машины, в сущности, располагается в один длинный ряд.

Следовательно, отдельные части этого длинного ряда могут использоваться для хранения комбинаций двоичных разрядов, длина которых будет больше длины отдельной ячейки. В частности, если память разделена на ячейки размером 1 байт, то для сохранения строки из 16 бит можно просто воспользоваться двумя последовательными ячейками памяти.

В состав основной памяти машины, помимо электрической схемы, фиксирующей значения битов, входит и другая схема, позволяющая остальным компонентам машины записывать данные в ячейки памяти и извлекать их оттуда. Благодаря этому другие схемы могут считывать информацию из памяти посредством электронного запроса на извлечение содержимого ячейки с определенным адресом (это действие называется *операцией считывания*) или записывать информацию в память, посылая запрос на помещение определенной комбинации двоичных разрядов в ячейку с указанным адресом (это действие называется *операцией записи*).

Поскольку основная память машины организована в виде небольших прямо адресуемых ячеек, это позволяет адресовать каждую ячейку памяти в отдельности, т.е. данные, помещенные в основную память, могут обрабатываться в произвольном порядке. Это поясняет, почему основную память машины часто называют **памятью с произвольной выборкой** (random access memory — *RAM*). Возможность произвольного доступа к небольшим блокам данных совершенно противоположна принципам работы с устройствами массовой памяти, которые будут обсуждаться в следующем разделе. В этих устройствах длинные строки битов приходится обрабатывать как единые блоки.

Хотя ранее мы представили триггеры как удобное средство для хранения битов, в большинстве современных компьютеров память типа RAM строится с использованием аналогичных, но более сложных технологий, предоставляющих бо́льшие возможности миниатюризации и сокращения времени ответа схем. Многие из этих технологий предусматривают хранение битов в виде крошечных электрических зарядов, которые сами по себе достаточно быстро рассеиваются. В результате подобные устройства требуют дополнительной управляющей схемы, называемой схемой регенерации, которая регулярно, много раз в секунду, подзаряжает заряды, представляющие значения битов. По этой причине память, изготовленную по данной технологии, называют *динамической памятью* и обозначают аббревиатурой **DRAM**, т.е. *Dynamic RAM*. Иногда также используется термин **SDRAM**, что является аббревиатурой от *Synchronous DRAM*, — для ссылок на память DRAM, в которой используются дополнительные методы с целью сокращения времени, необходимого для выборки содержимого из ячеек памяти.

Измерение емкости памяти

Как вы узнаете из следующей главы, удобнее всего разрабатывать такие системы памяти компьютеров, в которых общее количество ячеек является степенью числа 2. По этой причине размер основной памяти в раннем поколении компьютеров часто измерялся в единицах по 1024 (что есть 2^{10}) ячеек. Поскольку число 1024 довольно близко к 1000, компьютерное сообщество воспользовалось префиксом *кило* для ссылок на эту единицу измерения объема памяти компьютеров. Иначе говоря, термин *килобайт* (сокращенно — Кбайт) использовался для ссылок на блок в 1024 байт. В результате, если у машины было 4096 ячеек памяти, то говорили, что объем ее основной памяти равен 4 Кбайт (4096 = 4 × 1024). По мере того как размер основной памяти компьютеров увеличивался, соответствующая терминология расширялась с включением таких единиц, как *мегабайт* (Мбайт), *гигабайт* (Гбайт) и *терабайт* (Тбайт). К сожалению, подобное применение префиксов *кило-*, *мега-* и других представляет собой неправильное использование данной терминологии, поскольку эти префиксы уже давно и широко используются в других областях науки и техники применительно к единицам, которые представляют степени 10. Например, при измерении расстояний километр — это 1000 метров, а при измерении радиочастот мегагерц соответствует 1 000 000 герц. В конце 1990-х годов международная организация стандартов разработала специальную терминологию, соответствующую степеням двойки (*киби-* (kibi), *меби-* (mebi), *гиби-* (gibi) и *тебибайт* (tebibyte)) и обозначающую степени числа 2, а не степени числа 10. Тем не менее, хотя эти различные обозначения утверждены законом во многих уголках света, как обычная публика, так и многие специалисты-компьютерщики неохотно отказываются от более знакомого, хотя и двусмысленного термина "мегабайт". Поэтому будет нелишним еще раз предупредить вас. Следуя общепринятым правилам, можно считать, что термины кило-, мега- и другие обозначают степени 2, если используются применительно к компьютерам, и подразумевают степени 10 во всех остальных случаях.

1.2. Вопросы и упражнения

1. Если ячейка памяти с адресом 5 содержит число 8, то в чем состоит различие между записью числа 5 в ячейку с номером 6 и пересылкой содержимого ячейки с номером 5 в ячейку с номером 6?
2. Предположим, что требуется поменять местами значения, хранящиеся в ячейках памяти с номерами 2 и 3. Найдите ошибку в следующей последовательности действий.

Этап 1. Переместите содержимое ячейки с номером 2 в ячейку с номером 3.

Этап 2. Переместите содержимое ячейки с номером 3 в ячейку с номером 2.

3. Предложите последовательность действий, которая позволит корректно поменять местами содержимое указанных ячеек. При необходимости можете воспользоваться дополнительными ячейками.
4. Какое количество битов содержится в памяти компьютера, если ее размер равен 4 Кбайт?

1.3. Массовая память

В связи с невозможностью постоянного хранения данных в основной памяти и ограниченным ее объемом большинство компьютеров оборудуется устройствами дополнительной памяти, которые называются **массовой памятью**, или запоминающими устройствами большой емкости, в число которых входят магнитные диски, компакт-диски, DVD-диски, магнитные ленты, флеш-накопители и твердотельные накопители (все эти типы устройство будут коротко обсуждаться ниже). Преимущества таких устройств по сравнению с основной памятью компьютера состоят в долговременности хранения данных, большей емкости при меньшей стоимости и, в большинстве случаев, возможности извлечения носителя информации из машины в целях архивирования.

Основным недостатком магнитных и оптических устройств массовой памяти является то, что они обычно требуют механических перемещений носителя или устройства считывания. Поэтому им требуется существенно больше времени для записи и извлечения данных по сравнению с основной памятью машины, в которой все необходимые действия выполняются на уровне электрических сигналов. Более того, устройства хранения с перемещающимися элементами более подвержены механическим отказам, чем твердотельные системы. Тем не менее, хотя флеш-накопители и твердотельные диски не имеют перемещающихся частей, их скорость работы или долговечность в сравнении с основной памятью могут ограничивать иные соображения в отношении электронных схем этих устройств.

Основные положения для запоминания

- **Выбор типа носителя для хранения данных влияет как на методы, так и на стоимость манипулирования данными, которые на них записываются.**

Магнитные системы

Многие годы магнитные технологии доминировали в устройствах массовой памяти. На сегодня типичным примером таких устройств являются **магнитные диски** или **жесткие диски** (HDD), в которых для хранения данных используется тонкий вращающийся диск с магнитным покрытием (рис. 1.9). Головки чтения/записи размещаются над и/или под диском таким образом, что во время вращения диска каждая головка описывает над ним круг, называемый **дорожкой**. Перемещая головки чтения/записи над поверхностью диска, можно получить доступ к различным концентрическим дорожкам. Чаще всего дисковая система памяти состоит из нескольких дисков, смонтированных на общей оси и расположенных друг над другом. Между дисками оставляется пространство, достаточное для перемещения головок чтения/записи между пластинами. Все головки чтения/записи в этом случае двигаются как единое целое. При каждом перемещении головок становится доступной новая группа дорожек, которую принято называть **цилиндром**.

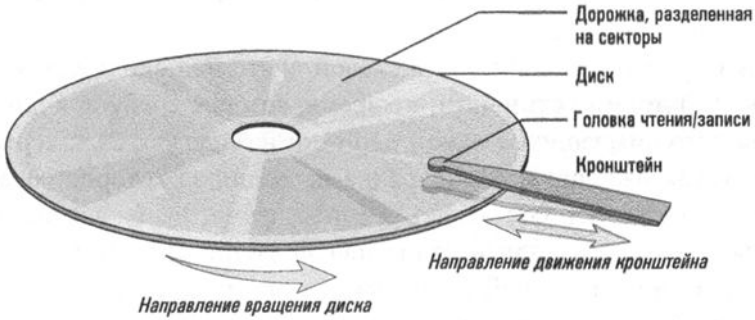


Рис. 1.9. Дисковое запоминающее устройство

Так как дорожка может содержать больше информации, чем обычно требуется одновременно обрабатывать, все дорожки разделены на зоны, или **сектора**, в которых информация записывается в виде непрерывной строки битов. Все секторы на диске содержат одинаковое количество битов (типичный размер сектора находится в диапазоне от 512 байт до нескольких килобайтов), и в простейших устройствах дисковой памяти каждая дорожка содержит одинаковое количество секторов. Это означает, что в секторах, которые находятся ближе к центру диска, биты данных размещаются более компактно по сравнению

с дорожками, расположенными ближе к внешнему краю, поскольку внешние дорожки длиннее внутренних. В противоположность этому в дисковых устройствах высокой емкости дорожки на внешнем крае могут содержать существенно больше секторов, чем те, которые находятся ближе к центру, — такая возможность реализуется за счет использования технологии, называемой **зонированной записью**. В этом случае несколько соседних дорожек в совокупности называют зоной, при этом типичное дисковое устройство может включать до десяти зон. Все дорожки в пределах зоны имеют одинаковое количество секторов, но в каждой из зон секторов на дорожке будет больше, чем у зоны, расположенной внутри нее. Подобным способом достигается более эффективное использование поверхности диска. Независимо от деталей реализации дисковые устройства массовой памяти всегда состоят из большого количества отдельных секторов, причем доступ к каждому из них может быть получен как к строке битов, независимой от прочих секторов.

Емкость дисковых устройств зависит от числа используемых в нем дисковых пластин, а также от плотности размещения дорожек и секторов на их поверхности. Дисковые системы малой емкости состоят из единственной пластины, особенно в тех случаях, когда физические размеры устройства должны быть максимально компактными. Дисковые устройства высокой емкости, способные вмещать многие терабайты информации, обычно состоят из нескольких (чаще всего — от трех до шести) пластин, смонтированных на одном шпинделе. Более того, данные в таких устройствах могут записываться на обеих, верхней и нижней, сторонах пластин.

Для оценки производительности дисковой системы используется несколько параметров: 1) **время установки**, т.е время, которое требуется для перемещения головки чтения/записи с одной дорожки на другую; 2) **задержка вращения**, или время ожидания, — половина времени, за которое совершается полный оборот диска, что составляет среднее время, необходимое для того, чтобы нужные данные появились под головкой чтения/записи после того, как она разместится над выбранной дорожкой; 3) **время доступа**, определяемое как сумма времени установки и времени ожидания; и наконец, 4) **скорость передачи данных** (скорость, с которой данные могут передаваться дисковому устройству или считываться с него). (Обратите внимание, что в случае зонированной записи количество данных, проходящих через головку чтения/записи за один оборот диска, для дорожек во внешних зонах будет больше, чем для дорожек во внутренних зонах, а следовательно, скорость передачи данных будет меняться в зависимости от того, какая часть диска используется в данный момент.) Другим важным показателем производительности, применимым как к устройствам массовой памяти, так и к другим системам коммуникации, являются: 1) **пропускная способность**, т.е. общее количество битов данных, кото-

рое может быть передано устройством за единицу времени, — скажем, 3 Мбит в секунду; и 2) **задержка** (общее количество времени, прошедшее между поступлением запроса на передачу данных и началом их поступления).

Фактором, ограничивающим время доступа и скорость передачи данных, является скорость вращения диска. Чтобы обеспечить высокую скорость вращения, головки чтения/записи в таких устройствах не касаются поверхности диска, а "плавают" над его поверхностью. При этом зазор между головкой и поверхностью диска настолько мал, что даже единственная частичка пыли может застрять между ними, повредив и то, и другое (явление, известное как *разрушение головки*). По этой причине дисковые устройства обычно размещают в корпусах, которые герметически запечатываются еще на заводе. При таком подходе дисковые устройства могут обеспечивать скорость вращения на уровне нескольких сотен оборотов в секунду и обеспечивать скорость передачи данных, измеряемую в мегабитах в секунду.

Поскольку дисковые устройства для своей работы требуют физических перемещений их элементов, они уступают в скорости работы по сравнению с электронными схемами. Время задержки в электронных схемах измеряется в наносекундах (миллиардных долях секунды) и даже меньше, тогда как время установки, задержка вращения и время доступа дисковых устройств измеряется в миллисекундах (тысячных долях секунды). В результате время, требуемое для получения информации с дискового устройства, может показаться просто бесконечным для электронных схем, ожидающих ее поступления.

Технологии магнитной записи, которые сейчас используются менее широко, включают **магнитную ленту**, когда информация записывается на магнитное покрытие тонкой пластиковой ленты, наматываемой на катушку, и устройства с **гибкими дисками**, когда единственная гибкая пластина с магнитным покрытием помещается в пластиковый корпус (дискету), который можно вставлять или вынимать из устройства считывания. Устройства чтения магнитных лент имеют исключительно большое время поиска информации (а значит, большую задержку доступа), как и их близкие родственники — аудио- и видеокассеты, использование которых затрудняется необходимостью продолжительной прямой и обратной перемотки. Тем не менее низкая стоимость и высокая емкость по-прежнему делают магнитную ленту подходящим средством для приложений, в которых данные предварительно считываются или записываются строго последовательно, как при архивировании данных в резервные копии. В настоящее время съемный характер дискет с гибкими дисками совершенно обесценивается их малой емкостью и низкой скоростью считывания данных в сравнении с жесткими дисками, однако в первые десятилетия компьютерной эры они играли очень важную роль — вплоть до появления флеш-накопителей с большей емкостью и повышенной надежностью.



Основные положения для запоминания

- **Пропускная способность системы является мерой скорости передачи данных — это количество данных (измеряемое в битах), которое может быть передано за фиксированный промежуток времени.**
- **Задержка системы представляет собой время, прошедшее между моментом поступления запроса на получение данных и началом их выдачи.**

Оптические системы

Другим классом устройств массовой памяти являются системы, использующие оптические технологии. В качестве примера можно привести **компакт-диск** (CD). Это диски диаметром 12 сантиметров (около 5 дюймов), изготовленные из отражающего материала, покрытого прозрачным защитным слоем. Информация записывается посредством создания изменений на отражающей поверхности диска и считывается с помощью лазерного луча, который отслеживает неравномерности на отражающей поверхности диска во время его вращения.

Технология изготовления компакт-дисков изначально применялась в производстве аудиозаписей с использованием формата, известного как **CD-DA** (компакт-диск — цифровое аудио). Компакт-диски, используемые в настоящее время для хранения компьютерных данных, похожи на своих аудиопредшественников. В частности, информация на эти диски записывается в виде одной дорожки, спирально закрученной от центра диска к его краю (рис. 1.10). Эта дорожка разделена на секторы, каждый из которых имеет идентификационные метки и размер 2 Кбайт, что эквивалентно 1/75 секунды звучания в случае аудиозаписи.

Обратите внимание, что длина одного оборота спиральной дорожки увеличивается по направлению от внутренней части диска к внешней. Из соображений увеличения емкости компакт-диска информация записывается с одной и той же линейной плотностью по всей длине спиральной дорожки, а это означает, что на витке во внешней части спирали хранится больше информации, чем на витке в ее внутренней части. Поэтому за один оборот диска будет считываться больше секторов, когда лазерный луч будет сканировать внешнюю часть спиральной дорожки, и меньше секторов, когда луч будет сканировать внутреннюю часть дорожки. В результате, чтобы получить равномерную скорость пересылки данных, CD-плееры разрабатываются таким образом, чтобы можно было изменять скорость вращения диска в зависимости от расположения лазерного луча.



Рис. 1.10. Формат записи данных на компакт-диске

Благодаря подобным конструктивным решениям запоминающие системы с компакт-дисками имеют большую производительность при работе с длинными, непрерывными строками данных, например при воспроизведении музыки. Однако, если прикладной программе требуется произвольный доступ к данным, подход, используемый в устройствах магнитных дисков (отдельные концентрические дорожки, каждая из которых разбита на секторы, доступ к которым может быть получен независимо от других), оказывается эффективнее спирального метода записи, используемого в компакт-дисках.

Емкость компакт-диска в формате CD составляет от 600 до 700 Мбайт. Однако диски формата **DVD** (Digital Versatile Disk — цифровой универсальный диск), созданные из нескольких полупрозрачных слоев, которые служат отдельными поверхностями при считывании точно сфокусированным лазерным лучом, позволяют достичь емкости носителя до нескольких гигабайтов. На таких компакт-дисках можно хранить продолжительные мультимедиапрезентации, в которых комбинируется аудио- и видеоинформация. И наконец, однослойный **BD-диск** (Blu-ray Disks) обладает емкостью, которая в пять раз превосходит емкость DVD-диска. Недавно появившийся многослойный BD-формат позволяет достичь емкости диска в пределах 100 Гбайт. Этот кажущийся просто огромным объем носителя в действительности вполне отвечает требованиям видео, записанного в формате ультравысокого разрешения (UHD или 4K).

Флеш-накопители

Общим свойством устройств массовой памяти, построенных на основе магнитной или оптической технологии, является наличие физически движущихся элементов, — таких, как вращающиеся диски, перемещающиеся головки чтения/записи и направляемые лазерные лучи, — используемых для записи и считывания данных. А это означает, что данные помещаются на носитель и

считываются с него с относительно небольшой скоростью в сравнении со скоростью работы электронных схем. Технология **флеш-памяти** имеет потенциал, позволяющий смягчить этот недостаток. В устройстве флеш-памяти биты сохраняются посредством отправки электронных сигналов непосредственно в сохраняющую среду, где они вызывают попадание и захват электронов в крошечных камерах из двуокиси кремния, вследствие чего изменяются характеристики микроскопических электронных схем. Поскольку эти камеры способны многие годы удерживать захваченные электроны без использования внешних источников энергии, эта технология превосходно подходит для создания портативных, энергонезависимых хранилищ данных.

Хотя данные, сохраненные во флеш-памяти, могут быть считаны в виде небольших фрагментов размером один байт, — как это имеет место в основной памяти компьютеров, — современные технологии диктуют, чтобы хранимые данные *стирались* в больших блоках. Более того, многократное стирание данных медленно разрушает камеры из двуокиси кремния, а это означает, что современная технология флеш-памяти не может использоваться для изготовления основной памяти компьютеров, содержимое ячеек которой может изменяться тысячи раз в секунду. Однако в тех приложениях, в которых частота изменения содержимого памяти сохраняется на приемлемом уровне, — как, например, в цифровых фотокамерах или смартфонах, — флеш-память вполне может найти себе самое широкое применение. Действительно, поскольку флеш-память не чувствительна к физическим воздействиям (в противоположность магнитным и оптическим устройствам), она сейчас активно заменяет прочие технологии массовой памяти и в портативных компьютерных устройствах, таких как ноутбуки и даже ПК.

Устройства, изготовленные по флеш-технологии, называют **флеш-накопителями**, их емкость сейчас может достигать сотен гигабайтов и они вполне применимы для типичных случаев использования устройств массовой памяти. Эти накопители чаще всего упаковывают в пластиковые корпуса небольших размеров со съемными крышками на одном конце, защищающими разъем, когда накопитель извлечен из компьютера. Высокая емкость этих портативных устройств наряду с тем фактом, что они очень легко подключаются к компьютеру или отключаются от него, делают их идеальными портативными хранилищами данных. Тем не менее уязвимость их крошечных, хранящих заряды камер приводит к тому, что эти устройства следует считать не столь надежными, как оптические диски, в смысле действительно долговременного использования.

Более крупные устройства флеш-памяти называют **SSD-дисками** (solid-state drives), они были разработаны непосредственно для того, чтобы занять место магнитных жестких дисков. В сравнении с жесткими дисками SSD-диски выигрывают в плане их устойчивости к вибрациям и резким физическим воздей-

ствиям, бесшумностью работы (у них нет подвижных частей) и меньшим временем доступа. SSD-диски все еще дороже жестких дисков при сравнимых объемах, и по этой причине все еще рассматриваются как элементы высшего класса при приобретении настольного компьютера. В случае мобильных устройств, таких как ноутбуки и смартфоны, SSD-устройства имеют очевидные преимущества. Секторам SSD-дисков свойственен более ограниченный срок службы, что характерно для всех устройств памяти, изготовленных по флеш-технологии, однако использование технологии **выравнивания износа** позволяет ослабить влияние этого фактора посредством регулярного перемещения часто изменяемых блоков данных в другие местоположения на носителе.

Другим приложением флеш-технологии являются **карты памяти** типа **SD** (Secure Digital) или просто SD-карты. Их объем может достигать 2 Гбайт, и при этом они просто запрессованы в пластиковые корпуса — пластины размером с небольшую почтовую марку (SD-карты также выпускаются в корпусах меньших размеров — в формате мини-SD и микро-SD). Карты памяти формата **SDHC** (High Capacity) могут иметь емкость до 32 Гбайт, а более нового формата **SDXC** (Extended Capacity) могут иметь объем до одного терабайта. Благодаря компактным физическим размерам карты памяти очень удобно вставлять в слоты различных миниатюрных электронных приборов, а значит, это идеальное решение для таких устройств, как цифровые фотокамеры, аудиоплееры, навигационные приборы для автомобилей, видеорегистраторы и т.д.

1.3. Вопросы и упражнения

1. Какие преимущества дает большая скорость вращения устройствам с жестким диском или компакт-диском?
2. При записи информации на устройство дисковой памяти с несколькими дисковыми пластинами следует ли сначала использовать всю поверхность одной дисковой пластины, прежде чем приступить к записи на поверхность другой пластины, или же целесообразнее осуществить запись на всем цилиндре, прежде чем переходить к следующему?
3. Почему информация в системе резервирования авиабилетов, которая подвержена постоянному обновлению, должна храниться на магнитном диске, а не на CD или DVD?
4. Какие факторы обеспечивают компакт-дискам, DVD-дискам и BD-дискам возможность их прочтения на одном и том же устройстве чтения?

5. Какие преимущества характерны именно для флеш-накопителей в сравнении другими устройствами массовой памяти, обсуждавшимися в этом разделе?
6. Какие преимущества магнитных жестких дисков позволяют им продолжать успешно конкурировать с другими типами устройств массовой памяти?

1.4. Представление информации в виде комбинации двоичных разрядов

Познакомившись с различными технологиями хранения данных, в этом разделе мы рассмотрим, как эти данные в машине могут быть представлены в виде комбинации двоичных разрядов (битов). В частности, мы обсудим популярные методы кодирования текста, представления цифровых данных, кодирования изображений и звука. Каждый из указанных методов имеет специфические особенности, с которыми часто сталкивается типичный пользователь компьютера. Наша цель — ознакомиться с этими технологиями в достаточной степени, чтобы в полной мере оценить их важность для тех, для кого они предназначены.

Представление текста

Информация в форме текста обычно представляется с помощью кода, причем каждому отличному от других символу текста (например, букве алфавита или знаку пунктуации) присваивается уникальная комбинация двоичных разрядов. В этом случае текст будет представлен как длинный ряд битов, в котором следующие одна за другой комбинации битов отражают последовательность символов в исходном тексте.

В 1940- и 1950-е годы было разработано много подобных кодов, причем каждый из них использовался в различных типах оборудования, что привело к появлению ряда проблем, связанных с передачей информации. Во избежание этих проблем *Американский национальный институт стандартов* (American National Standards Institute — **ANSI**) принял американский *стандартный код для обмена информацией* (American Standard Code for Information Interchange — **ASCII**, произносится как "эс-ки-и"), который приобрел очень большую популярность. В этом коде комбинации двоичных разрядов длиной 7 бит используются для представления строчных и прописных букв английского алфавита, знаков пунктуации, цифр от 0 до 9, а также кодов управления передачей инфор-

мации (таких, как перевод строки, возврат каретки или табуляция). В наше время код ASCII часто употребляется в расширенном восьмиразрядном формате, который получается посредством добавления нуля в старший конец каждого семиразрядного кода. Благодаря этому можно получить не только коды, размер которых соответствует типичной однобайтовой ячейке памяти, но и 128 новых дополнительных комбинаций двоичных разрядов (которые получаются в результате добавления в старший конец бита со значением `1`). Это позволяет представлять символы, не поддерживаемые исходной версией кода ASCII.[1]

В приложении А представлена часть таблицы кода ASCII в восьмиразрядном формате, а на рис. 1.12 показано, как в этой кодировке приветствие "Hello." представляется с помощью следующей комбинации битов:

```
01001000 01100101 01101100 01101100 01101111 00101110
```

Международная организация по стандартизации (International Organization for Standardization, часто именуемая **ISO**, от греческого *isos* — одинаковый) разработала множество расширений кодировки ASCII, каждое из которых было предназначено для представления дополнительных символов различных алфавитов, используемых в той или иной языковой группе. Например, один из таких стандартов обеспечивал представление символов, необходимых для полноценного представления текстов на большинстве западноевропейских языков. Он включал 128 дополнительных символов и знаков, в частности таких, как символ обозначения британской валюты (фунт стерлингов: £) или букв, представляющих немецкие гласные (ä, ö и ü).

01001000	01100101	01101100	01101100	01101111	00101110
H	e	l	l	o	.

Рис. 1.11. Представление фразы "Hello." в кодировке ASCII или UTF-8

Расширяя исходный стандарт ASCII, организация ISO добилась огромного прогресса в направлении поддержки всего многообразия существующих в мире языков в целях обеспечения многоязычного общения. Однако на этом пути возникли два серьезных препятствия. Во-первых, количества дополнительных кодовых комбинаций, доступных в расширенном стандарте ASCII, было просто недостаточно для представления полных алфавитов многих азиатских и некоторых восточноевропейских языков. Во-вторых, поскольку каждый документ создавался с использованием символов, доступных только в одном выбранном стандарте, создание документов, содержащих тексты на языках из

[1] К сожалению, из-за того, что фирмы-разработчики широко использовали собственные варианты толкования этих дополнительных кодов, данные, представленные в этих кодах, оказалось не так-то просто переносить из одной программы в другую, особенно если эти программы были разработаны разными фирмами.

Американский национальный институт стандартов

Американский национальный институт стандартов (ANSI) был основан в 1918 году небольшим консорциумом машиностроительных ассоциаций и государственными агентствами как некоммерческое объединение. Его задача — управление разработкой различных стандартов в частном секторе. В настоящее время число членов ANSI превысило 1300 деловых организаций, профессиональных объединений, торгово-промышленных ассоциаций и государственных агентств. Штаб-квартира этой организации расположена в Нью-Йорке. Институт ANSI представляет США в международной организации ISO. Веб-сайт Американского национального института стандартов расположен по адресу `http://www.ansi.org`.

В других странах также имеются подобные учреждения, среди которых Standards Australia (Австралия), Standards Council of Canada (Канада), China State Bureau of Quality and Technical Supervision (Китай), Deutsches Institur fur Normung (Германия), Japanese Industrial Standards Committee (Япония), Direccion General de Normas (Мексика), State Committee of the Russian Federation for Standardization and Metrology (Российская Федерация), Swiss Association for Standardization (Швейцария) и British Standards Institution (Великобритания).

разных языковых групп, было просто невозможно. Оба эти ограничения создавали серьезные препятствия для широкого международного использования стандартов ASCI. Чтобы устранить указанные затруднения, был создан код, получивший название **Unicode**. Он был разработан в результате объединенных усилий нескольких ведущих фирм — производителей программного и аппаратного обеспечения и очень быстро получил широкую поддержку компьютерного сообщества. В этом коде для представления отдельного символа используется уникальное битовое сочетание длиной до 21 бит. Когда набор символов кода Unicode комбинируется со стандартом кодировки *Формат преобразования Юникода, 8-бит* (Unicode Transformation Format 8-bit — **UTF-8**), исходный набор ASCII-символов может по-прежнему быть представлен с помощью 8 бит, тогда как тысячи дополнительных символов из таких языков, как китайский, японский или иврит, могут быть представлены с использованием 16 бит. Помимо кодирования символов, необходимых для всех общеупотребительных языков мира, стандарт UTF-8 позволяет использовать 24- или 32-битные комбинации для представления пока еще не принятых символов Юникода, оставляя достаточно места для будущего расширения.

Файл, состоящий из длинной последовательности символов, закодированных в кодировках ASCII или Unicode, часто называют **текстовым файлом**. Очень важно понимать отличия, существующие между *простыми* текстовыми файлами, которые создаются и обрабатываются служебными программами,

Международная организация по стандартизации

Международная организация по стандартизации (ISO) была основана в 1947 году как Всемирная федерация органов стандартизации, представляющая все страны, по одному представителю от каждой страны. Сегодня эта федерация, штаб-квартира которой находится в Женеве (Швейцария), насчитывает более 100 организаций-членов, а также большое количество членов-корреспондентов. (Членами-корреспондентами обычно являются организации из тех стран, которые не имеют своей организации по стандартизации. Они не могут непосредственно участвовать в разработке стандартов, однако им предоставляется информация о деятельности ISO.) Организация ISO поддерживает веб-сайт, доступный по адресу `http://www.iso.org`.

получившими **текстовые редакторы** (или просто редакторы), и гораздо более сложными файлами, создаваемыми **текстовыми процессорами**, такими как Microsoft Word. Файлы обоих типов содержат текстовый материал, однако простой текстовый файл содержит только текст, представленный последовательностью соответствующих кодов символов, тогда как файл, созданный текстовым процессором, включает множество дополнительных структур данных, представляющих форматирование текста, информацию о выравнивании абзацев, изображения, таблицы и много другое.

Представление числовых значений

Хотя метод хранения информации в виде закодированных символов в целом достаточно удобен, он оказывается неэффективным при записи чисто числовой информации. Попробуем разобраться, почему это так. Предположим, что в память требуется записать число 25. Если воспользоваться символами в кодах ASCII, то для записи этого числа потребуется 1 байт на каждый символ, а всего — 16 битов. Более того, самое большое число, которое мы сможем представить с помощью 16 битов, — это 99. Однако, как мы вскоре увидим, используя двоичную нотацию в этих же 16 битах, можно сохранить любое целое число в диапазоне от 0 до 65 535. Именно по этой причине двоичная нотация (или ее модифицированные варианты) широко используется для представления числовых данных в памяти компьютеров.

Двоичная система счисления представляет собой средство выражения цифровых величин с помощью только двух цифр, `0` и `1`, а не всех десяти — 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, — как в традиционной десятичной системе счисления. Подробно двоичная система счисления будет обсуждаться в разделе 1.5, а пока нам доста-

точно самых элементарных представлений о том, что она собой представляет. Для этой цели представим себе старомодный прибор-одометр, измеряющий пробег автомобиля, — несколько вращающихся колес с цифрами, общее положение которых и представляет величину пробега. Быстрее всех вращается крайнее справа колесико, и когда оно пройдет полный оборот, соседнее слева колесико сдвинется на один шаг. После следующего оборота правого колесика левое колесико сдвинется еще на один шаг и т.д. Однако в нашем случае на колесиках будут представлены не все десять цифр, а только две — нуль и единица. В начале работы все колесики нашего одометра установлены в положение `0`, а когда автомобиль проедет первый километр, крайнее справа колесико окажется в положении `1`. Далее, когда машина проедет еще километр, правое колесико совершит полный оборот, и на нем вновь появится значение `0`, тогда как на колесике слева от него появится `1`. Таким образом, общее показание прибора будет `10`. Еще через километр правое колесико будет вновь показывать `1`, а весь прибор — `11`. После четвертого километра правое колесико вернется в положение `0`, в это же положение вернется и колесико левее его, которое при этом сдвинет на единицу положение следующего колесика левее от него, так что прибор теперь будет показывать `100`. Следуя этой схеме, на нашем приборе через каждый километр, начиная с исходной точки, последовательно будут появляться следующие значения:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
```

Эта последовательность являет собой двоичное представление целых чисел от нуля до восьми. Хотя это будет довольно утомительно, мы легко могли продолжить использовать этот метод подсчета, чтобы обнаружить, что комбинация, состоящая из шестнадцати единиц, представляет значение 65 535, что и подтверждает наше приведенное выше утверждение о том, что любое целое число в диапазоне от 0 до 65 535 может быть закодировано с использованием 16 бит.

Именно благодаря подобной эффективности общепринятой практикой является хранение числовой информации в формате двоичной нотации, а не в виде кодов символов. Мы используем термин “в формате двоичной нотации” по той причине, что только что описанная простейшая двоичная система является только основой, на которой строится несколько более сложных мето-

дов представления числовых данных, используемых в современных машинах. Некоторые из этих вариаций на базе двоичной системы счисления будут подробно обсуждаться ниже в этой главе. А сейчас мы только отметим, что система, получившая название **двоичный дополнительный код** (см. раздел 1.6), обычно используется для хранения *целых* чисел, поскольку предоставляет удобный способ хранения как положительных, так и отрицательных целых значений, тогда как для представления дробных чисел, таких как $4^1/_2$ или $^3/_4$, обычно используется другой метод, получивший название **формат с плавающей точкой** (см. раздел 1.7).

И наконец, записываем ли мы число, используя двоичную нотацию (основание 2), более привычную нам десятичную нотацию (основание 10) или более компактную шестнадцатеричную нотацию (основание 16), представленные ниже числовые величины остаются теми же самыми. Иначе говоря, два плюс два равно четырем, и это утверждение остается верным, записываем ли мы его в двоичном формате (`010 + 010 = 100`), в десятичной нотации (2 + 2 = 4), шестнадцатеричном представлении (`0x2 + 0x2 = 0x4`) или в числовой нотации с любом другим основанием. Что касается компьютеров, то для них это всегда лишь двоичные единицы и нули.



Основные положения для запоминания

- **Числовые нотации с различным основанием, включая двоичную, десятичную и шестнадцатеричную, используются для представления и изучения числовых данных.**
- **Числа всегда могут быть преобразованы из представления в одной системе счисления в систему счисления с любым другим основанием.**

Представление изображений

Один из методов представления изображений заключается в его интерпретации как совокупности точек (растра), которые в этом случае называют **пикселями**, что является сокращением от англ. *picture element* — "элемент изображения". Вид каждого пикселя затем тем или иным образом кодируется, а все изображение представляется совокупностью этих закодированных пикселей. Такая совокупность называется **битовой картой**. Данный подход очень популярен, поскольку многие отображающие устройства, такие как принтеры и экраны мониторов, телевизоров или смартфонов, работают именно по принципу манипулирования пикселями. В результате изображения в формате битовой карты очень просто отображаются такими устройствами.

Метод кодирования значений пикселей в битовой карте варьируется в зависимости от конкретных приложений. В случае простых черно-белых изображений каждый пиксель может быть представлен единственным битом, значение которого зависит от того, каким является соответствующий ему пиксель — белым или черным. Этот подход используется в большинстве факсимильных машин. Для более сложного случая черно-белых фотографий каждый пиксель может быть представлен совокупностью битов (чаще всего их восемь), что позволяет кодировать его значение в оттенках серого. В случае цветных изображений каждый пиксель кодируется с использованием более сложных методов. Чаще всего используются два подхода. В первом, известном как кодирование в формате **RGB**, каждый пиксель представляется тремя цветовыми компонентами, красным, зеленым и синим, в соответствии с тремя первичными цветами света. Интенсивность каждого цвета обычно представляется значением одного байта, в результате чего для кодирования цвета одного пикселя исходного изображения необходимо использовать *три* байта.

Альтернативой относительно простому методу кодирования в формате RGB является использование "яркостной" компоненты и двух компонент, представляющих цветовые характеристики, — такой формат обозначают как **Lab**. В случае "яркостной" компоненты, которую в данном формате называют "L" (*luminance* — яркость), ее значение, по сути, представляет собой сумму красной, зеленой и синей составляющих цвета. (В действительности она понимается как количество белого света в пикселе, но эти подробности в данном случае не имеют для нас значения.) Две остальные компоненты, a и b, называют хроматическими, определяющими тон, насыщенность цвета. Они вычисляются как разность между яркостью пикселя и количеством в нем синего (a) и красного (b) цветов соответственно. Все вместе эти компоненты включают всю необходимую информацию для воспроизведения пикселя.

Популярность кодирования изображений в формате Lab имеет свои корни в постепенном распространении цветного телевидения, когда оно пришло на смену черно-белому. Этот формат обеспечивал такой способ кодирования цветных изображений, который был легко совместим с более старыми черно-белыми телевизионными приемниками. И действительно, версия изображения в оттенках серого легко может быть получена посредством использования только компонента яркости закодированного в формате Lab цветного изображения.

Существенным недостатком представления изображений в формате битовой карты является трудность пропорционального изменения размеров изображения до произвольно выбранного значения. В сущности, единственный способ увеличить изображение — это увеличить сами пиксели. Однако это приводит к появлению зернистости. (Такой подход называется "цифровым увеличением" и довольно широко используется в простых цифровых фотокамерах в проти-

воположность "оптическому увеличению", которое достигается посредством изменения взаимного расположения линз объектива и характерно для более сложных и дорогих фотокамер.)

Альтернативный способ представления изображений, позволяющий избежать проблем масштабирования, характерных для растровых методов, состоит в описании изображения как совокупности геометрических структур, таких как линии, плоскости и кривые, которые могут быть закодированы с использованием средств аналитической геометрии. Такое описание позволяет устройству, которое будет отображать это изображение, самому решить, как в конечном счете должна быть представлена та или иная геометрическая структура, вместо того чтобы просто отобразить жестко заданную в описании совокупность пикселей. Именно такой подход используется для отображения масштабируемых шрифтов, широко используемых в современных текстовых процессорах. Например, система кодирования шрифтов TrueType (разработана корпорациями Microsoft и Apple) представляет собой набор правил геометрического описания текстовых символов. Аналогичным образом система PostScript (разработана корпорацией Adobe Systems) предоставляет инструменты описания символов текста, а также графических данных более общего характера. Схожие геометрические инструменты представления изображений также широко используются в приложениях *систем автоматического проектирования* (Computer-aided design — **CAD**), которые отображают на экране компьютерных дисплеев чертежи сложных трехмерных объектов и предоставляют средства манипулирования ими.

Различия между представлением изображений в виде совокупности геометрических структур и в виде битовой карты совершенно очевидны для пользователей многих графических приложений (таких, как утилита Microsoft Paint), которые предоставляют пользователю возможность рисовать собственные изображения, состоящие из различных заранее предопределенных элементов, таких как треугольники, эллипсы или элементарные кривые. При рисовании пользователь просто выбирает желаемый геометрический элемент в меню и управляет прорисовкой этого элемента на экране с помощью мыши. В процессе рисования программное обеспечение формирует геометрическое описание элемента непосредственно в процессе его создания. По мере того, как характеристики рисуемого элемента определяются движением мыши, его внутреннее геометрическое представление модифицируется, конвертируется в битовую карту и она отображается на экране. Такой подход позволяет легко масштабировать изображение и менять форму его отдельных элементов. Однако в очень простом приложении Paint, как только процесс рисования элемента завершается, его внутреннее геометрическое описание аннулируется и сохраняется только конечное представление этого элемента в виде битовой карты. А это означает, что любые дополнительные изменения данного элемента потребуют

утомительной процедуры модификации отдельных его пикселей. Тем не менее существует множество других графических приложений, в которых геометрическое описание элементов изображения сохраняется в файле, что позволяет легко модифицировать их впоследствии. Такой формат изображений называется **векторным** и, в отличие от растрового, обеспечивает возможность произвольного изменения размеров как изображения в целом, так и отдельных его элементов, гарантируя при этом максимально четкое их отображение.

Представление звука

Наиболее общим методом кодирования аудиоинформации для сохранения и последующей обработки в компьютерах являются определение амплитуды звуковой волны через регулярные промежутки времени и запись в файл серии полученных значений. Например, серия значений 0; 1,5; 2,0; 1,5; 2,0; 3,0; 4,0; 3,0; 0 будет представлять звуковую волну, которая нарастает по амплитуде, потом несколько слабеет и вновь возрастает до еще большего уровня, после чего ее амплитуда падает до нуля (рис. 1.12). Этот метод с частотой выборки в 8000 измерений в секунду многие годы широко использовался для обеспечения голосовой телефонной связи на большие расстояния. Голос на одном конце линии связи кодировался цифровыми значениями, представлявшими амплитуду голосового сигнала каждую одну восьмитысячную секунды. Затем эти числовые значения передавались по линии связи второму абоненту, где использовались для воспроизведения звуков голоса второго абонента.

Хотя фиксация 8000 значений амплитуды в секунду может показаться довольно большой величиной, этого совершенно недостаточно для высококачественной записи музыки. Для качественного воспроизведения звука, обеспечиваемого музыкальными компакт-дисками, требуется частота выборки на уровне как минимум 44 100 значений в секунду. При этом данные, полученные при каждой выборке, представляются значением 16 бит (32 бита для стереозаписей). Соответственно, каждая секунда музыкальной записи для сохранения требует более миллиона битов памяти.

Альтернативной системой кодирования звука является *цифровой интерфейс музыкальных инструментов* (Musical Instrument Digital Interface — **MIDI**), который широко применяется в музыкальных синтезаторах, использующих электронную клавиатуру, для звукового оформления компьютерных игр, а также для создания звукового оформления веб-сайтов. Кодируя *указания* по воспроизведению музыки на синтезаторе, вместо кодирования собственно музыки, формат MIDI позволяет избежать чрезмерных требований к объемам памяти, типичных для предыдущего формата. Если говорить точнее, в формате MIDI кодируется, какой инструмент должен звучать, какую ноту он исполняет, как

долго и с какой громкостью. Это означает, например, что кларнет должен воспроизводить ноту "ре" первой октавы в течение двух секунд. Для кодирования этой информации достаточно трех байтов вместо двух миллионов битов при записи звука по первому методу с частотой дискретизации 44 100 Гц.

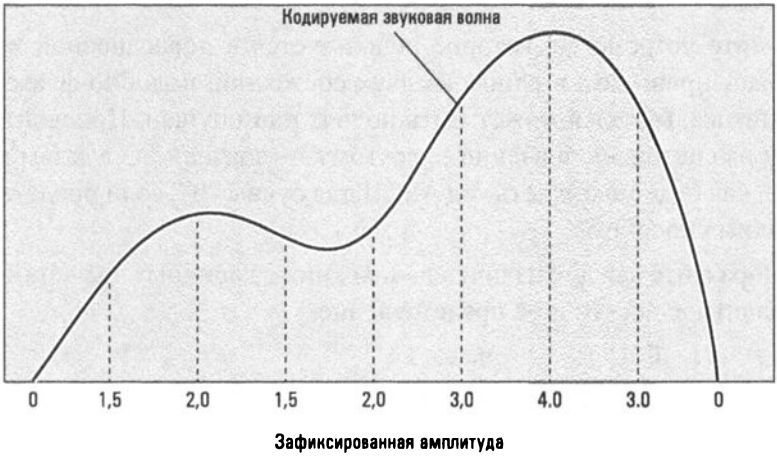


Рис. 1.12. Звуковая волна представляется последовательностью значений 0; 1,5; 2.0; 1.5; 2,0; 3,0; 4,0; 3,0; 0

Говоря простым языком, формат MIDI можно понимать как закодированную инструкцию, которую должен прочитать исполнитель, а не как запись самого исполнения. А это, в свою очередь, означает, что "запись" в формате MIDI может звучать по-разному, если будет воспроизводиться на разных устройствах воспроизведения.

1.4. Вопросы и упражнения

1. Ниже приведено сообщение, представленное в виде символов ASCII с использованием восьми битов на один символ. Каков текст этого сообщения? (См. приложение А.)

```
01000011 01101111 01101101 01110000 01110101 01110100
01100101 01110010 00100000 01010011 01100011 01101001
01100101 01101110 01100011 01100101
```

2. Какая взаимосвязь между представлением строчных и соответствующих прописных букв существует в кодировке ASCII? (См. приложение А.)

3. Зашифруйте приведенные ниже предложения с помощью символов кода ASCII.

   а. "Stop!" Cheryl shouted.

   б. Does 2 + 3 = 5?

4. Опишите устройство, которое используется в повседневной жизни и может пребывать в одном из двух состояний, подобно флажку на флагштоке, который может быть поднят или опущен. Присвойте одному из состояний значение `1`, другому — значение `0`, а затем покажите, как будет выглядеть код ASCII для буквы "b", если представить ее таким способом.

5. Преобразуйте каждое из приведенных ниже двоичных значений в эквивалентное десятичное представление.

   а. `0101` **б.** `1001` **в.** `1011`

   г. `0110` **д.** `10000` **е.** `10010`

6. Преобразуйте каждое из приведенных ниже десятичных значений в эквивалентное двоичное представление.

   а. 6 **б.** 13 **в.** 11

   г. 18 **д.** 27 **е.** 4

7. Какое наибольшее числовое значение может быть представлено в трех байтах памяти, если каждая цифра будет зашифрована в виде символа кода ASCII? Каким будет ответ при использовании двоичного представления числа?

8. Альтернативой шестнадцатеричной системе счисления в отношении представления битовых комбинаций является десятичная нотация с точками, в которой каждый байт битовой комбинации представляется эквивалентным десятичным значением. Эти представления отдельных байтов, в свою очередь, разделяются точками. Например, значение 12.5 представляет комбинацию `0000110000000101` (байт `00001100` представлен числом 12, а байт `00000101` — числом 5), а комбинация `100010000010000000000111` может быть представлена как 136.16.7. Представьте приведенные ниже битовые комбинации в десятичной нотации с точками.

   а. `0000111100001111` **б.** `001100110000000010000000`

   в. `0000101010100000`

9. Какими преимуществами обладает метод представления изображений с помощью геометрических структур в сравнении с растровым методом? А что можно сказать о достоинствах растрового метода в сравнении с методом представления изображений с помощью геометрических структур?
10. Предположим, что стереозапись одного часа музыки выполнена с частотой дискретизации 44 100 раз в секунду, как это обсуждалось выше. Что можно сказать о размере этой записи в сравнении с объемом памяти, доступным на компакт-диске?

1.5. Двоичная система счисления

В разделе 1.4 вы познакомились с двоичной системой счисления (двоичной нотацией) как средством представления числовых значений с использованием только двух цифр — `0` и `1` вместо десяти цифр от 0 до 9, как это делается в привычной нам нотации с основанием десять (десятичной системе счисления). Теперь пришло время познакомиться с двоичной нотацией более подробно.

Двоичная нотация

Вспомним, что в десятичной системе счисления каждой цифровой позиции в представлении числа приписывается определенное весовое значение. Например, в представлении числа 375 позиция цифры 5 имеет весовое значение "единица", позиция цифры 7 — весовое значение "десять", а позиция цифры 3 — весовое значение "сто" (рис. 1.13, *а*). В этом случае весовое значение каждой позиции в десять раз превосходит весовое значение предыдущей позиции (расположенной правее). Представляемая величина определяется посредством умножения каждой цифры на весовое значение занимаемой ею позиции с последующим сложением полученных результатов. Таким образом, комбинация цифр 375 представляет величину (3 × *сто*) + (7 × *десять*) + (5 × *один*), что в более технической нотации можно записать как $(3 \times 10^2) + (7 \times 10^1) + (5 \times 10^0)$.

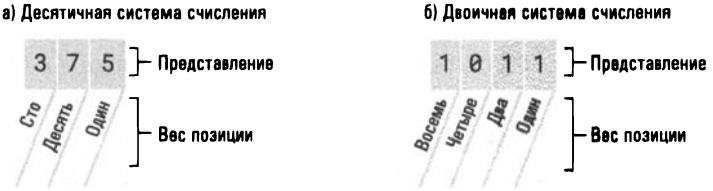


Рис. 1.13. Двоичная и десятичная системы счисления

В двоичной системе счисления позиция каждой цифры также связывается с определенным весовым значением, однако в этом случае весовое значение каждой позиции превосходит последующее только в два раза. Для большей определенности скажем, что крайняя справа цифровая позиция в двоичном представлении числа имеет весовое значение "один" (2^0), следующая цифровая позиция слева — весовое значение "два" (2^1), следующая позиция — весовое значение "четыре" (2^2), а позиция за ней — весовое значение "восемь" (2^3) и т.д. Например, в двоичном числе `1011` позиция крайней справа цифры `1` имеет весовое значение "один", позиция следующей единицы — весовое значение "два", позиция цифры `0` — весовое значение "четыре", а позиция крайней слева единицы имеет весовое значение "восемь" (см. рис. 1.13, *б*).

Для определения числового значения, представленного в двоичной системе счисления, выполняются те же действия, что и при его записи в десятичной системе счисления: каждая его цифра умножается на весовое значение занимаемой ею позиции и полученные результаты суммируются. Например, двоичное число `100101` имеет значение 37, как показано на рис. 1.14. Более того, поскольку в двоичной системе счисления используются только цифры `0` и `1`, общая процедура умножения и суммирования результатов сокращается до суммирования весовых значений позиций, в которых находятся единицы. Например, двоичное число `1011` представляет значение 11, так как единицы в нем расположены в позициях с весовыми значениями "один", "два" и "восемь".

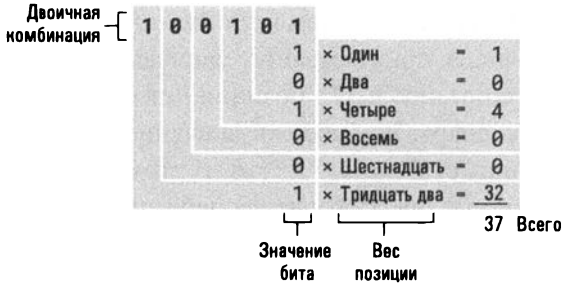


Рис. 1.14. Расшифровка значения двоичного числа 100101

Из раздела 1.4 вы узнали, как в двоичной нотации осуществляется последовательный счет, что позволило нам закодировать несколько первых целых чисел. Для определения двоичного представления бо́льших чисел можно использовать более систематический подход, описанный в алгоритме, представленном на рис. 1.15. Давайте применим этот алгоритм для определения двоичного представления числа "тринадцать" (рис. 1.16). Сначала следует поделить это число на два, в результате получим частное шесть и остаток — единицу. Так как частное не равняется нулю, этап 2 предписывает поделить число

шесть на два, в результате будет получено частное три и остаток, равный нулю. Поскольку новое частное тоже не равняется нулю, следует разделить его на два, получив частное единица и остаток единица. Затем вновь следует поделить очередное частное (единица) на два, и на этот раз частное оказывается равным нулю, а остаток — единице. Теперь, когда мы получили частное, равное нулю, можно перейти к этапу 3. В результате для исходного числа 13 будет получено двоичное представление 1101, построенное как последовательность остатков в операциях деления, записанных справа налево.

Этап 1. Поделите число на два и запишите остаток.

Этап 2. Пока не будет получено частное, равное нулю, выполняйте операцию деления предыдущего частного на два и записывайте полученный при каждом делении остаток.

Этап 3. Как только будет получено частное, равное нулю, двоичное представление исходного числа можно будет записать как последовательность всех полученных остатков от деления, последовательно расположенных в направлении справа налево.

Рис. 1.15. Алгоритм вычисления двоичного представления для произвольного положительного числа

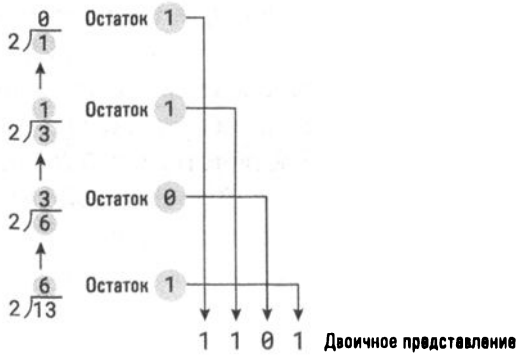


Рис. 1.16. Применение алгоритма, представленного на рис. 1.15, для вычисления двоичного представления числа "тринадцать"

Двоичное сложение

Чтобы понять процедуру сложения двух целых чисел, представленных в двоичной нотации, давайте прежде всего освежим в памяти процедуру сложения чисел, представленных в традиционной десятичной системе счисления. Рассмотрим, например, как решается следующая задача.

```
  58
+ 27
```

Вначале необходимо сложить цифры 8 и 7 в крайнем справа столбце, — получится 15. Далее записываем цифру 5 младшего разряда полученной суммы под этим столбцом, а цифру 1 из старшего разряда полученной суммы переносим в следующий слева столбец.

```
  1
  58
+ 27
  --
   5
```

Теперь складываем цифры 5 и 2 в следующем столбце и добавляем к их сумме единицу, которая была перенесена из предыдущего столбца. Получаем результат 8, который и записываем под данным столбцом. Полученный результат будет следующим.

```
  58
+ 27
  --
  85
```

В общем виде процедура будет следующей: в направлении справа налево необходимо сложить цифры в каждом столбце, записать цифру младшего разряда полученной суммы под этим столбцом, а цифру старшего разряда полученной суммы (если таковая имеется) перенести в следующий слева столбец.

Для суммирования двух чисел, представленных в двоичной системе счисления, мы будем следовать той же процедуре за исключением того, что все суммы будут вычисляться с использованием соответствующей таблицы правил двоичного сложения, представленной на рис. 1.17, вместо той традиционной таблицы правил сложения для десятичной системы счисления, которую мы изучали в начальной школе. В качестве примера рассмотрим решение следующей задачи:

```
 111010
+ 11011
```

Начинаем с операции сложения крайних справа цифр `0` и `1`, в результате получим число `1` и запишем его под этим столбцом. Затем сложим цифры `1` и `1` в следующем столбце и получим число `10`. Цифру `0` этого числа запишем под данным столбцом, а цифру `1` перенесем в следующий столбец и запишем над ним. На данном этапе это будет выглядеть следующим образом:

```
   1
 111010
+ 11011
 ------
   01
```

```
 0   1   0   1
+0  +0  +1  +1
--  --  --  --
 0   1   1  10
```

Рис. 1.17. Правила двоичного сложения

В следующем столбце сумма цифр 1, 0 и 0 будет равна числу 1, поэтому под данным столбцом запишем цифру 1. Сумма цифр 1 и 1 в очередном столбце составляет число 10. Запишем под данным столбцом цифру 0, а цифру 1 перенесем в следующий столбец, как показано ниже:

```
  1
 111010
+ 11011
   0101
```

Сумма цифр 1, 1 и 1 в очередном столбце составит число 11 (в двоичной нотации это число три), поэтому цифра 1 из младшего разряда записывается под данным столбцом, а цифра 1 из старшего разряда переносится в следующий столбец. Суммируя цифры 1, 1 и 0 в следующем столбце, получаем число 10. И вновь, цифра 0 из младшего разряда записывается под данным столбцом, а цифра 1 из старшего разряда переносится в следующий столбец, как показано ниже:

```
 1
  111010
+  11011
  010101
```

В очередном столбце единственным имеющимся значением является цифра 1, перенесенная из предыдущего столбца. Записываем ее в ответ и получаем окончательное решение задачи, которое выглядит следующим образом:

```
  111010
+  11011
 1010101
```

Представление дробей в двоичных кодах

Чтобы иметь возможность работать с дробями в двоичной системе счисления, мы применяем **позиционную точку**, подобную используемой в десятичных дробях. Цифры слева от точки представляют целую часть числа (мантиссу) и обрабатываются точно так, как целые числа, записанные в двоичной системе, о чем речь уже шла выше. Цифры справа представляют дробную часть значения и обрабатываются аналогично любым другим битам, за исключением того, что их позициям присвоены дробные весовые значения. Это означает, что первая цифровая позиция справа от точки имеет весовое значение 1/2 (что есть 2^{-1}), а следующим позициям присваиваются соответственно весовые значения 1/4 (т.е. 2^{-2}), 1/8 (т.е. 2^{-3}) и т.д. Следует отметить, что описанный механизм является простым расширением изложенного выше правила — каждой цифровой позиции присваивается весовое значение, в два раза большее, чем последующей в направлении слева направо. Благодаря тому, что позиционным разрядам

присваиваются указанные выше весовые значения, процедура расшифровки двоичного представления с точкой, отделяющей дробную часть от целой, аналогична той, которая применяется к целым числам. Значение в каждом разряде просто умножается на весовое значение, присвоенное его позиции в представлении числа, а затем полученные результаты суммируются. Чтобы пояснить эту процедуру нагляднее, на рис. 1.18 показан пример расшифровки двоичного числа `101.101`, десятичное представление которого равно $5^5/_8$.

В отношении операции сложения можно сказать, что для дробных двоичных чисел используются те же методы, что и для десятичных дробей. Таким образом, чтобы сложить два дробных двоичных числа, нужно их просто выровнять по положению точки и после этого применить процедуру сложения, описанную выше. Например, после сложения чисел `10.011` и `100.11` будет получен результат `11.001`, как показано ниже:

```
  10.011
+ 100.110
  111.001
```

Аналоговые и цифровые компьютеры

В ранний период развития компьютерной техники проводились многочисленные дискуссии в отношении того, на какой основе должны создаваться будущие вычислительные устройства — на цифровой или аналоговой. В цифровых системах данные представляются с помощью строго ограниченного количества различающихся цифровых значений (например, нуль и единица). В аналоговых системах каждое значение представляется с помощью единственного устройства, способного хранить любое значение из некоторого непрерывного диапазона. Давайте сравним эти два подхода, применив в качестве примера ведра с водой. При использовании цифровой системы договоримся, что пустое ведро представляет цифру 0, а полное — 1. В результате мы можем представить любое числовое значение как ряд ведер, используя для их кодирования двоичную нотацию с плавающей точкой (см. раздел 1.7). В отличие от этого для представления чисел в аналоговой системе достаточно использовать всего лишь одно ведро, частично наполняя его водой до той отметки, которая соответствует представляемому числу. На первый взгляд, аналоговая система может показаться более точной, так как здесь отсутствуют ошибки усечения, неизбежные во всех цифровых системах (также см. раздел 1.7). Однако любой случайный толчок ведра в аналоговой системе вызовет ошибку в определении уровня воды, тогда как в цифровой системе легко можно будет отличить пустое ведро от полного, даже если из него выльется значительное количество воды. Поэтому вероятность возникновения случайных ошибок в цифровой системе существенно меньше, чем в аналоговой. Именно эта устойчивость явилась основной причиной того, что многие исходно аналоговые технологии (например, телефонная связь, аудиозаписи, телевидение) перешли на цифровую технологию.



Рис. 1.18. Декодирование двоичного числа 101.101

1.5. Вопросы и упражнения

1. Преобразуйте каждое из приведенных ниже двоичных чисел в десятичный формат.

   а. 101010 **б.** 100001 **в.** 10111 **г.** 0110 **д.** 11111

2. Преобразуйте каждое из приведенных ниже десятичных чисел в двоичный формат.

   а. 32 **б.** 64 **в.** 96 **г.** 15 **д.** 27

3. Преобразуйте каждое из приведенных ниже двоичных чисел в десятичный формат.

   а. 11.01 **б.** 101.111 **в.** 10.1 **г.** 110.011 **д.** 0.101

4. Преобразуйте каждое из приведенных ниже десятичных чисел в двоичный формат.

   а. 4½ **б.** 2¾ **в.** 1⅛ **г.** 5/16 **д.** 5⅝

5. Выполните операцию сложения для следующих двоичных чисел.

   а.
   ```
     11011
   + 1100
   ```
   б.
   ```
     1010.001
   +   1.101
   ```
   в.
   ```
     11111
   + 0001
   ```
   г.
   ```
     111.11
   + 00.01
   ```

1.6. Представление целых чисел

Математики долгое время занимались цифровыми системами счисления, и многие выдвинутые ими идеи оказались весьма полезными для разработки цифровых электронных схем. В этом разделе мы обсудим две такие системы нотации, а именно — двоичный дополнительный код и двоичную нотацию с избытком. Эти системы построены на основе двоичной системы счисления,

обсуждавшейся в разделе 1.5, и имеют определенные преимущества, благодаря которым широко используются при конструировании компьютеров. Однако эти преимущества могут обернуться и недостатками. Наша задача — уяснить свойства этих систем и понять, как они влияют на работу компьютеров.

Двоичный дополнительный код

На сегодняшний день наиболее распространенной системой представления целых чисел в компьютерах является **двоичный дополнительный код**, в котором для представления каждого числа используется фиксированное количество битов. В настоящее время данная система чаще всего реализуется с использованием 32-х двоичных разрядов для представления любого числа. Подобное решение позволяет предоставить широкий диапазон целых чисел, однако демонстрация его работы вызывает определенные затруднения. Поэтому при изучении свойств двоичного дополнительного кода мы будем оперировать числами меньшей длины.

На рис. 1.19 показаны два варианта дополнительного двоичного кода, в которых для представления чисел используются три и четыре бита соответственно. Построение подобной системы начинается с записи строки нулей, количество которых равно числу используемых двоичных разрядов. Далее ведется обычный двоичный отсчет до тех пор, пока не будет получено значение, состоящее из единственного нуля, за которым следуют лишь единицы. Полученные комбинации будут представлять положительные числа 0, 1, 2, 3, ... Для представления отрицательных чисел выполняется обратный отсчет начиная со строки из всех единиц соответствующей длины. Обратный счет продолжается до тех пор, пока не будет получена строка, состоящая из одной единицы, за которой будут следовать все нули. Полученные комбинации будут представлять числа –1, –2, –3, ... (Если вам покажется трудным вести обратный отсчет в двоичной системе счисления, можете отсчитывать комбинации в обратном порядке начиная со строки с одной единицей и всеми нулями и заканчивая строкой, состоящей из одних единиц.)

Обратите внимание, что в дополнительном двоичном коде крайний слева бит в значении каждого числа определяет знак представляемой числовой величины, поэтому этот бит принято называть **знаковым разрядом**. В двоичном дополнительном коде отрицательные числа представляются комбинациями со знаковым битом, равным `1`, а положительные числа — комбинациями со знаковым битом, равным `0`.

а) Трехразрядный дополнительный код

Двоичное представление	Десятичное представление
011	3
010	2
001	1
000	0
111	–1
110	–2
101	–3
100	–4

б) Четырехразрядный дополнительный код

Двоичное представление	Десятичное представление
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	–1
1110	–2
1101	–3
1100	–4
1011	–5
1010	–6
1001	–7
1000	–8

Рис. 1.19. Схемы кодирования в двоичном дополнительном коде

В двоичном дополнительном коде очень удобно представлена взаимосвязь между комбинациями битов, представляющими положительные и отрицательные значения, одинаковые по модулю. Последовательность битов оказывается идентичной при чтении справа налево до первой единицы включительно. С этой позиции и далее коды являются дополнительными друг другу. (**Дополнением** двоичной комбинации называется такая комбинация, которая получается в результате изменения всех нулей в исходном значении на единицы, а всех единиц — на нули. Так, двоичные комбинации `0110` и `1001` являются дополнительными друг другу.) Например, в четырехразрядном коде (рис. 1.19) обе битовые комбинации, представляющие числа 2 и –2, заканчиваются на `10`, однако комбинация, представляющая число 2, начинается с `00`, тогда как комбинация, представляющая число –2, начинается с `11`. Данное наблюдение позволяет сформулировать алгоритм взаимного преобразования битовых комбинаций, представляющих положительные и отрицательные числа, имеющие одно и то же значение по модулю. Достаточно просто копировать исходную комбинацию справа налево до тех пор, пока не будет встречена единица, а затем последовательно заменять значения оставшихся битов их дополнениями (рис. 1.20).

Ясное понимание описанных выше основных свойств двоичного дополнительного кода позволяет также сформулировать алгоритм преобразования значений этого кода в десятичное представление. Если битовая комбинация имеет нулевой знаковый бит, то это значение рассматривается просто как обычное двоичное число. Например, битовая комбинация `0110` представляет число 6,

поскольку комбинация битов 110 является двоичным представлением числа 6. Если битовая комбинация содержит знаковый бит, равный единице, то она представляет отрицательное число, и нам остается лишь определить абсолютную величину этого числа. Это выполняется посредством записи исходной комбинации справа налево, вплоть до первой встретившейся единицы, после чего для оставшихся битов в порядке их следования записываются дополнительные значения. Полученная комбинация битов дешифруется как обычное двоичное число. Например, чтобы определить десятичное значение комбинации 1010, мы, прежде всего, отмечаем, что это значение является отрицательным, так как исходная комбинация содержит единицу в знаковом бите. Затем исходная комбинация преобразуется в комбинацию 0110, которая представляет собой двоичное число 6. Теперь можно сделать окончательное заключение, что исходная двоичная комбинация представляет число –6.

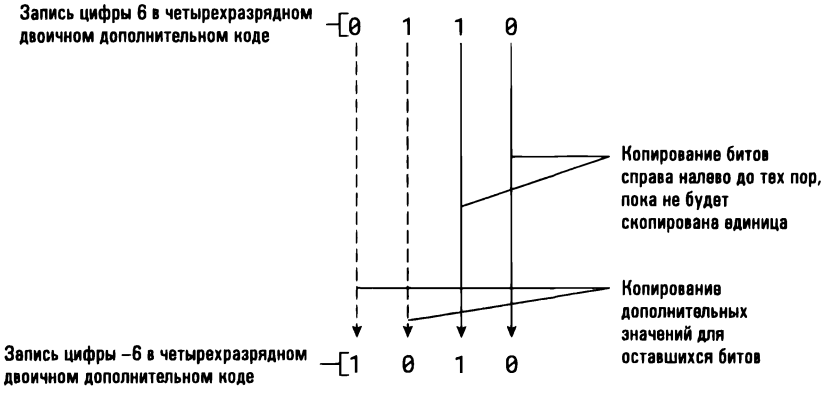


Рис. 1.20. Представление числа –6 в четырехразрядном дополнительном коде

Сложение чисел в двоичном дополнительном коде

Чтобы сложить числа, представленные в двоичном дополнительном коде, следует использовать тот же алгоритм, что и для сложения обычных двоичных чисел, за исключением того, что в этом коде все представляемые числа, включая искомый результат, имеют одинаковую длину. Это означает, что при суммировании представленных в этом коде чисел любой бит переноса, появляющийся на левом конце результирующего значения при сложении самых старших разрядов, должен отбрасываться. Например, при суммировании битовых комбинаций 0101 и 0010 будет получен результат 0111, а при сложении комбинаций 0111 и 1011 — результат 0010 (0111 + 1011 = 10010, после чего результат усекается до 0010).

Учитывая сказанное выше, рассмотрим три примера сложения, показанные на рис. 1.21. В каждом случае сначала исходные числовые значения преобразуются в четырехразрядный двоичный дополнительный код, а затем выполняется операция суммирования согласно описанному выше алгоритму. Полученный результат вновь преобразуется в десятичное значение.



Рис. 1.21. Сложение чисел в двоичном дополнительном коде

Обратите внимание: если бы при сложении использовался традиционный метод, которому нас обучали еще в начальной школе, то для решения третьей задачи потребовались бы совершенно иные действия (операция вычитания), отличные от используемых в двух предыдущих задачах. Однако за счет преобразования исходных данных в двоичные дополнительные коды можно вычислить результат с помощью одного и того же алгоритма сложения. Таким образом, основным преимуществом двоичного дополнительного кода является то, что операция сложения для любых целых чисел со знаком осуществляется с помощью одного и того же алгоритма. В отличие от учеников начальной школы, которые должны вначале освоить операцию сложения, а затем операцию вычитания, машины, в которых используется двоичный дополнительный код, должны уметь только суммировать числа и изменять знак числа на обратный.

Например, операция вычитания 7 – 5 аналогична операции сложения 7 + (–5). Следовательно, если машине потребуется вычесть число 5 (представленное битовой комбинацией `0101`) из числа 7 (представленного битовой комбинацией `0111`), то она сначала поменяет знак числа 5 на –5 (представляемое как битовая комбинация `1011`), а затем выполнит операцию сложения для значений `0111` и `1011`. В результате будет получено значение `0010`, представляющее десятичное число 2. Все это будет выглядеть следующим образом:

```
 7         0111         0111
-5   ->  - 0101   ->  + 1011
                        0010   ->  2
```

Из этого примера видно, что при использовании двоичного дополнительного кода необходимо реализовать электронные схемы только для осуществления операций сложения и отрицания. Этого будет достаточно для выполнения операций как сложения, так и вычитания. (Описание построения подобных электронных схем и объяснение принципов их работы можно найти в приложении Б.)

Проблема переполнения

В предыдущих примерах мы не упомянули об одной важной проблеме. Дело в том, что любая обсуждаемая в этой книге система представления чисел накладывает определенные ограничения на размер представляемых чисел. Например, при использовании четырехразрядного двоичного дополнительного кода самым большим представляемым положительным значением будет 7, а отрицательным значением –8, т.е. не существует битовой комбинации, представляющей, например, число 9. А это означает, что мы не можем получить верный результат при решении задачи 5 + 4. Фактически при сложении указанных чисел будет получен результат –7. Этот тип ошибок называют **переполнением**. Они возникают в том случае, когда результат операции по абсолютной величине превышает наибольшее представимое в выбранном варианте кодировки значение. При использовании двоичного дополнительного кода подобная ошибка может возникнуть при сложении двух положительных или отрицательных чисел. В любом случае ошибку можно выявить путем проверки значения знакового бита результата. Признаком переполнения является отрицательный результат сложения двух положительных чисел или положительный результат сложения двух отрицательных чисел.

Безусловно, поскольку компьютеры, использующие двоичный дополнительный код, обычно работают с гораздо более длинными битовыми комбинациями, чем те, которые были использованы в наших примерах, и ошибка переполнения, даже при обработке относительно больших чисел, возникает достаточно редко. В настоящее время для хранения чисел в двоичном дополнительном коде обычно применяются битовые комбинации длиной 32 бита, что позволяет без возникновения переполнения обрабатывать числа до 2 147 483 647. Если же требуется обработка чисел, превышающих это значение, можно использовать более длинные битовые комбинации или же просто изменить применяемую единицу измерения. Например, при обработке значений, измеренных в километрах, а не сантиметрах, можно манипулировать меньшими числами, что, возможно, позволит сохранить достаточную точность вычисления.

Из сказанного выше следует, что компьютеры также могут ошибаться. Поэтому пользователи должны знать о существовании подобной опасности. Зачастую программисты и пользователи созданных ими приложений слишком самоуверенны и забывают о том, что небольшие числовые значения могут накапливаться, давая в результате очень большие величины. Раньше для представления чисел в двоичном дополнительном коде широко использовались шестнадцатиразрядные двоичные комбинации. Это означало, что переполнение возникало при значениях, превышающих $2^{15} = 32\,768$. Например, 19 сентября 1989 года компьютер в одной из больниц выдал ошибку в расчетах, и это после долгих лет безупречной работы. Была проведена тщательная проверка, которая показала, что до наступления этой даты прошло ровно 32 768 дней, отсчитывая от 1 января 1900 года. Как вы полагаете, что оказалось причиной ошибки компьютера? Ну конечно, все дело в том, что в этой программе при вычислении дат именно начало 1900 года рассматривалось как точка отсчета, и когда 19 сентября 1989 года при переходе к следующей дате произошло переполнение, т.е. при сложении двоичного числа 32 768, представляющего эту дату, и единицы было получено отрицательное значение, программа оказалась не способной обнаружить этот факт и правильно его обработать.



Основные положения для запоминания

- **Для представления *бесконечной* математической концепции числа в компьютерах для ее представления используются *конечные* модели.**
- **Во многих языках программирования для представления символов или целых чисел используется фиксированное количество битов, что ограничивает доступный диапазон целых чисел и математических операций над ними. Это ограничение может приводить к переполнению и другим ошибкам.**
- **По причине ограниченных размеров отводимой для их представления памяти, диапазон целых чисел, которыми может оперировать программа, ограничен определенными максимальным и минимальным значениями.**

Двоичная нотация с избытком

Еще одним способом представления целых чисел является **двоичная нотация с избытком**. Как и в случае двоичного дополнительного кода, в этой нотации каждое число также представлено битовой комбинацией одной и той же длины. Чтобы сформировать представление числа в двоичной нотации с избытком, сначала выбирается длина битовой комбинации, а затем в порядке счета

в обычной двоичной системе последовательно записываются все возможные битовые комбинации, имеющие установленную длину. При анализе полученного результата можно заметить, что первая битовая комбинация с единицей в старшем разряде находится почти в середине списка. Именно она выбирается в этой нотации для представления числа 0. Все последующие комбинации с единицей в старшем разряде будут представлять числа 1, 2, 3, ... соответственно, а предыдущие комбинации в обратном направлении используются для представления чисел –1, –2, –3, ... Кодовые значения, получаемые при использовании четырехразрядных битовых комбинаций, показаны на рис. 1.22. В частности, число 5 представлено комбинацией `1101`, а число –5 представлено комбинацией `0011`. (Обратите внимание, что различие между двоичной нотацией с избытком и двоичным дополнительным кодом состоит только в противоположности значений знаковых битов.)

Комбинация битов	Представляемое значение
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	–1
0110	–2
0101	–3
0100	–4
0011	–5
0010	–6
0001	–7
0000	–8

Рис. 1.22. Значения четырехразрядных битовых комбинаций в двоичной нотации с избытком

Таблица значений, представленная на рис. 1.22, известна как двоичная нотация *с избытком восемь*. Чтобы понять, почему она так называется, сначала определим значения кодовых комбинаций как обычных двоичных значений, а затем сравним полученный результат с тем значением, которое присвоено каждой кодовой комбинации в двоичной нотации с избытком восемь. В результате мы обнаружим, что соответствующее кодовой комбинации двоичное число превышает представляемое этой комбинацией значение на 8. Например, комбинация `1100` обычно используется для представления числа 12, а в двоичной но-

тации с избытком восемь эта же комбинация представляет число 4. То же самое справедливо и для комбинации 0000, которая обычно представляет число 0, а в данной нотации — число –8. Если же двоичная нотация с избытком создается для комбинаций длиной пять битов, она будет называться двоичной нотацией с избытком 16. В этом случае комбинация 10000 будет представлять число 0, а не 16, как в обычной двоичной системе. Этот же принцип может быть использован для именования любой конкретной схемы двоичной нотации с избытком. Например, схему с тремя двоичными разрядами, представленную на рис. 1.23, можно назвать двоичной нотацией с избытком четыре.

Комбинация битов	Представляемое значение
111	3
110	2
101	1
100	0
011	–1
010	–2
001	–3
000	–4

Рис. 1.23. Значения трехразрядных битовых комбинаций в двоичной нотации с избытком 4

1.6. Вопросы и упражнения

1. Преобразуйте каждое представленное ниже значение в двоичном дополнительном коде в десятичный формат.

   а. 00011 **б.** 01111 **в.** 11100

   г. 11010 **д.** 00000 **е.** 10000

2. Преобразуйте каждое представленное ниже десятичное значение в двоичный дополнительный код длиной восемь бит.

   а. 6 **б.** –6 **в.** –17

   г. 13 **д.** –1 **е.** 0

3. Предположим, что приведенные ниже комбинации битов представляют числа в двоичном дополнительном коде. Запишите представление обратных им значений в этом же коде.

   а. 00000001 **б.** 01010101 **в.** 11111100

   г. 11111110 **д.** 00000000 **е.** 01111111

4. Предположим, что числа в машине сохраняются в двоичном дополнительном коде. Какие наибольшее и наименьшее числа могут быть записаны, если используются битовые комбинации следующей длины?

   а. четыре **б.** шесть **в.** восемь

5. В следующих задачах каждая битовая комбинация представляет число, записанное в двоичном дополнительном коде. Вычислите все операции сложения, а затем проверьте ваши результаты посредством преобразования исходных текстов задач в десятичную систему и вычисления их ответов.

   а. `0101 + 0010` **б.** `0011 + 0001` **в.** `0101 + 1010`

   г. `1110 + 0011` **д.** `1010 + 1110`

6. Решите следующие задачи с числами в двоичном дополнительном коде, однако на этот раз следите за переполнением и укажите неверные ответы, полученные в результате этой ошибки.

   а. `0100 + 0011` **б.** `0101 + 0110` **в.** `1010 + 1010`

   г. `1010 + 0111` **д.** `0111 + 0001`

7. Переведите все приведенные ниже задачи из десятичного представления в четырехразрядный двоичный дополнительный код, а затем преобразуйте их в эквивалентные задачи сложения (как это сделала бы машина) и выполните операции суммирования. Проверьте полученные ответы с помощью преобразования их в десятичное представление.

   а. 6 – (–1) **б.** 3 – (–2) **в.** 4 – 6

   г. 2 – (–4) **д.** 1 – 5

8. Может ли возникнуть ошибка переполнения при сложении двух чисел в дополнительном коде, если одно из суммируемых чисел будет положительным, а другое отрицательным? Поясните ваш ответ.

9. Преобразуйте приведенные ниже комбинации битов в двоичной нотации с избытком восемь в десятичный формат, не прибегая к помощи приведенной выше таблицы.

   а. `1110` **б.** `0111` **в.** `1000`

   г. `0010` **д.** `0000` **г.** `1001`

10. Преобразуйте приведенные ниже десятичные числа в коды двоичной нотации с избытком восемь без помощи приведенной выше таблицы.

    а. 5 **б.** –5 **в.** 3

    г. 0 **д.** 7 **г.** –8

11. Можно ли представить число 9 в двоичной нотации с избытком восемь? А что можно сказать по поводу представления числа 6 в двоичной нотации с избытком четыре? Поясните ваш ответ.

1.7. Представление дробных значений

В отличие от методов представления целых чисел, задача представления числовых значений с дробной частью требует не только сохранения комбинаций из нулей и единиц, образующих его двоичное представление, но и запоминания позиции точки, отделяющей целую часть от дробной. Наиболее распространенный способ решения этой задачи, именуемый **двоичной нотацией с плавающей точкой**, состоит в экспоненциальном представлении чисел. Конечные объемы памяти в компьютере ограничивают точность, с которой дроби могут быть представлены в компьютере, — к этой проблеме мы еще вернемся ниже в этом разделе.



Основные положения для запоминания

- **В нотации с плавающей точкой действительные числа представляются лишь с определенным приближением, которое не всегда гарантирует бесконечную точность.**

Двоичная нотация с плавающей точкой

Для пояснения принципа, положенного в основу двоичной нотации с плавающей точкой, рассмотрим пример, в котором для хранения числа используется всего один байт. Несмотря на то что в машинах обычно используются более длинные битовые комбинации, восьмиразрядный формат достаточно наглядно демонстрирует используемые принципы без ненужной избыточности длинных битовых комбинаций.

Для начала давайте условимся считать старший бит *знаковым*. Как и в предыдущих примерах, значение "нуль" в знаковом бите означает, что представляемое число неотрицательно, а значение "единица", наоборот, указывает, что

число является отрицательным. Далее разделим оставшиеся биты байта на две группы или два поля, а именно: **поле порядка** числа и **поле мантиссы**. Следующие три бита после знакового бита будем считать полем порядка числа, а оставшиеся четыре бита — полем мантиссы. Описанный выше способ разделения байта на поля представлен на рис. 1.24.

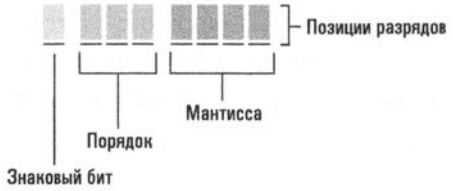


Рис. 1.24. Компоненты представления числа в двоичной нотации с плавающей точкой

Пояснить назначение отдельных полей в этом представлении можно на следующем примере. Предположим, что байт содержит следующую комбинацию битов: `01101011`. Анализируя эту комбинацию в соответствии с представленным выше форматом, мы видим, что знаковый бит имеет значение `0`, поле порядка содержит `110`, а в поле мантиссы находится значение `1011`. Для декодирования значения байта в десятичную систему прежде всего возьмем значение мантиссы и поставим слева от нее десятичную точку, как показано ниже:

```
0.1011
```

Далее, значение в поле порядка (`110`) интерпретируем как целое число, сохраненное в двоичной нотации с избытком длиной три бита (см. рис. 1.23). В этом случае значение в поле порядка представляет положительное число 2. Это значение указывает на то, что десятичную точку в значении мантиссы следует переместить вправо на две позиции. (Отрицательное значение в поле порядка говорит о том, что десятичную точку в поле мантиссы следует переместить влево.) В результате мы получим следующее значение:

```
10.11
```

Это является двоичным представлением числа $2^3/_4$. (Вспомните метод представления двоичных дробей, представленный на рис. 1.18.) Наконец, определяем, что представляемое число является положительным, поскольку знаковый бит имеет значение `0`. Таким образом, мы установили, что битовая комбинация `01101011` в двоичной нотации с плавающей точкой представляет число $2^3/_4$.

Рассмотрим еще один пример, в котором байт содержит битовую комбинацию `00111100`. Выделив мантиссу, получим следующее значение:

```
0.1100
```

Теперь перенесем плавающую точку на один бит влево, так как в поле порядка содержится значение `011`, представляющее число –1. Поэтому окончательный вид закодированного двоичного числа будет следующим:

```
0.01100
```

Это двоичное число имеет значение 3/8. Закодированное в значении байта число является неотрицательным, поскольку его знаковый бит равен `0`. Из этого следует, что битовая комбинация `00111100` в двоичной нотации с плавающей точкой представляет число 3/8.

Для представления чисел в двоичной нотации с плавающей точкой необходимо следовать описанной выше процедуре, но уже в обратном порядке. Например, для определения представления в этой нотации числа $1^1/_8$ сначала необходимо записать его двоичное представление: `1.001`. Затем эта битовая комбинация копируется в поле мантиссы слева направо, начиная с крайней слева единицы в двоичном представлении числа. Это будет выглядеть так:

```
_ _ _ _ 1 0 0 1
```

Теперь остается заполнить поле порядка числа. Представим содержимое поля мантиссы, слева от которого расположена плавающая точка, и определим число разрядов, а также направление, в котором будет перемещаться плавающая точка для получения исходного значения двоичного числа. Обратившись к нашему примеру, можно увидеть, что точка в комбинации `.1100` должна быть перемещена на один бит вправо; в результате будет получено исходное значение `1.001`. Таким образом, порядок числа равен положительному числу 1, поэтому в соответствующее поле следует поместить значение `101` (представляющее число +1 в двоичной нотации с избытком четыре; см. рис.1 23). Окончательное значение в байте будет выглядеть следующим образом:

```
0 1 0 1 1 0 0 1
```

При заполнении поля мантиссы имеется один тонкий момент, на который вы могли не обратить внимания. Правило требует копировать битовую комбинацию двоичного представления числа в поле мантиссы *слева направо*, начиная с *крайней* слева единицы. Чтобы прояснить для себя этот нюанс, рассмотрим процесс кодирования числа 3/8, двоичным представлением которого является битовая комбинация `0.011`. В этом случае мантисса должна иметь следующее значение:

```
_ _ _ _ 1 1 0 0
```

Любой другой вариант — например, представленный ниже — недопустим:

```
_ _ _ _ 0 1 1 0
```

Так происходит потому, что согласно правилам в представлении всех чисел, отличных от нуля, мантисса *всегда* должна содержать значение 1 в ее старшем разряде. Такое представление чисел называется **нормализованной формой**.

Требование соблюдения нормализованной формы исключает возможность различного представления одного и того же значения. Например, обе последовательности битов, 00111100 и 01000110, можно декодировать как число 3/8, но только первая последовательность будет его нормализованной формой. Следование правилу использования только нормализованной формы означает, что для всех чисел, отличных от нуля, в их представлении в двоичной нотации с плавающей точкой мантисса всегда будет начинаться с единицы. В связи с этим заметим, что представление числа "нуль" является особым случаем, а соответствующая битовая комбинация представляет собой строку из одних нулей.

Нотация в формате с плавающей точкой обычной точности

Вариант представления двоичных чисел в формате с плавающей точкой, предложенный в этой главе (раздел 1.7), является слишком упрощенным для того, чтобы его можно было использовать в реальных компьютерах. Посудите сами: его 8 бит позволяют представить всего 256 чисел из всего бесконечного диапазона действительных чисел. В нашем рассмотрении размер 8 бит был выбран исключительно из соображений максимального упрощения изложения и охвата при этом всех важных базовых концепций.

В большинстве современных компьютеров поддерживается формат, в котором для представления чисел используется 32 бита, — такой вариант принято называть **форматом с плавающей точкой обычной точности**. В этом формате 1 бит используется как знаковый, следующие 8 бит образуют поле порядка (в нотации с избытком), а оставшиеся 23 бита отведены для поля мантиссы. В результате в формате с плавающей точкой обычной точности можно представлять широкий диапазон значений от очень больших чисел (порядка 10^{38}) до исключительно малых (порядка 10^{-37}) с точностью до 7 значащих десятичных цифр. Последнее означает, что первые 7 цифр любого десятичного числа могут быть сохранены с очень хорошей точностью (тем не менее небольшая погрешность все же может присутствовать). А вот все остальные цифры значения, стоящие после седьмой, определенно, будут потеряны по причине ошибки усечения (хотя порядок числа при этом сохраняется).

Другой широко распространенный вариант носит название **формат с плавающей точкой двойной точности**. В этом случае для хранения числа отводится 64 бита, что обеспечивает точность представления числовых значений до 15 десятичных цифр.

Ошибки усечения

Давайте рассмотрим неприятную проблему, которая возникает при попытке представить число $2^5/_8$ в виде однобайтового кода в двоичной нотации с плавающей точкой. Прежде всего определим двоичное представление числа $2^5/_8$, которое имеет вид `10.101`. Однако при копировании этого значения в поле мантиссы имеющихся четырех разрядов оказывается недостаточно и крайняя справа единица в двоичном представлении, имеющая весовое значение 1/8, теряется (рис. 1.25). Если не обратить на это внимания и продолжить заполнение поля порядка и знакового бита числа, будет получена комбинация `01101010`, которая на самом деле представляет число $2^1/_2$, а не $2^5/_8$. Это явление называется **ошибкой усечения** или **ошибкой округления**. Оно означает, что некоторая часть кодируемого числа теряется, поскольку размер поля мантиссы оказывается недостаточным для ее представления.



Рис. 1.25. Схема кодирования числа $2^5/_8$

Основные положения для запоминания

- **Во многих языках программирования фиксированное количество битов, используемых для представления действительных чисел (в формате с плавающей точкой), ограничивает доступный диапазон значений этих чисел и математических операций. Это ограничение может служить причиной появления ошибок округления и ошибок других типов.**

Во избежание подобных ошибок можно использовать поле мантиссы большего размера. Как и в случае целых чисел, для представления значений в нотации с плавающей точкой принято использовать комбинации не менее 32 бит, а не 8 бит, как в приведенных выше примерах. Одновременно это позволяет расширить и размер поля порядка числа. Но даже при использовании более длинных полей достигаемая точность представления числовых значений в некоторых случаях оказывается недостаточной.

Существует еще одна причина появления ошибок усечения, с которой каждый из нас уже встречался при изучении десятичной системы счисления. Это проблема бесконечного количества дробных знаков в представлении числа, которая встречается, например, при выражении числа 1/3 в виде десятичной дроби. Дело в том, что некоторые числа невозможно точно выразить, сколько бы цифр не использовалось для их представления. Различие между традиционной десятичной системой счисления и двоичной системой состоит в том, что в двоичной системе больше чисел имеют бесконечное представление, чем в десятичной. Например, даже такое число, как одна десятая, имеет в двоичной системе бесконечное представление. Попробуйте представить себе, какие могут возникнуть проблемы, если какой-либо неосторожный человек решит использовать двоичные числа с плавающей точкой для хранения и обработки данных, представляющих собой суммы в долларах и центах. Например, если единицей измерения данных является доллар, то оказывается невозможным точно представить даже обычную десятицентовую монету. Хорошее решение в подобном случае — измерять данные в единицах центов, тогда все значения окажутся целыми числами, и их можно будет представить с помощью двоичного дополнительного кода.

Ошибки усечения и все связанные с ними проблемы являются предметом изучения тех, кто работает в области численного анализа. Эта отрасль математики исследует проблемы, связанные с реальным выполнением вычислений, отличающихся очень большим объемом и требующих повышенной точности результатов.

Ниже приведен пример, который порадует душу любого специалиста по численному анализу. Предположим, что нужно сложить три следующие числа, представленных в однобайтовых кодах двоичной нотации с плавающей точкой (описанной выше):

$2^1/_2 + {}^1/_8 + {}^1/_8$

Если суммировать эти числа в указанном порядке, то сначала будет получено промежуточное значение $2^5/_8$ (в результате сложения чисел $2^1/_2$ и $^1/_8$), двоичным представлением которого является битовая комбинация `10.101`. К сожалению, это число не может быть представлено точно (в чем мы убедились

раньше), поэтому в результате сложения будет получено число $2^1/_2$ (т.е. первое из слагаемых). Если теперь прибавить к полученному результату следующее число 1/8, то опять возникнет та же ошибка усечения и вновь будет получен тот же неверный ответ — $2^1/_2$.

А теперь попробуем сложить те же числа, но в обратном порядке. Сначала сложим числа 1/8 и 1/8, в результате чего получим число 1/4, двоичным представлением которого является битовая комбинация `0.01`; соответствующий байт результата будет иметь вид `00111000`, отражающий точное значение. Теперь прибавим число 1/4 к следующему числу в списке, $2^1/_2$. В результате будет получено правильное значение $2^3/_4$, которое может быть точно представлено в байте в виде кода `01101011`. На этот раз мы получили правильный ответ.

Можно сделать заключение, что при сложении чисел в двоичной нотации с плавающей точкой большое значение может иметь порядок, в котором они суммируются. Вся проблема состоит в том, что если очень большое число прибавить к очень маленькому, то маленькое число может быть утеряно в результате усечения. Поэтому общее правило суммирования большого количества чисел требует начинать операцию сложения с самых малых чисел в предположении, что в результате будет получено достаточно большое промежуточное значение, которое затем можно будет безопасно сложить с оставшимися большими числами. В противном случае велика вероятность столкнуться с описанной выше проблемой.

Разработчики современного коммерческого программного обеспечения прилагают значительные усилия, чтобы оградить пользователя от подобных проблем. В типичном приложении электронных таблиц корректные результаты могут быть достигнуты, если различия между суммируемыми значениями не превосходят 10^{16} или меньше. Поэтому, если потребуется добавить единицу к числу

10 000 000 000 000 000

то велика вероятность, что будет получен ответ

10 000 000 000 000 000

вместо предполагаемого значения

10 000 000 000 000 001

Такие проблемы имеют очень большое значение в приложениях, подобных навигационным системам, в которых даже малейшие ошибки могут накапливаться в ходе последующих вычислений, что может привести к весьма серьезным последствиям. Что же касается рядовых пользователей ПК, то степень точности вычислений, обеспечиваемая большинством коммерческого программного обеспечения, оказывается вполне достаточной для их нужд.

1.7. Вопросы и упражнения

1. Декодируйте приведенные ниже битовые комбинации в формате с плавающей точкой, описанном в этом разделе.

   а. `01001010` **б.** `01101101` **в.** `00111001`

   г. `11011100` **д.** `10101011`

2. Представьте приведенные ниже числа в формате с плавающей точкой, описанном выше в этом разделе. Укажите на случаи появления ошибок усечения.

   а. $2^{3}/_{4}$ **б.** $5^{1}/_{4}$ **в.** $^{3}/_{4}$ **г.** $-3^{1}/_{2}$ **д.** $-4^{3}/_{8}$

3. При использовании формата с плавающей точкой, описанного выше в этом разделе, какая из битовых комбинаций, `01001001` или `00111101`, представляет большее числовое значение? Опишите простейшую процедуру определения, какое из двух представленных в этом формате чисел является б*о*льшим.

4. Какое наибольшее число может быть представлено в формате с плавающей точкой, описанном выше в этом разделе? Какое наименьшее положительное число может быть представлено в этой системе?

1.8. Данные и программирование

Пока человечество было занято разработкой тех методов представления данных и основных операций, которые реализованы в современных компьютерах, лишь немногие могли успешно работать с компьютерами непосредственно на этом, самом низком уровне — уровне собственно машинных команд. Как правило, люди предпочитают описывать вычислительные задачи на достаточно высоком уровне абстракции, а в отношении соответствующих манипуляций на нижних уровнях детализации они охотно полагаются на компьютеры. Язык программирования представляет собой компьютерную систему, созданную с единственной целью: предоставить людям возможность точно представлять компьютерам требуемые алгоритмы с использованием достаточно высокого уровня абстракции.

В XX веке написание компьютерных программ рассматривалось как область деятельности для относительно немногих высококвалифицированных специалистов. И это неудивительно: в вычислительных системах того времени было еще много проблем, требовавших внимания опытных компьютерных специа-

листов и инженеров-программистов. Однако к началу XXI века, когда компьютеры и компьютерные вычисления уже достаточно широко и глубоко проникли во все аспекты современной жизни, становилось все труднее найти такую область человеческой деятельности, в которой от специалиста не требовалось иметь хотя бы минимальные навыки в области программирования. И действительно, уже имели место попытки определить программирование или *кодирование* как следующее фундаментальное умение современного человека наряду с умением читать, писать и считать.

Из этого раздела, а также из последующих разделов других глав, посвященных программированию, вы узнаете, как язык программирования позволяет выражать основные идеи той или иной главы, предоставляя людям возможность проще решать любые задачи, связанные с вычислениями.

Знакомимся с языком Python

Язык Python — это язык программирования, созданный Гвидо ван Россумом (Guido van Rossum) в конце 1980-х годов. Сегодня он входит в десятку языков, получивших наибольшее распространение, является популярным инструментом при разработке веб-приложений и выполнении научных расчетов и часто используется как вводный язык для студентов. Диапазон организаций, использующих Python для своих целей, простирается от корпорации Google до агентства NASA, от компании DropBox до компании Industrial Light & Magic, охватывая весь спектр пользователей компьютеров — разработчиков, исследователей, ученых-компьютерщиков и специалистов по компьютерной графике. Язык Python ориентирован на высокую читаемость кода и поддерживает парадигмы императивного, объектно-ориентированного и функционального программирования, которые подробно обсуждаются в главе 6.

Программное обеспечение, предназначенное для написания и выполнения программ на языке Python, является бесплатным и доступно всем желающим на сайте `www.python.org` наряду с множеством других полезных ресурсов для начинающих. Следует отметить, что с течением времени язык Python активно развивался и продолжает свое развитие в настоящее время. Во всех примерах в этой книге используется версия языка, получившая название "Python 3". Более ранние версии языка Python способны выполнять весьма близкие версии предлагаемых программ, однако в них следует внести много мелких изменений, скажем, в отношении пунктуации, которая изменилась в сравнении с версией языка Python 2.

Отметим, что Python является *интерпретирующим языком*, и для начинающих это означает, что инструкции Python могут либо вводиться непосредственно в командную строку, либо быть предварительно записанными в простой

текстовый файл (называемый "сценарием"), который затем запускается на выполнение. В приведенных ниже примерах может использоваться любой из этих вариантов, но в случае упражнений и заданий в конце главы обычно предполагается использование сценариев на языке Python.

Программа Hello Python

По устоявшейся традиции первой программой, описываемой во множестве введений в различные языки программирования, является программа "Hello, World" (Здравствуй, Мир). Эта простая программа выводит указанное приветствие, демонстрируя, как в том или ином языке достигается этот результат, а также как в этом языке представляются текстовые данные. В языке Python[2] эта программа записывается в виде

```
print('Hello, World!')
```

Введите этот оператор в командной строке интерактивного интерпретатора языка Python либо поместите ее в файл сценария, а затем выполните его. В любом случае результат будет следующим:

```
Hello, World!
```

Как видите, эта программа на языке Python возвращает пользователю текст, размещенный между одинарными кавычками.

Есть несколько важных аспектов, на которые следует обратить внимание даже в этом простейшем сценарии на языке Python. Во-первых, выражение `print()` является встроенной функцией, заранее определенным оператором, который в сценариях Python можно использовать для осуществления *вывода*, т.е. такого действия, когда результаты работы программы становятся видимы пользователю. За словом `print` следуют открывающая и закрывающая скобки, между которыми и находится то значение, которое будет выведено (отпечатано) в результате выполнения оператора.

Во-вторых, в языке Python текстовые строки задаются с использованием одинарных кавычек. В данном случае кавычки перед прописной буквой `H` и после восклицательного знака определяют начало и окончание строки символов, которые в языке Python будут восприняты как текстовое значение.

В языках программирования все поступающие инструкции выполняются исключительно точно. Если пользователь в операторе вывода внесет даже незначительные изменения в сообщение между начальной и конечной кавычками,

[2] Приведенный здесь код на языке Python предназначен для выполнения в среде языка версии 3. Во всех последующих примерах в книге это подразумевается при любых ссылках в виде "Python", без указания версии. В более ранних версиях языка открывающая и закрывающая скобки требуются не всегда.

выведенный текст изменится в точном соответствии с внесенными изменениями. Выберите время и попробуйте сделать какие-либо изменения в исходном операторе вывода (в использовании прописных букв, пунктуации или вообще измените слова в этом операторе) — и вы убедитесь, что это на самом деле так.

Переменные

В языке Python пользователю разрешается присваивать величинам имена, для удобства обращения к ним в дальнейшем, что является важной абстракцией для построения компактных, легких для понимания сценариев. Такие именованные местоположения в памяти называют *переменными* по аналогии с математическими переменными, широко используемыми в курсах алгебры. Рассмотрим несколько измененную версию программы "Hello World", приведенную ниже.

```
message = 'Hello, World!'
print(message)
```

В этом сценарии первая строка представляет собой *оператор присваивания*. Символ `'='` иногда может сбивать с толку начинающих, привыкших к алгебраическому использованию знака равенства. Этот оператор присваивания следует читать так: "Переменной `message` присваивается строковое значение `'Hello, World!'`". В общем случае оператор присваивания будет содержать имя переменной слева от знака равенства и присваиваемое ей значение — справа от него.

Python является языком с *динамической типизацией*, а это означает, что в нашем сценарии нет необходимости заранее определять, какой именно тип данных будет храниться в переменной с именем `message` или значение какого типа будет выведено в сообщении. В этом сценарии вполне достаточно указать, что переменной `message` в качестве значения присваивается строка текста, а затем просто сослаться на переменную `message` в следующем операторе `print()`.

Именование переменных в языке Python в максимальной степени возлагается на пользователя. Простые правила Python лишь указывают, что имя переменной должно начинаться с буквы и может содержать произвольное количество букв, цифр и знаков подчеркивания `'_'`. Хотя назначить переменной имя `m` может быть вполне резонным решением для двух строк кода, опытные программисты в своих программах стараются выбирать используемым переменным осмысленные, описательные имена.

В языке Python имена переменных *чувствительны к регистру*, т.е. строчное и прописное написание одной и той же буквы различаются. В результате переменная с именем `size` воспринимается как отличная от переменных с именами `Size` или `SIZE`. Небольшое количество **ключевых слов**, т.е. имен, которые в

языке Python зарезервированы для специальных целей, запрещается использовать в качестве имен переменных. Ознакомиться со списком ключевых слов можно во встроенной справочной системе Python. Для доступа к этому списку в командной строке интерпретатора языка достаточно ввести команду

```
help('keywords')
```

Переменные могут использоваться для хранения данных любых типов, которыми язык Python может манипулировать, например

```
my_integer = 5            — целые числа
my_floating_point = 26.2  — числа с плавающей точкой
my_Boolean = True         — логические (булевы) значения
my_string = 'characters'  — строка символов (текст)
```

Обратите внимание, что все упомянутые здесь типы данных прямо соответствуют тем представлениям двоичных данных, которые рассматривались в этой главе: логические (булевы) значения "ложь" и "истина" (раздел 1.1), текст (раздел 1.4), целые числа (раздел 1.6) и числа с плавающей точкой (раздел 1.7). В других программах на языке Python (вне рамок этого простого введения в данном курсе) вы также сможете сохранять в переменных языка Python изображения и аудиоданные (раздел 1.4).

В языке Python шестнадцатеричные значения записываются с префиксом `0x`, например

```
my_integer = 0xFF
print(my_integer)
```

Представление значения в шестнадцатеричной нотации не оказывает влияния на представление этого значения в памяти компьютера, в которой целочисленные значения всегда сохраняются в виде последовательности единиц и нулей независимо от основания системы счисления, которой воспользовался программист при его записи. Шестнадцатеричные значения являются удобным вариантом записи в тех ситуациях, когда это представление может помочь в понимании сценария. В приведенном выше примере оператор `print(my_integer)` выведет `'255'`, что является представлением шестнадцатеричного значения `0xFF` в десятичной системе счисления, — такое поведение принято для оператора `print()` по умолчанию. Более сложные варианты записи оператора `print()` можно использовать для вывода значений и в других представлениях, но здесь мы ограничим наше обсуждение лишь более знакомым десятичным представлением.

Символы Unicode, включая те, которые не входят в базовое подмножество кодировки ASCII, могут быть включены непосредственно в строку, если используемый текстовый редактор позволяет это сделать.

```
print('₹1000')  # Печатается ₹1000, что представляет одну тысячу
                   индийских рупий
```

Если текстовый редактор не позволяет вставлять такие символы, необходимый символ может быть определен с использованием четырех шестнадцатеричных цифр, следующих за префиксом '\u'.

```
print('\u00A31000')  # Печатается £1000, что представляет одну
                       тысячу британских фунтов стерлингов
```

Первая часть строки `\u00A3` кодирует символ £, принятый для обозначения британских фунтов стерлингов. То, что четыре символа `1000` следуют непосредственно за первой частью, указывает на то, что между указанным знаком валюты и числовым значением не должно быть пробела, т.е. вывод должен выглядеть так: `£1000`.

Приведенные выше примеры, помимо строк с использованием символов Unicode, также знакомят нас с еще одной возможностью языка Python. В строках кода символ # отмечает начало поля *комментариев*, т.е понятных человеку замечаний и указаний, помещенных непосредственно в текст сценария, которые компьютер будет игнорировать при его выполнении. Опытные программисты используют комментарии в своих кодах для объяснения работы сложных фрагментов алгоритма, для внесения исторических сведений или информации об авторстве или просто для указания тех моментов, на которые следует обратить внимание при чтении программы. Тщательно продуманное краткое описание в начале сценария обычно знакомит читателя с общим назначением сценария и основными принципами его работы. Согласно общему правилу все символы справа от символа # и до конца строки просто игнорируются при выполнении сценария на языке Python.



Основные положения для запоминания

- **Продуманное описание программы поможет людям понять ее функциональные возможности и назначение.**

Операторы и выражения

Встроенные операторы языка Python позволяют манипулировать числовыми значениями и комбинировать их разными знакомыми нам способами.

```
print(3 + 4)    # Печатает "7",  т.е. сумму 3 плюс 4
print(5 - 6)    # Печатает "-1", т.е. разность 5 минус 6
```

```
print(7 * 8)     # Печатает "56",     т.е. произведение 7 на 8
print(45 / 4)    # Печатает "11.25", т.е. частное от деления 45 на 4
print(2 ** 10)   # Печатает "1024",   т.е. 2 в степени 10
```



Основные положения для запоминания

- **Использование в программе целых чисел или чисел с плавающей запятой не требует понимания того, как они представлены в памяти машины.**
- **Числа и концепции их обработки являются фундаментальными понятиями в программировании.**
- **Математические выражения с использованием арифметических операций являются частью большинства языков программирования.**

Когда выполнение арифметического оператора, требующего, например, разделить 45 на 4, приводит к получению нецелочисленного результата, например 11,25, интерпретатор языка Python автоматически преобразует его в число с плавающей точкой. Если требуется получить строго целочисленный результат, для этой цели можно использовать другие операторы.

```
print(45 // 4)  # Печатает "11", целую часть частного от
                  деления 45 на 4
print(45 % 4)   # Печатает "1", остаток от деления нацело
                  45 на 4, т.е. 4 * 11 + 1 = 45
```

Двойная косая черта определяет операцию *получения целой части от деления*, а символ процента — операцию *получения остатка* от деления. Собрав все вместе, можно сказать, что "число сорок пять можно представить как сумму одиннадцати четверок плюс единица". В предыдущем примере использование ** определяет операцию возведения в степень, которая в некоторых других языках программирования может быть представлена символом "^". Однако в языке Python этот символ принадлежит к группе *побитовых булевых операторов*, речь о которых пойдет в следующей главе.

Строки также можно комбинировать и манипулировать ими следующими интуитивно понятными способами.

```
s = 'hello' + 'world'
t = s * 4
print(t)  # Печатает "helloworldhelloworldhelloworldhelloworld"
```

Оператор "плюс" реализует *конкатенацию* (слияние) строковых значений, а оператор умножения вызывает их *репликацию* (повторение).



Основные положения для запоминания

- **Строки и строковые операторы, включая конкатенацию и некоторые формы выделения подстрок, часто используются во многих программах.**

Многозначность некоторых встроенных операторов иногда может приводить к ошибкам. Так, выполнение следующего сценария завершится сообщением об ошибке.

```
print('USD$' + 1000)  # Печатает "TypeError: Can't convert
                         'int' to str implicitly"
```

Это сообщение об ошибке говорит о том, что операция конкатенации строк не может быть выполнена, поскольку второй ее операнд не является строкой символов. К счастью, язык Python предоставляет функции, позволяющие выполнить преобразование значения одного типа в другой тип. Так, функция `int()` позволяет преобразовать значение с плавающей точкой в целочисленное, при этом дробная часть числа отбрасывается. Также можно преобразовать строку из цифровых символов в целочисленное представление, при условии, что эта строка корректно отображает допустимое числовое значение. Аналогичным образом функция `str()` позволяет преобразовать числовое значение в текстовую строку, состоящую из символов в кодировке UTF-8. Таким образом, следующий модифицированный вариант предыдущего оператора `print()` уже не вызовет появления сообщения об ошибке.

```
print('USD$' + str(1000))     # Печатает "USD$1000"
```

Программа конвертирования валюты

Приведенный ниже полноценный сценарий на языке Python иллюстрирует многие концепции, представленные выше в этом разделе. Оперируя заданной суммой в долларах США, сценарий осуществляет конвертирование этой заданной суммы в четыре другие валюты.

```
# Конвертер суммы в долларах в другие валюты.
USD_to_GBP = 0.76   # Курс обмена долларов США на фунты стерлингов
USD_to_EUR = 0.86   # Курс обмена долларов США на евро
USD_to_JPY = 114.08 # Курс обмена долларов США на японские йены
USD_to_INR = 63.64  # Курс обмена долларов США на индийские рупии
GBP_sign = '\u00A3'  # Коды символов валюты в Unicode
EUR_sign = '\u20AC'
JPY_sign = '\u00A5'
INR_sign = '\u20B9'
```

```
dollars = 1000      # Сумма конвертируемых долларов
pounds = dollars * USD_to_GBP    # Конвертирование
euros = dollars * USD_to_EUR
yen = dollars * USD_to_JPY
rupees = dollars * USD_to_INR
print('Today, $' + str(dollars)) # Вывод результатов
print('converts to ' + GBP_sign + str(pounds))
print('converts to ' + EUR_sign + str(euros))
print('converts to ' + JPY_sign + str(yen))
print('converts to ' + INR_sign + str(rupees))
```

При выполнении этот сценарий выведет следующее:

```
Today, $1000
converts to £760.0
converts to €860.0
converts to ¥114080.0
converts to ₹63640.0
```

Отладка

Языки программирования не склонны прощать ошибки начинающим, поэтому значительную часть времени, потраченного на обучение написанию программ, им приходится проводить в попытках найти **баги** (bugs — *жучки*), — так на жаргоне программистов принято называть ошибки в программном коде. Обнаружение таких багов и их исправление называют **отладкой** (debugging). Существуют три основные категории ошибок, которые обычно допускают при написании программного кода: **синтаксические ошибки** (ошибки в символах, которые были случайно сделаны при вводе текста программы), **семантические ошибки** (ошибки, связанные с неверным использованием операторов, — смысловые ошибки в программах) и **ошибки времени выполнения** (ошибки, которые по разным причинам возникают при выполнении программы).



Основные положения для запоминания

- **Обнаружение и исправление ошибок в программе называют отладкой программы.**

Для новичков наиболее характерны синтаксические ошибки, такие как пропуск по забывчивости одной из кавычек в начале или конце текстовой строки, не закрытые по невнимательности открывающие скобки или ошибки в написании ключевых слов, например синтаксическая ошибка в имени функции `print()`.

Интерпретатор языка Python, как правило, всегда предпринимает попытки указать на подобные ошибки в случае их обнаружения, отображая номер строки кода с ошибкой и ее краткое описание. При достаточной практике новичок, как правило, быстро учится распознавать и правильно интерпретировать типичные сообщения об ошибках. Рассмотрим следующий пример.

```
print(5 + )
SyntaxError: invalid syntax
```

Здесь в выражении в скобках отсутствует второй операнд операции сложения: после знака "+" сразу же стоит закрывающая скобка.

```
print(5.e)
SyntaxError: invalid token
```

В этом операторе интерпретатор Python ожидает, что за десятичной точкой будет следовать цифра, а не буква.

```
pront(5)
NameError: name 'pront' is not defined
```

В данном случае (как если обратиться к кому-то, неправильно произнеся его фамилию) ошибочное написание имени известной функции или переменной способно привести к путанице и затруднениям в интерпретации программы.

Семантические ошибки — это ошибки в алгоритме или ошибки в том, как алгоритм был представлен средствами языка. Примером такой ошибки может быть использование в вычислениях имени не той переменной или неверный порядок выполнения математических операций в сложных выражениях. В языке Python всегда предполагаются стандартные правила в отношении порядка выполнения математических операторов, поэтому в выражении `total_pay = 40 + extra_hours * pay_rate` умножение будет выполнено *перед* сложением, что приведет к неправильному вычислению общей суммы выплаты. (Исключением будет только тот случай, если вам платят по одному доллару в час.) Для точного определения порядка вычислений в выражениях следует использовать скобки, что не только позволит избежать семантических ошибок, но и значительно улучшит читабельность кода (т.е. правильная запись в этом случае — `total_pay = (40 + extra_hours) * pay_rate`).

И наконец, ошибки времени выполнения на этом уровне могут включать непреднамеренное деление на нуль или использование переменной до того, как она будет определена, т.е. до того, как ей будет присвоено требуемое (или вообще какое-либо) значение. Интерпретатор Python считывает операторы от начала сценария к концу, и он должен обработать оператор присвоения переменной значения до того, как эта переменная будет использована в каком-либо выражении.

Тестирование является неотъемлемой частью написания сценария на языке Python, — а в действительности и абсолютно любой другой программы. Методическое тестирование скриптов требует, прежде всего, ясного понимания того, что он, как ожидается, должен делать. Запускайте свои сценарии как можно чаще по мере их написания, вплоть до того, что выполняйте их после завершения ввода каждой строки кода. Понимание, что сценарий в конечном счете должен делать, поможет вам немедленно найти и устранить синтаксические ошибки, а значит, сфокусироваться на том, что должно происходить на каждом этапе его выполнения.



Основные положения для запоминания

- **Знание того, что программа должна делать, а чего не должна, необходимо для того, чтобы найти бо́льшую часть имеющихся в ней ошибок.**

1.8. Вопросы и упражнения

1. Что делает Python интерпретируемым языком программирования?
2. Напишите операторы языка Python, которые будут выводить следующее.
   а. Слова "Computer Science Rocks", за которыми будет следовать восклицательный знак.
   б. Число 42.
   в. Приблизительное значение числа p с точностью до четырех десятичных знаков.
3. Напишите операторы языка Python, в которых переменным буду присвоены следующие значения.
   а. Слово "programmer" переменной с именем `rockstar`.
   б. Количество секунд в часе переменной с именем `seconds_per_hour`.
   в. Средняя температура тела человека переменной с именем `bodyTemp`.
4. Напишите оператор языка Python, который из существующей переменной `bodyTemp` будет извлекать значение в градусах Фаренгейта и записывать его эквивалент в градусах Цельсия в новую переменную с именем `metricBodyTemp`.

1.9. Сжатие данных

При записи или передаче данных часто полезно (а иногда просто необходимо) сократить размер обрабатываемых данных, сохранив при этом ту информацию, которая в них закодирована. Технология, позволяющая достичь этой цели, называется **сжатием данных**. В этом разделе мы сначала рассмотрим некоторые общие методы сжатия данных, а затем обсудим несколько конкретных приемов, разработанных специально для определенных вариантов использования.

Универсальные методы сжатия данных

Все существующие методы сжатия данных можно разделить на две категории. Первые обеспечивают сжатие **без потерь**, тогда как во вторых сжатие всегда происходит **с потерями**, в той или иной степени. К методам сжатия без потерь относятся такие, которые полностью исключают потери информации в процессе сжатия. И наоборот, в методах сжатия с потерями при сжатии неизбежна утрата исходной информации в той или иной степени. Однако методы с потерями могут обеспечивать более высокий уровень сжатия, чем методы без потерь, и это делает их достаточно привлекательными в тех случаях, когда наличие незначительных ошибок можно считать приемлемым, например при сохранении изображений или аудиозаписей.



Основные положения для запоминания

- **В использовании для хранения и передачи данных методов сжатия с потерями и без потерь существуют определенные компромиссы.**

Если сжимаемые данные состоят из длинных последовательностей одних и тех же значений, наилучшие результаты дает **метод кодирования длины серий**, обеспечивающий сжатие без потерь. Этот метод кодирования состоит в замене последовательностей идентичных данных кодовым значением, определяющим само повторяющееся значение и количество его повторений в данной серии. Например, для записи кодированной информации о том, что битовая последовательность состоит из 253 единиц, за которыми следуют 118 нулей и еще 87 единиц, потребуется существенно меньше места, чем для перечисления всех этих 458 бит.

Еще один метод сжатия данных без потерь предполагает применение **частотно-зависимого кодирования**, при котором длина битовой комбинации,

представляющей элемент данных, обратно пропорциональна частоте использования этого элемента. Такие коды входят в группу кодов переменной длины, т.е. элементы данных в этих кодах представляются битовыми комбинациями различной длины. Построение алгоритма, который обычно используется при разработке частотно-зависимых кодов, приписывают Дэвиду Хоффману (David Huffman), поэтому такие коды часто называются **кодами Хоффмана**. Большинство используемых сегодня частотно-зависимых кодов является кодами Хоффмана

В качестве примера частотно-зависимого кодирования обратимся к задаче кодирования текста на английском языке. В этом языке буквы *e*, *t*, *a* и *i* встречаются гораздо чаще, чем буквы *z*, *q* и *x*. Поэтому при разработке частотно-зависимого кода для этой задачи целесообразно будет чаще всего встречающиеся символы (*e*, *t*, *a* и *i*) представить самыми короткими битовыми комбинациями, а реже всего встречающиеся символы (*z*, *q* и *x*) — самыми длинными битовыми комбинациями. В результате мы получим более компактное представление всего текста, чем при использовании обычного кода, подобного ASCII.

В некоторых случаях поток сжимаемых данных может состоять из блоков данных, каждый из которых лишь немного отличается от предыдущего. Примером могут служить последовательные кадры видеоизображения. Для таких случаев используется метод **относительного кодирования**. Данный подход предполагает запись отличий, существующих между последовательными блоками данных, вместо записи самих этих блоков, т.е. каждый блок кодируется с точки зрения его взаимосвязи с предыдущим блоком. Относительное кодирование может осуществляться как с потерями информации, так и без потерь, в зависимости от того, как именно фиксируются различия между последовательными блоками — точно или приблизительно.

Совершенно иные способы сжатия используются в **методах словарного кодирования**. В данном контексте термин *словарь* означает набор строительных блоков, из которых создается сжатое сообщение, а само сообщение кодируется как последовательность ссылок на этот словарь. Обычно методы словарного кодирования относят к группе методов сжатия без потерь, но, как будет показано ниже, при обсуждении методов сжатия изображений, могут иметь место варианты, когда элементы в словаре могут представлять собой только приблизительное представление точных значений элементов данных, вследствие чего сжатие в данном случае будет осуществляться с потерями.

Словарное кодирование может применяться для сжатия текстовых документов в текстовых процессорах, поскольку встроенные словари, которые эти приложения уже содержат с целью поддержки функции проверки грамматики, могут служить превосходными словарями и для целей сжатия. В частности, все слово может быть закодировано и сохранено как единственная ссылка на этот

словарь, а не как последовательность составляющих его отдельных символов, закодированных в системе, подобной UTF-8. Типичный словарь в текстовом процессоре содержит приблизительно 25 тысяч элементов, а это означает, что каждый элемент может быть представлен целым числом в диапазоне от 0 до 24 999, т.е. для представления кода каждого элемента словаря будет достаточно всего 15 бит. В противоположность этому, если слово, на которое имеется ссылка в словаре, состоит из шести букв, то его представление в качестве последовательности символов в кодировке UTF-8 потребует 48 бит.

Вариантом метода словарного кодирования является **кодирование с применением адаптивного словаря** (этот подход также известен как *метод динамического словарного кодирования*). В этом методе содержание словаря разрешается изменять непосредственно в процессе кодирования. Популярным примером методов динамического кодирования является метод **Lempel-Ziv-Welsh (LZW)**, названный в честь его создателей, Абрахама Лемпеля (Abraham Lempel), Джэкоба Зива (Jacob Ziv) и Терри Уэлча (Terry Welch). При кодировании сообщения с помощью метода LZW работа начинается с использованием словаря, содержащего лишь базовые строительные блоки, из которых построено данное сообщение. Однако, как только в сообщении будет найдена более крупная единица, она добавится в словарь — в том смысле, что все последующие вхождения этой единицы в тексте могут быть закодированы единственной, а не множеством ссылок на словарь. Например, при кодировании текста на английском языке словарь в начале работы может содержать отдельные буквы, цифры и знаки препинания. Но как только в сообщении будут идентифицированы отдельные слова, они смогут быть добавлены в словарь. В результате по ходу кодирования сообщения словарь будет увеличиваться в размерах, и по мере его роста все больше слов (или даже повторяющихся последовательностей слов) в сообщении можно будет закодировать одной ссылкой на словарь.

Итак, может быть получено сообщение, закодированное в терминах относительно большого словаря, который будет уникальным для данного конкретного сообщения. Однако для декодирования данного сообщения иметь этот большой словарь вовсе необязательно, достаточно воспользоваться малым *исходным* словарем. Действительно, процесс декодирования может начаться с того же самого малого словаря, который использовался в начале кодирования сообщения. Далее, по ходу процесса декодирования могут быть обнаружены те же единицы, которые были найдены в процессе кодирования, а значит, их можно будет добавить в словарь и использовать в процессе дальнейшего декодирования, как это было сделано при кодировании данного сообщения.

Чтобы излагаемый материал стал понятнее, давайте детально рассмотрим применение метода LZW для кодирования следующего сообщения:

```
xyx xyx xyx xyx
```

Начнем с работы со словарем, содержащим три значения: первым будет *x*, вторым — *y*, а третьим — пробел. Закодируем последовательность *xyx* как `121`, что следует понимать таким образом: сообщение начинается с последовательности из трех блоков, состоящей из первого элемента словаря, второго элемента словаря и вновь первого элемента словаря. Далее кодируется пробел — и мы получаем последовательность `1213`. Однако, встретив в сообщении пробел, мы понимаем, что предшествующая строка представляет собой слово, поэтому добавляем сочетание *xyx* в словарь как новый, четвертый элемент. Действуя дальше аналогичным образом, получаем закодированное сообщение в виде `121343434`.

Если теперь возникнет необходимость декодировать сообщение, то работа будет начата с исходного варианта словаря с тремя элементами. Сначала первые четыре символа `1213` будут декодированы как *xyx* с пробелом на конце. Наличие в результирующей строке пробела указывает, что первые три символа образуют слово *xyx*, которое следует добавить в словарь четвертым элементом, точно так, как это было сделано при кодировании сообщения. Продолжая декодирование и принимая во внимание, что очередной код 4 в закодированном сообщении ссылается на новый четвертый элемент в словаре, а значит, к уже имеющемуся декодированному тексту следует добавить слово *xyx*, получаем новый вариант декодированной последовательности:

```
xyx xyx
```

Продолжив декодирование в той же манере, в конечном счете мы установим, что закодированное сообщение `121343434` соответствует исходному сообщению `xyx xyx xyx xyx`, что в точности совпадает с оригиналом.



Основные положения для запоминания

- **Сжатие данных без потерь уменьшает количество битов, требуемое для их хранения или передачи, но позволяет впоследствии полностью восстановить данные в их исходном виде.**
- **Сжатие данных с потерями позволяет существенно сократить количество битов, требуемое для их хранения или передачи, но при этом позволяет восстановить лишь приблизительную (в той или иной степени) версию их исходного состояния.**

Сжатие изображений

В разделе 1.4 было показано, как изображения могут быть закодированы с использованием методов битовой карты. К сожалению, такой подход часто при-

водит к созданию очень больших файлов, что весьма характерно для растровых изображений. По этой причине специально для этого формата было разработано множество схем сжатия, предназначенных для уменьшения места, занимаемого файлами изображений на диске.

Одной из таких схем является формат **GIF** (Graphic Interchange Format), разработанный компанией CompuServe. Используемый в ней метод заключается в уменьшении количества цветовых оттенков пикселя до 256, в результате чего цвет каждого пикселя может быть представлен одним байтом вместо трех. Комбинация красного, зеленого и синего цветов для каждого из этих 256 оттенков сохраняется в таблице (словаре), называемой палитрой. В результате каждый пиксель в изображении может быть представлен единственным байтом, значение которого указывает на требуемый цвет в палитре. (Вспомните, что один байт позволяет представить одно значение из 256 различных битовых последовательностей.) Следует отметить, что формат GIF относится к категории методов сжатия с потерями, поскольку цвета в палитре могут не вполне точно соответствовать цветам в исходном изображении.

В формате GIF можно получить дополнительное сжатие за счет расширения его простой словарной системы до адаптивной словарной системы, используемой в методах LZW. В частности, поскольку образцы пикселей анализируются непосредственно в процессе кодирования, они могут быть добавлены в словарь, и в дальнейшем сходные пиксели будут кодироваться более эффективно. Таким образом, результирующий словарь будет состоять из оригинальной палитры и набора образцов пикселей.

Обычно один из цветов палитры в формате GIF воспринимается как обозначение "прозрачности". Это означает, что в закрашенных этим "цветом" участках изображения должен отображаться цвет того фона, на котором оно находится. Благодаря этому, а также относительной простоте использования изображений, формат GIF получил широкое распространение в тех компьютерных играх, в которых множество различных картинок перемещается по экрану. В то же время ограничение палитры на уровне 256 цветов делает нецелесообразным использование этого формата для приложений, в которых требуется высокая точность цветопередачи, например в цифровой фотографии.

Другим примером системы сжатия изображений является формат **JPEG**. Это стандарт, разработанный ассоциацией **Joint Photographic Experts Group** (отсюда и название этого стандарта) в рамках организации ISO. Формат JPEG показал себя как эффективный метод представления цветных фотографий и сейчас широко используется в этой индустрии, о чем свидетельствует тот факт, что в большинстве цифровых фотокамер от любых изготовителей именно формат JPEG используется как метод сжатия изображений, выбираемый по умолчанию.

В действительности стандарт JPEG включает несколько способов представления изображения, каждый из которых имеет собственное назначение. Например, когда требуется максимальная точность представления изображения, формат JPEG предлагает режим "без потерь", название которого прямо указывает, что процедура кодирования изображения будет выполнена без каких-либо потерь информации. Однако режим "без потерь" формата JPEG не обеспечивает высокого уровня сжатия в сравнении с другими режимами этого формата. Более того, другие режимы формата JPEG показали себя как очень удачные, вследствие чего режим JPEG "без потерь" на практике используется довольно редко. Вместо него большинство существующих приложений используют другой стандартный режим формата JPEG — режим "базовых строк".

Сжатие изображений в режиме "базовых строк" формата JPEG осуществляется в несколько этапов, первый из которых основан на использовании особенностей восприятия человеческим глазом уровня яркости. В частности, глаз человека более чувствителен к изменениям яркости, чем цвета, поэтому исходное изображение, закодированное в терминах яркости и цветности каждого пикселя, подвергается обработке с целью усреднения показателей цветности для квадратных блоков, состоящих из четырех пикселей. Это позволяет сократить объем информации о цветности в четыре раза, сохранив при этом исходную информацию о яркости пикселей. В результате мы получаем существенное повышение уровня сжатия изображения при отсутствии заметных потерь в его качестве.

На следующем этапе изображение разделяется на квадратные блоки со стороной восемь пикселей и дальнейшее сжатие информации осуществляется уже для каждого из них в отдельности. Сжатие каждого блока выполняется с использованием математического метода, называемого дискретным косинусным преобразованием, подробное рассмотрение которого выходит за рамки данного обсуждения. Важным для нас моментом является лишь то, что это преобразование конвертирует исходный блок размером восемь на восемь пикселей в другой блок, элементы которого отражают, насколько пиксели исходного блока *отличаются* один от другого, вместо того чтобы представлять *реальные характеристики* этих пикселей. В пределах этого нового блока значения, не превышающие определенного порога, заменяются нулями, отражая тот факт, что изменения, представленные этими значениями, слишком малы, чтобы их мог различить человеческий глаз. Например, если исходный блок содержит узор, подобный шахматной доске, новый блок может указывать на равномерную окраску его пикселей некоторым усредненным цветом. (Как правило, квадратный блок со стороной восемь пикселей представляет очень маленький фрагмент изображения, поэтому глаз человека в любом случае не сможет выявить в нем шахматный узор.)

Далее, для получения дополнительного сжатия к данным могут применяться обычные методы кодирования длины серий, относительного кодирования и кодирования переменной длины. С учетом всего сказанного выше режим "базовых строк" формата JPEG обычно позволяет достичь сжатия цветных изображения с фактором не менее 10, а часто и до 30, без возникновения заметных потерь качества изображений.

Совершенно иной метод сжатия применяется к изображениям в формате, получившем название **TIFF** (Tagged Image File Format). Однако наибольшую популярность формат TIFF приобрел не как средство сжатия данных, а как стандартный формат сохранения фотографий вместе с относящейся к ним дополнительной информацией, такой как дата и время съемки, использованная при съемке фотокамера и параметры ее настройки. В этом контексте изображение само по себе обычно сохраняется с указанием значений красной, зеленой и синей компонент пикселей без сжатия.

В совокупность стандартов формата TIFF входят и методы сжатия изображений, большинство из которых разработаны для сжатия изображений страниц текстовых документов, создаваемых факсимильными приложениями. В них используются те или иные варианты метода кодирования длины серий, выбранного исходя из того факта, что в текстовых документах всегда присутствуют длинные последовательности белых пикселей. Варианты методов сжатия цветных изображений, включенные в стандарты формата TIFF, построены на методах, схожих с теми, которые используются в формате GIF, и по этой причине не получили широкого распространения в сообществе фотографов.

Сжатие аудио и видео

Наиболее распространенные стандарты для кодирования и сжатия аудио- и видеоданных были разработаны экспертной группой **MPEG** (Motion Picture Experts Group) под эгидой ISO. Именно по этой причине эти стандарты и получили общее название "MPEG".

Разработки MPEG включают целую группу стандартов различного назначения. Например, требования к телевидению высокой четкости (HDTV) сильно отличаются от тех, которые выдвигает технология проведения видеоконференций, в соответствии с которой предполагается, что рассылаемые данные должны доставляться адресатам по самым разным каналам связи, в том числе имеющим ограниченные возможности. Эти варианты применения видео существенно расходятся в своих требованиях в отношении задержки поступления видеоданных и возможности повторения или пропуска их сегментов.

Обсуждение технологий, используемых в стандартах MPEG, выходит далеко за рамки этого курса, но в целом методы сжатия видеоданных построены на

том, что они рассматриваются как последовательность изображений — примерно так, как последовательность отснятых кадров образует киноленту. Для сжатия таких последовательностей лишь некоторые из общего числа изображений, называемые *опорными кадрами*, кодируются во всей их полноте. Все остальные изображения между двумя опорными кадрами кодируются с использованием методов относительного кодирования. Иначе говоря, кодируются не все эти изображения целиком, а только различия между этой картинкой и предыдущей. Кодирование опорных кадров обычно осуществляется методами, подобными используемым в формате JPEG.

Наиболее известным методом сжатия аудиоданных является формат **MP3**, который входит в группу стандартов разработки ассоциации MPEG. В действительности обозначение "MP3" является сокращением от *MPEG layer 3* (MPEG уровня 3). Помимо прочих методов сжатия, дополнительные преимущества в стандарте MP3 обеспечиваются с учетом всех особенностей восприятия человеческого уха — с целью удаления из аудиоданных всех тех деталей, которые человек просто не способен воспринять. Одно из таких качеств, называемое **временной маскировкой**, заключается в том, что после громкого звука человеческое ухо на короткий период оказывается не может различать более тихие звуки, которые легко различает в иных случаях. Другая особенность, называемая **частотной маскировкой**, состоит в том, что звук определенной частоты способен маскировать более тихие звуки соседних частот. Извлекая преимущества из использования подобных особенностей человеческого восприятия, формат MP3 позволяет достичь весьма значительного уровня сжатия аудиоданных, сохраняя при этом их качество на уровне, свойственном записям на компакт-дисках.

Используя методы сжатия форматов MPEG и MP3, современные видеокамеры способны записывать до одного часа видео хорошего качества в памяти объемом 128 Мбайт, а портативные музыкальные проигрыватели могут хранить до 400 популярных песен в хранилище размером 1 Гбайт. Однако в противоположность целям, преследуемым при сжатии прочих данных, цель сжатия аудио- и видеоданных необязательно может заключаться лишь в уменьшении размера внешней памяти, требуемой для их хранения. Не менее важной целью является получение таких методов кодирования, которые позволят передавать информацию по современным каналам связи со скоростью, достаточной для проведения аудио- или видеопередач в реальном времени. Если каждый кадр видео будет иметь размер до 1 Мбайт и эти кадры потребуется передавать по каналу, пропускная способность которого измеряется в килобайтах, об успешном проведении видеконференций не может быть и речи. В результате, помимо

приемлемого качества воспроизведения, системы сжатия аудио и видео часто оценивают по требуемой в этом случае скорости передачи данных, обеспечивающей их воспроизведение в реальном времени. Скорость передачи данных по каналам обычно измеряют в битах за секунду. Чаще всего используются производные единицы, такие как **Кбит/с**, **Мбит/с** и **Гбит/с**. При использовании технологий MPEG видеопрезентации можно успешно проводить при наличии каналов с пропускной способностью от 40 Мбит/с. Трансляция в реальном времени аудиозаписей в формате MP3 в общем случае потребует каналов связи с пропускной способностью не ниже 64 Кбит/с.

1.9. Вопросы и упражнения

1. Перечислите четыре общих метода сжатия.
2. Как будет выглядеть закодированная версия сообщения

   `xyx yxxxy xyx yxxxy yxxxy`

   при использовании сжатия по методу LZW со словарем, исходно содержащим элементы *x*, *y* и пробел (как описывалось в примере, приведенном выше)?
3. Выше утверждалось, что формат GIF позволяет лучше представлять цветные мультипликационные изображения, чем формат JPEG. Объясните, почему это действительно так.
4. Предположим, что вы — участник команды, занятой разработкой космического летательного аппарата, предназначенного для полетов на другие планеты с целью их фотографирования и отправки полученных фотоматериалов на Землю. Будет ли хорошей идеей использовать форматы GIF или JPEG в режиме "базовых строк" для сокращения объема ресурсов, требуемых для хранения и передачи изображений?
5. Какие особенности человеческого глаза используются в режиме "базовых строк" формата JPEG?
6. Какие особенности слуха человека используются в формате MP3?
7. Укажите вызывающую затруднения особенность, общую для использования числовой информации, изображений и аудио, закодированных в виде последовательности битов.

1.10. Ошибки при передаче информации

Когда информация постоянно передается между различными частями компьютера, пересылается от Земли к Луне и обратно либо просто сохраняется в устройстве памяти, существует вероятность, что полученная в конечном счете битовая комбинация будет отличаться от исходной. Частички грязи или жира на магнитной записывающей поверхности, случайная ошибка в работе электронной схемы — все это может вызвать ошибки при записи или чтении данных. Статические заряды на пути передачи данных могут привести к их искажению. Более того, при использовании некоторых технологий хранения данных фоновое радиационное излучение может изменять битовые комбинации, записанные в основной памяти машины.

Для решения этих проблем было разработано множество технологий кодирования данных, позволяющих обнаруживать и даже исправлять подобные ошибки. В настоящее время эти технологии широко используются при создании внутренних компонентов компьютеров, поэтому они остаются незаметными для пользователей, работающих с машиной. Однако они чрезвычайно важны, и это лишний раз подчеркивает значение результатов выполненных научных исследований. В сущности, бо́льшую часть работы по созданию подобных технологий выполнили математики-теоретики. Ниже мы рассмотрим некоторые из тех методов, которые обеспечивают надежность функционирования современных вычислительных машин.

Биты четности

Существует достаточно простой способ определения ошибок, построенный на том принципе, что если каждая обрабатываемая битовая комбинация будет состоять из нечетного количества единиц, то обнаружение комбинации с четным количеством единиц будет свидетельствовать о возникновении ошибки. Чтобы использовать этот принцип, необходимо создать систему кодирования, в которой любая битовая комбинация будет содержать нечетное количество единиц. Проще всего это можно достичь путем добавления к уже существующему коду дополнительного бита (который называется **битом четности** или **контрольным битом**), чаще всего помещаемого в старший конец комбинации. В любом случае этому биту присваивается значение `1` или `0` таким образом, чтобы общее количество единиц в битовой комбинации всегда было нечетным. Если система кодирования будет модифицирована подобным образом, то при обнаружении значения данных с четным числом единиц будет очевидно, что здесь имеет место ошибка и данное значение следует обрабатывать как некорректное.

На рис. 1.26 показано, как бит четности может быть добавлен в коды ASCII, представляющие символы A и F. Обратите внимание, что код символа A теперь будет представлен как `101000001` (бит четности равен `1`), тогда как код символа F в этом случае будет иметь вид `001000110` (бит четности равен `0`). Хотя исходная восьмиразрядная комбинация, представляющая букву A, содержит четное количество единиц, а аналогичная комбинация, представляющая букву F, — нечетное количество единиц, оба девятиразрядных кода имеют нечетное количество единиц. Если этот метод будет применен ко всем восьмиразрядным кодам символов в кодировке ASCII, то мы получим девятиразрядную кодировку, в которой появление битовой комбинации с четным количеством единиц будет свидетельствовать об ошибке.

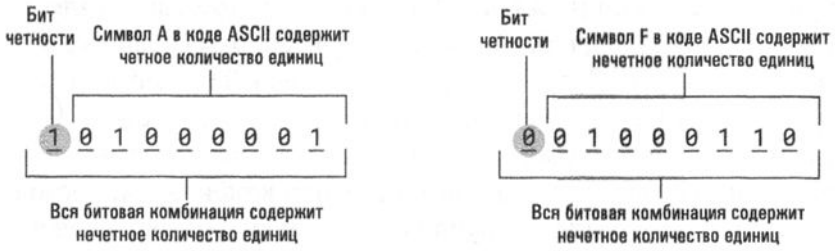


Рис. 1.26. Представление символов A и F в коде ASCII с использованием контрольного бита

Описанная выше система контроля четности называется **проверкой на нечетность**, поскольку все обрабатываемые битовые комбинации должны содержать нечетное количество единиц. Также существует и иной способ, именуемый **проверкой на четность**. В этом случае все обрабатываемые комбинации должны содержать четное количество единиц, а показателем ошибки является нечетное количество единиц в битовой комбинации.

В наше время использование битов четности является типовым решением для основной памяти машины. Хотя создается впечатление, что компьютеры используют восьмиразрядные ячейки памяти, в действительности они являются девятиразрядными, причем девятый бит используется как контрольный. Каждый раз, когда в память записывается некоторая восьмибитовая комбинация, схема управления памятью автоматически добавляет к ней требуемый контрольный бит для получения девятиразрядной комбинации. При считывании информации схема управления памятью подсчитывает количество единиц в полученной комбинации. Если ошибка не обнаруживается, контрольный бит удаляется и образуется исходная восьмиразрядная битовая комбинация. В противном случае схема управления памятью возвращает считанное восьмиразрядное значение с указанием, что оно искажено и может отличаться от того исходного, которое записывалось в память.

Прямое использование контрольных битов является очень простым методом, но он имеет свои ограничения. Если в сочетании битов, которое исходно содержало нечетное количество единиц, произойдут *две* ошибки, общее количество единиц останется нечетным, а значит, система контроля четности этой ошибки не обнаружит. В действительности механизм с использованием бита контроля четности не позволит обнаружить любое четное количество ошибок, возникшее при считывании кодовой комбинации.

Один из методов, минимизирующих вероятность возникновения подобных проблем, иногда применяют к длинным битовым последовательностям, таким как строка битов, записываемая в сектор магнитного диска. В этом случае длинные битовые комбинации часто дополняются группой контрольных битов, образующих **контрольный байт**. Каждый бит в этом байте является контрольным и относится к определенной группе битов, разбросанных по основной битовой комбинации. Например, один контрольный бит может относиться к каждому восьмому биту, начиная с первого, тогда как другой — к каждому восьмому биту, начиная со второго, и т.д. В данном случае легче выявить ошибки, сконцентрированные в одной области исходной комбинации, поскольку их наличие будет контролироваться группой контрольных битов. Различные варианты данного подхода к созданию схем контроля называются методом контрольных сумм и методом **использования кода циклического контроля избыточности** (CRC).

Коды с исправлением ошибок

Несмотря на то что бит четности является эффективным методом выявления ошибок, он не дает информации, необходимой для исправления возникшей ошибки. Многих удивляет сам факт, что можно разработать **код с исправлением ошибок**, позволяющий не только выявлять ошибки, но и исправлять их. В конце концов, интуиция подсказывает, что мы не в состоянии исправить ошибку в полученном сообщении, если заранее не знаем, о чем там идет речь. Тем не менее существует довольно простой код, позволяющий исправлять возникающие ошибки (рис. 1.27).

Для того чтобы понять принцип действия этого кода, сначала необходимо определить понятие **дистанции Хэмминга** между двумя кодовыми комбинациями, которая будет равна количеству битов, различающихся в этих комбинациях. (Понятие дистанция Хэмминга получило свое название в честь Р.В. Хэмминга (R.W. Hamming), который провел первые исследования в области разработки кодов с исправлением ошибок. Он обратился к этой проблеме в 1940-х годах по причине крайней ненадежности существовавших в то время

релейных вычислительных машин.) Например, дистанция Хэмминга между кодами букв A и B (см. рис. 1.27) равна четырем, а дистанция Хэмминга между кодами букв B и C равна трем. Важной особенностью этого кода является то, что дистанция Хэмминга между любыми двумя комбинациями будет не меньше трех. Если в результате сбоя в каком-либо отдельном бите появится ошибочное значение, то ошибка будет легко установлена, так как получившаяся комбинация не является допустимым кодовым значением. В любой комбинации потребуется изменить не меньше трех битов, прежде чем она вновь станет допустимой.

Символ	Код
A	000000
B	001111
C	010011
D	011100
E	100110
F	101001
G	110101
H	111010

Рис. 1.27. Код с исправлением ошибок

Если в любой комбинации, показанной на рис. 1.27, возникла одиночная ошибка, то легко можно вычислить ее исходное значение. Дело в том, что дистанция Хэмминга для измененной комбинации по отношению к исходной форме будет равна единице, тогда как по отношению к другим разрешенным комбинациям она будет равна не менее чем двум.

Таким образом, для декодирования некоторого сообщения, которое исходно было закодировано в соответствии с таблицей, представленной на рис. 1.27, достаточно просто сравнивать каждую полученную недопустимую битовую комбинацию с допустимыми комбинациями кода, пока не будет найдена комбинация, находящаяся на дистанции, равной единице, от полученной комбинации. Найденная допустимая кодовая комбинация принимается за правильный символ, полученный в результате декодирования. Предположим, что получена битовая комбинация 010100. Если сравнить ее с допустимыми битовыми комбинациями кода, то будет получена таблица дистанций, представленная на рис. 1.28. По содержанию этой таблицы можно сделать заключение, что поступивший символ — это буква D, так как ее битовая комбинация наиболее точно соответствует полученной.

Символ	Код	Полученная комбинация	Дистанция между полученной комбинацией и кодом символа
A	000000	010100	2
B	001111	010100	4
C	010011	010100	3
D	011100	010100	1 — Наименьшая дистанция
E	100110	010100	3
F	101001	010100	5
G	110101	010100	2
H	111010	010100	4

Рис. 1.28. Декодирование битовой комбинации 010100 с помощью кода, представленного на рис. 1.27

Как видите, при использовании кода, представленного на рис. 1.27, действительно можно обнаружить до двух ошибок в одной комбинации и исправить одну ошибку. Если использовать код, в котором дистанция Хэмминга между комбинациями равняется как минимум пяти, то можно было бы обнаруживать до четырех ошибок в одной комбинации и исправлять до двух ошибок. Естественно, разработка эффективных кодов с достаточно длинными дистанциями Хэмминга является непростой задачей. Для этого требуется знание такой области математики, как алгебраическая теория кодирования, которая является частью линейной алгебры и теории матриц.

Методы коррекции ошибок широко используются в целях повышения надежности вычислительной техники. Например, они используются в драйверах магнитных дисков большой емкости, чтобы снизить вероятность искажения хранимой информации в результате дефектов поверхности диска. Более того, главное отличие между оригинальным цифровым форматом, используемым в звуковых компакт-дисках, и форматом CD-ROM, предназначенным для записи компьютерных данных, заключается именно в использовании кодов с исправлением ошибок. Функция исправления ошибок в формате аудиоCD позволяет устранять только одну ошибку на два компакт-диска, и этого вполне достаточно для аудиозаписей. Однако для компаний, поставляющих программное обеспечение, наличие ошибок в 50% поставляемых ими компакт-дисков является совершенно недопустимым. Поэтому в формат CD-ROM включены дополнительные средства, позволяющие снизить вероятность возникновения ошибки до одной на 20 тысяч компакт-дисков.

1.10. Вопросы и упражнения

1. Приведенные ниже битовые комбинации закодированы с использованием контроля по нечетности. Укажите комбинации с ошибками.

   а. `100101101` **б.** `100000001` **в.** `000000000`

   г. `111000000` **д.** `011111111`

2. Могли бы вы не заметить ошибки в байтах, представленных в п.1? Поясните свой ответ.

3. Какими бы были ваши ответы на вопросы 1 и 2, если бы в приведенных байтах использовался контроль на четность?

4. Закодируйте эти предложения в коде ASCII с использованием контроля по нечетности и помещением контрольного бита в старший конец кода каждого символа.

   а. "Stop!" Cheryl shouted.

   б. Does 2 + 3 = 5?

5. Используйте представленный на рис. 1.28 код с исправлением ошибок для декодирования следующих сообщений.

   а. `001111 100100 001100` **б.** `010001 000000 001011`

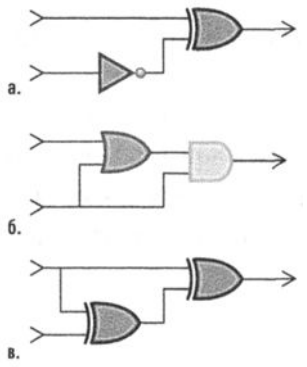
   в. `011010 110110 100000 011100`

6. Используя пятиразрядные битовые комбинации, разработайте для символов A, B, C, D такой код, чтобы дистанция Хэмминга между любыми двумя комбинациями составляла не меньше трех.

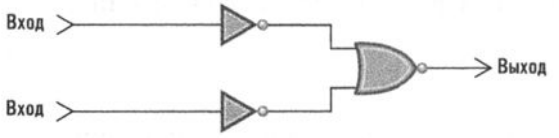
ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

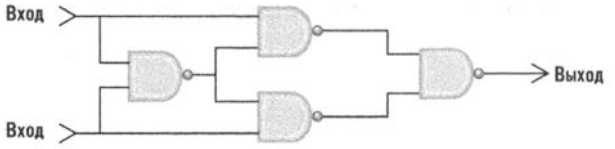
1. Какое выходное значение будет иметь каждая из приведенных ниже схем, если предположить, что на их верхний вход подано значение `1`, а на нижний — значение `0`?



2. **а.** Какая булева операция реализуется приведенной ниже схемой?

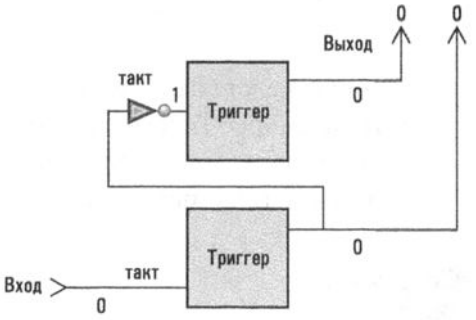


б. Какая булева операция реализуется приведенной ниже схемой?

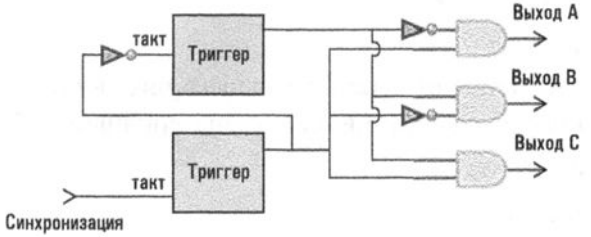


***3.** **а.** Если вы когда-либо приобретали микросхемы триггеров в магазине электронных компонентов, то, вероятно, обратили внимание, что эти микросхемы имеют дополнительные входы с названием "Такт". Когда уровень сигнала на этих входах меняется от `0` до `1`, уровень сигнала на выходе триггера меняется на обратный (если это был `0`, то появится `1`, и наоборот). Однако, когда уровень сигнала на тактовом входе меняется от `1` до `0`, ничего не происходит. Даже не зная всех деталей построения электронной схемы этой микросхемы, обеспечивающих такое ее поведение, мы все же можем использовать ее как абстрактный инструмент в других электронных схемах. Рассмотрим работу схемы, в которой используются два подобных триггера — ее исходное состояние представ-

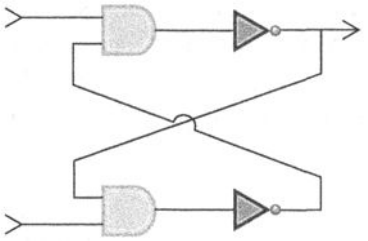
лено на рисунке ниже. Если подать импульс на вход нижнего триггера, он изменит свое состояние на 1, тогда как второй триггер при этом останется в прежнем состоянии, поскольку сигнал на его вход (подаваемый через вентиль НЕ, — *инвертор*) изменится с `1` на `0`. В результате на выходе всей схемы появятся сигналы `0` и `1`. Второй импульс, поданный на вход этой схемы, вызовет изменение состояния у обоих триггеров и на выходе появятся сигналы `1` и `0`. Что будет на выходе схемы после поступления на ее вход третьего импульса? А после поступления четвертого?



б. Очень часто возникает необходимость координировать действия различных электронных компонентов компьютера. Реализуется это требование посредством подачи периодического импульсного сигнала (называемого *синхронизирующим сигналом*) на электронную схему, подобную рассматривавшейся в предыдущем пункте этого задания. Дополнительные вентили (как показано на схеме ниже) предназначены для скоординированной передачи сигналов на другие подключенные к ним схемы. Внимательно проанализировав работу этой схемы, вы сможете подтвердить, что поступление 1-, 5-, 9-го... импульсов синхронизации вызывает появление `1` на выходе A. Какие импульсы синхронизации вызовут появление `1` на выходе B? А по каким импульсам синхронизации `1` будет появляться на выходе C? На каком выходе появится `1` после поступления 4-го импульса синхронизации?



4. Предположим, что на оба входа приведенной ниже схемы поданы значения `1`. Опишите, что произойдет, если на верхний вход временно подать значение `0`. А что произойдет, если временно подать значение `0` на нижний вход?



5. В приведенной ниже таблице представлены адреса и содержимое (в шестнадцатеричной нотации) некоторых ячеек основной памяти машины. Начиная с этого состояния, выполните приведенные инструкции и запишите содержимое этих ячеек памяти после выполнения всех указанных действий.

Адрес	Содержимое
`0x00`	`0xAB`
`0x01`	`0x53`
`0x02`	`0xD6`
`0x03`	`0x02`

Этап 1. Переместите содержимое ячейки с адресом `0x03` в ячейку с адресом `0x00`.

Этап 2. Поместите число `0x01` в ячейку с адресом `0x02`.

Этап 3. Переместите значение, сохраняемое по адресу `0x01`, в ячейку с адресом `0x03`.

6. Сколько ячеек может содержаться в основной памяти компьютера, если адрес каждой ячейки может быть представлен тремя шестнадцатеричными цифрами? А если четырьмя шестнадцатеричными цифрами?

7. Какие комбинации битов представлены следующими шестнадцатеричными обозначениями?

а. `0xCD` **б.** `0x67` **в.** `0x9A`

г. `0xFF` **д.** `0x10`

8. Определите значение старшего значащего бита в битовых комбинациях, представленных следующими шестнадцатеричными обозначениями.

а. `0x8F` **б.** `0xFF`

в. `0x6F` **г.** `0x1F`

9. Представьте следующие битовые комбинации в шестнадцатеричной системе счисления.
   а. `101000001010`
   б. `110001111011`
   в. `000010111110`
10. Предположим, что объем памяти цифровой фотокамеры составляет 256 Мбайт. Сколько фотографий может быть сохранено в памяти камеры, если каждая из них состоит из 1024 пикселей в строке и 1024 пикселей в столбце и каждый пиксель требует 3 байт для описания его состояния?
11. Предположим, что выводимое на экран монитора изображение представляет собой прямоугольник, состоящий из 1024 столбцов и 768 строк пикселей. Если для кодирования цвета и яркости каждого пикселя используется по 8 бит, то сколько однобайтовых ячеек памяти потребуется для сохранения всего выводимого изображения?
12. **а.** Назовите два преимущества, которые имеет основная память машины по отношению к магнитными дисковым устройствам хранения информации.
    б. Назовите два преимущества, которые имеют магнитные дисковые устройства хранения информации по отношению к основной памяти машины.
13. Предположим, что на жестком диске вашего компьютера, емкость которого составляет 120 Гбайт, осталось только 50 Гбайт свободного дискового пространства. Имеет ли смысл использовать компакт-диски для сохранения всего материала, хранящегося на этом жестком диске, как резервной копии? А что можно сказать в отношении DVD-дисков? Поясните свой ответ.
14. Если каждый сектор диска содержит 1024 байта, то сколько секторов потребуется для записи одной печатной страницы текста (примерно 50 строк по 100 символов) на иностранном языке, если для представления одного символа этого текста в кодировке Unicode необходимо 2 байта?
15. Сколько байтов памяти потребуется для сохранения романа объемом 400 страниц, если каждая страница содержит 3500 символов в кодировке ASCII? А сколько потребуется байтов в случае использования кодировки Unicode с двухбайтовым представлением символов?
16. Какой будет задержка вращения для типичного устройства внешней памяти с магнитным жестким диском, вращающимся со скоростью 3600 оборотов в минуту?
17. Каким будет среднее время доступа жесткого диска, вращающегося со скоростью 360 оборотов в секунду, если его время поиска равно 10 мс?

18. Предположим, что машинистка может непрерывно день за днем печатать текст со скоростью 60 слов в минуту. Сколько времени потребуется ей, чтобы заполнить компакт-диск емкостью 640 Мбайт? Будем считать, что каждое слово состоит из пяти букв, а для сохранения каждой буквы требуется один байт.

19. Ниже приведен текст сообщения, представленного в кодах ASCII. О чем говорится в этом сообщении?

```
01010111 00100000 01110011 00100000
00111111 01101000 01100100 00100000
01110011 01100001 01101111 01101001
01100001 01110100 01100101 01110100
01111001
```

20. Ниже приведен текст сообщения, представленного в кодах ASCII. В этой кодировке на один символ приходится один байт, который в этом упражнении представлен в шестнадцатеричной системе счисления. Декодируйте этот текст.

```
0x686578206E6F746174696F6E
```

21. Закодируйте приведенные ниже выражения в кодах ASCII, используя один байт на один символ.

а. Does 100 / 5 = 20?

б. The total cost is $7,25.

22. Представьте ответ из упражнения 21 в шестнадцатеричной системе счисления.

23. Перечислите двоичные представления для целых чисел от 8 до 18.

24. **а.** Запишите число 23, представив цифры 2 и 3 в коде ASCII.

б. Запишите число 23 в двоичной системе счисления.

25. В двоичном представлении каких чисел содержится только один единичный бит? Приведите двоичные представления для шести наименьших чисел такого вида.

***26.** Преобразуйте приведенные ниже двоичные числа в эквивалентные десятичные значения.

а. `1111` **б.** `0001` **в.** `10101` **г.** `1000`

д. `10011` **е.** `000000` **ж.** `1001` **з.** `10001`

и. `100001` **к.** `11001` **л.** `11010` **м.** `11011`

***27.** Представьте следующие десятичные числа в двоичном представлении.

а. 7 **б.** 11 **в.** 16

г. 17 **д.** 31

***28.** Преобразуйте приведенные ниже числа, представленные в двоичной нотации с избытком шестнадцать, в десятичную форму.

а. 10001 **б.** 10101 **в.** 01101
г. 01111 **д.** 11111

***29.** Представьте приведенные ниже десятичные числа в двоичной нотации с избытком четыре.

а. 0 **б.** 3 **в.** –2 **г.** –1 **д.** 2

***30.** Представьте в десятичной системе счисления приведенные ниже числа, записанные в двоичном дополнительном коде.

а. 01111 **б.** 10100 **в.** 01100
г. 10000 **д.** 10110

***31.** Представьте приведенные ниже десятичные числа в двоичном дополнительном коде разрядностью 7 битов.

а. 13 **б.** –13 **в.** –1 **г.** 0 **д.** 16

***32.** Выполните приведенные ниже операции сложения, полагая, что строки битов представляют числа в двоичном дополнительном коде. Укажите, когда полученный результат будет ошибочным из-за переполнения.

а. 00101 + 01000 **б.** 11111 + 00001
в. 01111 + 00001 **г.** 10111 + 11010
д. 11111 + 11111 **е.** 00111 + 01100

***33.** Решите приведенные ниже задачи посредством перевода десятичных чисел в двоичный дополнительный код (длиной 5 бит). Преобразуйте операции вычитания в эквивалентные операции сложения, а затем выполните суммирование. Проверьте полученные ответы, преобразовав их в десятичную систему счисления (не забывайте о возможности появления ошибок переполнения).

а. 5 + 1 **б.** 5 – 1 **в.** 12 – 5
г. 8 – 7 **д.** 12 + 5 **е.** 5 – 11

***34.** Запишите в десятичной системе счисления приведенные ниже числа в двоичном представлении.

а. 11.11 **б.** 100.0101 **в.** 0.1101
г. 1.0 **д.** 10.01

***35.** Запишите приведенные ниже числа в двоичной системе счисления.

а. $5^{3}/_{4}$ **б.** $15^{15}/_{16}$ **в.** $5^{3}/_{8}$
г. $1^{1}/_{4}$ **д.** $6^{5}/_{8}$

***36.** Декодируйте следующие битовые комбинации в формате с плавающей точкой, представленном на рис. 1.24.

а. 01011001 **б.** 11001000
в. 10101100 **г.** 00111001

***37.** Закодируйте приведенные ниже числа, используя восьмиразрядный формат с плавающей точкой, представленный на рис. 1.24. Укажите на ошибки усечения.

а. $-7\frac{1}{2}$ **б.** $\frac{1}{2}$ **в.** $-3\frac{3}{4}$ **г.** $\frac{7}{32}$ **д.** $\frac{31}{32}$

***38.** Предположим, что ограничение по использованию только нормализованных форм для чисел в формате с плавающей точкой было отменено. Приведите все возможные комбинации битов, которые могут быть в этом случае использованы для представления числа 3/8 в двоичном формате с плавающей точкой, представленном на рис. 1.24.

***39.** Какое наилучшее приближение для значения квадратного корня из числа 2 может быть представлено в восьмиразрядном формате с плавающей точкой, представленном на рис. 1.24? Какое число мы получим, если это приближенное представление корня будет возведено в квадрат в машине, использующей указанный формат с плавающей точкой?

***40.** Какое наилучшее приближение для значения числа 1/10 может быть достигнуто в восьмиразрядном формате с плавающей точкой, представленном на рис. 1.24?

***41.** Объясните, как могут возникнуть ошибки, если измерения, выполненные с использованием метрической системы, сохраняются в формате с плавающей точкой. Например, что произойдет, если значение 110 см будет записано в метрах?

***42.** Одна из двух битовых комбинаций, `01011` и `11011`, представляет некоторое число в двоичной нотации с избытком шестнадцать, а другая — это же число в двоичном дополнительном коде.

а. Что можно сказать относительно этого закодированного значения?

б. Какая взаимосвязь существует между комбинацией битов, представляющей число в двоичном дополнительном коде, и комбинацией битов, представляющей это же число в двоичной нотации с избытком, если в обоих случаях используется один и тот же размер комбинации?

***43.** Три битовые комбинации, `01101000`, `10000010` и `00000010`, представляют одно и то же число, записанное в разных форматах: двоичном дополнительном коде, двоичной нотации с избытком и восьмиразрядном формате с плавающей точкой, представленном на рис. 1.24, однако необязательно в указанном порядке. Определите это число и укажите, какая из битовых комбинаций принадлежит к каждому из перечисленных форматов.

***44.** Какие из приведенных ниже чисел невозможно точно представить в формате с плавающей точкой, представленном на рис. 1.24?

а. $6\frac{1}{2}$ **б.** $\frac{13}{16}$ **в.** 9 **г.** $\frac{17}{32}$ **д.** $\frac{15}{16}$

***45.** Если увеличить длину битовой строки, используемой для представления целых чисел в двоичной системе, от четырех до шести битов, то какие изменения вызовет это действие в значении наибольшего целого числа, которое может быть представлено этой строкой? Каков будет ответ при использовании двоичного дополнительного кода?

***46.** Каким будет шестнадцатеричное представление наибольшего адреса памяти, если размер этой памяти составляет 4 Мбайт и каждая ячейка имеет длину 1 байт.

***47.** Как будет выглядеть закодированная версия сообщения

```
xxy yyx xxy xxy yyx
```

при использовании сжатия по методу LZW со словарем, исходно содержащим элементы *x*, *y* и пробел (как описывалось в разделе 1.8)?

***48.** Следующее сообщение было сжато с использованием сжатия по методу LZW со словарем, исходно содержащим *x*, *y* и пробел как первый, второй и третий элементы соответственно. Каким было исходное сообщение?

```
22123113431213536
```

***49.** Пусть сообщение

```
xxy yyx xxy xxyy
```

было сжато с использованием сжатия по методу LZW со словарем, исходно содержащим *x*, *y* и пробел как первый, второй и третий элементы соответственно. Как будет выглядеть словарь после завершения сжатия этого сообщения?

***50.** Как вы узнаете из следующей главы, одним из способов передачи битов по обычным телефонным линиям связи предусматривается преобразование последовательностей битов в звуковые сигналы с последующей передачей их по телефонным линиям и обратным преобразованием этих звуковых сигналов в последовательность битов на приемной стороне. Этот метод обеспечивает предельную скорость передачи данных на уровне 57,6 Кбит/с. Достаточно ли этой скорости передачи данных для проведения видеоконференций, если видеоданные будут сжиматься с использованием стандарта MPEG?

***51.** Закодируйте следующие выражения в кодах ASCII, используя один байт на символ. Используйте старший бит каждого байта в качестве контрольного бита (по нечету).

а. Does 100/5 = 20?

б. The total cost is $7,25.

***52.** Следующее сообщение было передано с использованием контрольного бита (по нечету) в каждой короткой строке битов. В каких строках имеются ошибки?

```
11001   11011   10110   00000   11111   10001
10101   00100   01110
```

***53.** Предположим, что 24-разрядный код создан посредством представления каждого символа с помощью трех последовательных копий его ASCII-кода (например, символ A будет представлен строкой битов `010000010100000101000001`). Какие возможности исправления ошибок допускает этот код?

***54.** Для декодирования приведенных ниже слов используйте код с исправлением ошибок, представленный на рис. 1.28.

а. `111010 110110`

б. `101000 100110 001100`

в. `011101 000110 000000 010100`

г. `010010 001000 001110 101111 000000 110111 100110`

д. `010011 000000 101001 100110`

***55.** Курсы обмена различных существующих в мире валют часто меняются. Выясните текущие значения соответствующих курсов обмена и соответствующим образом обновите сценарий обмена валют, приведенный в разделе 1.8.

***56.** Выберите некоторую валюту, расчеты по которой не проводятся в конверторе валют из раздела 1.8, выясните ее курс обмена и найдите символ Unicode, которым эта валюта обычно представляется в Интернете, а затем дополните сценарий так, чтобы он выполнял конвертирование и этой валюты.

***57.** Если браузер и текстовый редактор, которыми вы обычно пользуетесь, корректно поддерживают кодировки Unicode и UTF-8, скопируйте в буфер обмена, а затем вставьте в сценарий конвертера валют из раздела 1.8 символ соответствующей международной валюты, заменив им соответствующее ему громоздкое представление, например `'\u00A3'`. (Если используемое вами программное обеспечение не поддерживает обработку символов Unicode, при попытке выполнить это задание в окне текстового редактора у вас появятся странные символы.)

***58.** В сценарии конвертера валют из раздела 1.8 переменная `dollars` используется для хранения конвертируемой суммы денег до момента выполнения каждой из операций умножения. Это делает сценарий на одну строку длиннее, чем в случае простого указания числа 1000 непосредственно в каждом операторе умножения. Чем может быть выгодно заранее создавать эту дополнительную переменную?

*59. Напишите и протестируйте работу сценария на языке Python, который заданное количество байтов будет выводить в виде эквивалентного количества килобайтов, мегабайтов, гигабайтов и терабайтов. Напишите и протестируйте дополнительный сценарий, который будет выводить заданное значение терабайтов в виде эквивалентного количества гигабайтов, мегабайтов, килобайтов и байтов.

*60. Напишите и протестируйте работу сценария на языке Python, который по заданному количеству минут и секунд продолжительности аудиозаписи будет рассчитывать количество битов, необходимых для кодирования несжатых аудиоданных указанной продолжительности, записанных с качеством, соответствующим стереозаписям на компакт-дисках. (Необходимые для расчетов параметры и уравнения вы найдете в разделе 1.4.)

*61. Найдите ошибку (ошибки) в следующем сценарии на языке Python.

```
days_per_week = 7
weeks_per_year = 52
days_per_year = days_per_week ** weeks_per_year
PRINT(days_per_year)
```

СОЦИАЛЬНЫЕ И ОБЩЕСТВЕННЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также понять, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. Предположим, что ошибка усечения возникла в критической ситуации и послужила причиной обширных разрушений и гибели людей. Должен ли кто-то за это ответить и, если должен, кто именно? Разработчик аппаратного обеспечения? Создатель программного обеспечения? Программист, который написал именно эту часть программы? Человек, который принял решение об использовании этой программы в данном приложении? Как поступить, если данная программа была исправлена компанией — разработчиком программного обеспечения, однако новый вариант не был закуплен и установлен в данном ответственном приложении? Как быть, если используемая программа является пиратской копией?

2. Допустимо ли частному пользователю игнорировать возможность возникновения ошибок усечения и их последствий при разработке прикладных программ для собственного компьютера?
3. Как вы считаете, этично ли было при разработке программ в 1970-х годах использовать всего лишь две цифры для представления года (например, цифры 76 представляют 1976 год) и игнорировать тот факт, что при смене столетия в системе могут возникнуть ошибки? Этично ли сегодня использовать только три цифры для представления года (например, цифры 982 представляют 1982 год, а цифры 015 представляют 2015 год)? Что можно сказать об использовании только четырех цифр?
4. Много споров ведется относительно того, что кодирование так или иначе лишает информацию выразительности или даже разрушает ее, поскольку требует представления данных в количественной форме. Как пример такого отрицательного воздействия кодирования приводится анкета, в которой респондент должен выразить свое мнение посредством указания оценки в диапазоне от одного до пяти, что приводит к определенным изъянам или ошибкам. Так в какой же мере информация поддается количественному измерению? Можно ли представить в количественном виде все "за" и "против" относительно размещения опасного для окружающей среды предприятия? Можно ли количественно выразить споры вокруг ядерной энергетики и связанной с ней опасности радиационного заражения? Опасно ли принимать решения, исходя из результатов усреднения и других типов статистического анализа? Этично ли поступают информационные агентства, сообщая результаты опроса без указания точной словесной формулировки вопроса? Можно ли представить ценность человеческой жизни в количественной форме? Является ли приемлемым для компании запретить вложение инвестиций для повышения качества продукции, если эти вложения помогут снизить риск фатальных случаев при ее использовании?
5. Должны ли существовать различия в правах сбора и распространения данных в зависимости от их особенностей? Другими словами, должно ли право сбора и распространения фотографии, аудио- и видеоинформации быть таким же, как и право сбора и распространения текстовой информации?
6. Любое выступление журналиста, как правило, умышленно или ненамеренно содержит какой-то определенный элемент пристрастного отношения к излагаемой информации. Часто достаточно изменить лишь пару слов, чтобы придать сообщению положительную или отрицательную окраску. (Сравните: "Большинство опрашиваемых не верят в то,

что...” и “Значительная часть опрашиваемых согласились с тем, что ...”.) Существует ли различие между изменением излагаемого материала (путем ухода от острых вопросов или тщательного подбора слов) и изменением фотографии?

7. Предположим, что сжатие данных привело к потере небольшой, но достаточно важной части информации. Какие могут возникнуть проблемы, связанные с ответственностью за это происшествие? Как эти проблемы могут быть решены?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Drew M., Li Z. *Fundamentals of Multimedia*. — Upper Saddle River, NJ: Prentice-Hall, 2004.
2. Halsall F. *Multimedia Communications*. — Boston, MA: Addison-Wesley, 2001.
3. Hamacher V.C., Vranesic Z.G., Zaky S.G. *Computer Organization*. — 5th ed. New York: McGraw-Hill, 2002.
4. Knuth D.E. *The Art of Computer Programming*, Vol. 2, 3rd ed. — Boston, MA: Addison-Wesley, 1998. (Имеется русский вариант этой книги: Кнут Д.Э. *Искусство программирования. Т.2. Получисленные алгоритмы*, 3-е изд. — М.: Издательский дом “Вильямс”, 2000.)
5. Long B. *Complete Digital Photography*, 3rd ed. — Hingham, MA: Charles River Media, 2005.
6. Miano J. *Compressed Image File Formats*. — New York: ACM Press, 1999.
7. Petzold C. *CODE: The Hidden Language of Computer Hardware and Software*. — Redman, WA: Microsoft Press, 2000.
8. Salomon D. *Data Compression: The Complete Reference*, 4th ed. — New York: Springer, 2007.
9. Sayood K. *Introduction to Data Compression*, 3rd ed. — San Francisco: Morgan Kaufmann, 2005.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Столлингс В. *Структурная организация и архитектура компьютерных систем*, 5-е изд. — М.: Издательский дом “Вильямс”, 2002.
2. Найджел Чепмен, Дженни Чепмен. *Цифровые технологии мультимедиа*, 2-е изд. — М.: Компьютерное издательство “Диалектика”, 2006.
3. Скотт Келби. *Цифровая фотография*. — М.: Издательский дом “Вильямс”, 2007.

Глава 2

Основная цель этой главы — ознакомить вас с тем, как компьютеры обрабатывают данные и взаимодействуют с периферийными устройствами, такими как принтер или клавиатура. Однако для этого вам прежде потребуется получить общее представление об архитектуре компьютеров и разобраться в том, как работа компьютера программируется с помощью закодированных инструкций, которые принято называть командами машинного языка.

Обработка данных

2.1. АРХИТЕКТУРА КОМПЬЮТЕРА
- Центральный процессор
- Концепция хранимой программы

2.2. МАШИННЫЙ ЯЗЫК
- Машинные команды
- *Vole*: пример простого машинного языка

2.3. ВЫПОЛНЕНИЕ ПРОГРАММЫ
- Пример выполнения программы
- Программы или данные

*2.4. АРИФМЕТИЧЕСКИЕ И ЛОГИЧЕСКИЕ КОМАНДЫ
- Логические операции
- Операции сдвига
- Арифметические операции

*2.5. ВЗАИМОДЕЙСТВИЕ С ДРУГИМИ УСТРОЙСТВАМИ
- Взаимодействие через управляющее устройство
- Прямой доступ к памяти
- Подтверждение установления связи
- Основные типы соединений
- Скорость передачи данных

*2.6. МАНИПУЛИРОВАНИЕ ДАННЫМИ В ПРОГРАММЕ
- Логические операции и операции сдвига
- Управляющие конструкции
- Функции
- Ввод-вывод
- Программа Marathon Training Assistant

*2.7. ДРУГИЕ ТИПЫ АРХИТЕКТУРЫ КОМПЬЮТЕРОВ
- Конвейерная обработка
- Многопроцессорные машины

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

В главе 1 рассматривались общие концепции методов хранения данных в компьютерной памяти. В этой главе речь пойдет о том, как компьютер манипулирует этими данными. Под манипулированием здесь понимается перемещение данных из одного местоположения в другое, а также выполнение различных операций, например арифметических вычислений, редактирования текста или обработки изображений. Мы начнем с расширения ваших представлений об архитектуре компьютеров, помимо используемых ими систем хранения данных, уже обсуждавшихся в предыдущей главе.

2.1. Архитектура компьютера

Электронные схемы компьютера, предназначенные для выполнения различных манипуляций с данными, в совокупности называют **центральным процессором** или **ЦП** (англ. *Central Processing Unit — CPU*), а в просторечье просто процессором. В машинах середины XX столетия ЦП представлял собой огромное устройство, состоящее из нескольких шкафов с электронными блоками, что только подчеркивало важное значение этого устройства. Однако с развитием технологий размеры ЦП кардинально уменьшились. В современных настольных ПК и ноутбуках центральный процессор представляет собой плоскую квадратную пластину столь малых размеров, что она легко помещается на ладони. Многочисленные выводы этой микросхемы вставляются в специальный разъем (сокет), установленный на основной монтажной плате компьютера, которую называют **материнской платой** (*motherboard*). В смартфонах, планшетах и других мобильных вычислительных устройствах используются центральные процессоры размером в половину почтовой марки. За их малые размеры эти ЦП обычно называют **микропроцессорами**.

Центральный процессор

Типичный центральный процессор состоит из трех частей (рис. 2.1): **арифметико-логического блока**, включающего электронные схемы, выполняющее различные операции с данными (такие, как сложение и вычитание); **блока управления**, объединяющего в себе схемы, координирующие работу всех остальных элементов машины; и **блока регистров**, содержащего ячейки хранения данных (подобные ячейкам основной памяти), называемые **регистрами**, которые предназначены для временного хранения информации в самом ЦП.

В блоке регистров некоторые из регистров называют **регистрами общего назначения**, а все остальные относят к категории **специализированных регистров**. Некоторые из специализированных регистров будут подробно обсуж-

даться в разделе 2.3, а здесь мы ограничимся рассмотрением только регистров общего назначения.

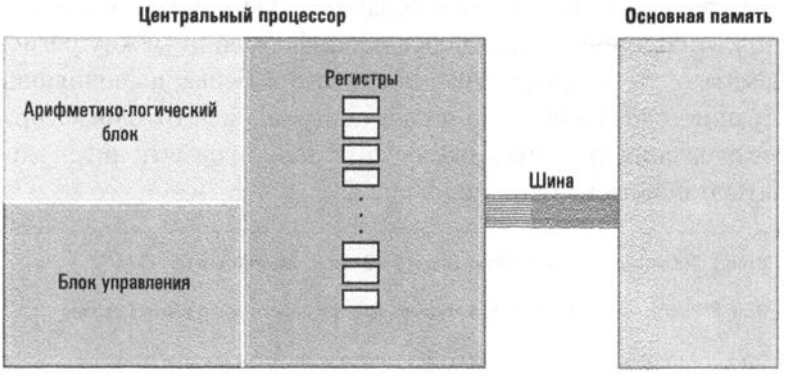


Рис. 2.1. Соединение центрального процессора и основной памяти с помощью шины

Регистры общего назначения используются для временного хранения данных, обрабатываемых в ЦП. В них сохраняются входные данные для схем арифметико-логического блока, а также эти регистры используются для размещения результатов, полученных при выполнении операций. Для обработки информации, сохраняемой в основной памяти машины, блок управления должен организовать передачу данных из памяти в регистры общего назначения, а также указать арифметико-логическому блоку, в каких регистрах содержатся необходимые входные данные, активизировать соответствующие электронные схемы этого блока, а также указать арифметико-логическому блоку тот регистр, в который должен быть помещен результат.

Для передачи битовых комбинаций между ЦП и основной памятью машины эти устройства соединяются группой проводов, которая называется **шиной** (см. рис. 2.1). Именно через эту шину центральный процессор извлекает (или считывает) данные из основной памяти, направляя в нее адрес необходимой ячейки памяти вместе с сигналом считывания, указывающим схемам основной памяти на необходимость извлечь данные из этой ячейки. Аналогичным образом ЦП помещает (или записывает) данные в память, указав адрес ячейки назначения и записываемую информацию, сопровождаемые сигналом записи, сообщающим основной памяти о требовании сохранить эту информацию в указанной ячейке.

Получив представление об этом механизме, можно понять, что даже такая простая операция, как сложение данных, сохраняемых в основной памяти машины, включает гораздо больше действий, чем собственно выполнение операции сложения. Данные должны быть перенесены из основной памяти в

регистры общего назначения в ЦП, значения должны быть просуммированы, а результат, также помещенный в регистр, затем должен быть сохранен в ячейке памяти. Такая процедура требует согласованных совместных действий как блока управления, координирующего передачу информации между регистрами и основной памятью, так и арифметико-логического блока, выполняющего собственно операцию сложения по команде, поступающей от блока управления. Весь процесс сложения двух сохраняемых в основной памяти чисел можно разделить на пять этапов, как показано на рис. 2.2.

Этап 1. Выбрать первое слагаемое из основной памяти и поместить его в регистр.

Этап 2. Выбрать второе слагаемое из основной памяти и поместить его в другой регистр.

Этап 3. Активизировать электронную схему суммирования, указав используемые на этапах 1 и 2 регистры в качестве входных и задав еще один регистр в качестве выходного, предназначенного для размещения результата.

Этап 4. Сохранить результат выполнения операции в основной памяти.

Этап 5. Завершить выполнение операции.

Рис. 2.2. Сложение двух чисел, сохраняемых в основной памяти машины

Концепция хранимой программы

Ранние модели вычислительных устройств не отличались особой гибкостью, так как программы их работы встраивались непосредственно в блок управления как неотъемлемая часть данной машины. Для достижения большей гибкости некоторые ранние электронные компьютеры проектировались так, чтобы блок управления машиной можно было перекоммутировать достаточно удобным способом. В таких случаях в блок управления включали коммутационную панель, напоминающую коммутаторы старинных телефонных станций. Концы коммутируемых линий выводились на штекеры, которые требовалось вставлять в соответствующие контактные гнезда.

Значительный шаг вперед (приписываемый, возможно несправедливо, Джону фон Нейману (John von Neuman)) состоял в осознании того, что программа, как и данные, тоже может быть закодирована в виде последовательности битов и сохранена в основной памяти машины. Если разработать блок управления таким образом, чтобы он был способен извлекать программу из памяти, расшифровывать команды, а затем выполнять их, то программу работы компьютера можно было бы изменять посредством изменения содержимого ячеек его основной памяти, вместо того чтобы перекоммутировать схемы блока управления.



Основные положения для запоминания

- **Последовательность битов может представлять как данные, так и команды управления работой компьютера.**

Идея хранения компьютерной программы в основной памяти компьютера получила название **концепция хранимой программы** и сегодня фактически является стандартным подходом — настолько стандартным, что кажется совершенно очевидной. Сейчас даже трудно представить, что прежде эта идея считалась очень сложной, поскольку многие относили данные и программы к совершенно разным категориям: данные хранились в памяти, а программы являлись частью блока управления. Это как раз тот случай, когда за деревьями не удавалось увидеть лес. В подобные сети очень легко попасть, и развитие компьютерных наук во многом еще зависит от таких несоответствий, хотя мы об этом иногда и не догадываемся. Одной из самых замечательных особенностей науки следует считать то, что каждый новый взгляд на проблему открывает пути к новым теориям и возможностям их применения.

Кеш-память

Будет поучительно сравнить различные категории памяти компьютеров в отношении их функционального назначения. Регистры используются для хранения данных, непосредственно используемых в выполняемых в данный момент операциях, основная память предназначена для хранения данных, которые потребуются в ближайшем будущем, а массовая память служит для долговременного хранения данных, которые, вероятнее всего, в ближайшем будущем не потребуются. Во многих машинах в эту иерархическую структуру включен дополнительный уровень, который называется сверхоперативной памятью или **кеш-памятью**. Кеш — это блок высокоскоростной памяти (чаще всего размером в несколько сот килобайтов), размещенный непосредственно в самом ЦП. В эту специальную область памяти машина стремится скопировать содержимое именно той части ячеек основной памяти, в которой содержатся данные, необходимые для работы в настоящий момент. В случае успеха обмен данными будет осуществляться уже не между регистрами и основной памятью, как это обычно бывает, а между регистрами и кешем. Затем в подходящий момент все внесенные в кеш-память изменения одновременно передаются в основную память машины. В результате ЦП получает возможность выполнять машинные циклы значительно быстрее, поскольку отсутствуют задержки, вызванные обращением к основной памяти машины.

2.1. Вопросы и упражнения

1. Как вы думаете, какая последовательность действий должна быть выполнена машиной для перемещения содержимого одной ячейки основной памяти в другую?
2. Какую информацию должен представить центральный процессор в электронные схемы основной памяти для сохранения числа в одной из ее ячеек?
3. Массовая память, основная память, регистры общего назначения — все это устройства для хранения данных. В чем заключаются основные различия в их использовании?

2.2. Машинный язык

Для реализации концепции хранимой программы ЦП должен быть разработан так, чтобы иметь возможность распознавать определенные битовые комбинации как представления конкретных команд. Весь набор выполняемых операций вместе с системой их кодирования называют **машинным языком**, поскольку он представляет собой средство передачи алгоритмов машине. Команды, закодированные на этом языке, называют командами машинного уровня или просто **машинными командами**.

Машинные команды

Полный список машинных команд, которые типичный ЦП должен уметь декодировать и выполнять, относительно невелик. В действительности, если машина способна выполнять определенный тщательно продуманный набор элементарных операций, то дальнейшее расширение набора команд машины не приведет к увеличению ее теоретических функциональных возможностей. Другими словами, после какого-то момента добавление новых функций позволяет повысить лишь удобство эксплуатации машины, однако никак не влияет на основные ее свойства.

Выбор степени, до которой создаваемые машины будут извлекать преимущества из этого факта, привел к разработке двух типов архитектуры ЦП. При первом подходе предполагается, что ЦП следует проектировать так, чтобы он обеспечивал выполнение лишь минимального набора машинных команд. Этот подход получил название **RISC-архитектура** (Reduced Instruction

Set Computer — *компьютер с ограниченным набором команд*). Аргументами в пользу RISC-архитектуры является то, что эти машины весьма быстродействующи, эффективны и дешевы в производстве. Во втором подходе предпочтение отдается машинам, ЦП которых способен выполнять большое количество сложных команд, пусть даже многие из них будут технически избыточными. Такой подход получил название **CISC-архитектура** (Complex Instruction Set Computer — *компьютер со сложным набором команд*). Основной аргумент в пользу CISC-архитектуры состоит в том, что более сложный процессор позволяет лучше справляться со все возрастающей сложностью современного программного обеспечения. ЦП с архитектурой CISC предоставляет программистам богатый набор мощных команд, реализация многих из которых в ЦП с архитектурой RISC потребует выполнения длинных последовательностей элементарных команд из его ограниченного набора.

Кто придумал это?

Признание определенной личности в качестве изобретателя того или иного новшества всегда является делом сомнительным. Так, Томас Эдисон признан изобретателем электрической лампы накаливания, но в то же самое время и другие исследователи занимались разработкой подобных ламп, и в этом смысле Эдисону просто повезло — он первым получил патент на это изобретение. Братья Райт считаются изобретателями аэропланов, но они конкурировали с другими первопроходцами и смогли извлечь пользу из результатов исследований многих своих современников, каждого из которых, в свою очередь, можно в определенной степени считать лишь последователем Леонардо да Винчи, обдумывавшего возможности построения летательных машин еще в XV столетии. И даже разработки Леонардо, по-видимому, основывались на более ранних идеях. Конечно, во многих случаях общепризнанный изобретатель имеет полное право считаться таковым. Однако в других случаях история, кажется, вознаградила не того, кого следовало, — и хорошим примером здесь может быть концепция хранимой программы. Не вызывает сомнения, что Джон фон Нейман был блестящим ученым и его вклад в науку общепризнан. Однако один из таких весомых вкладов, который история приписывает именно ему, т.е. концепция хранимой программы, в действительности был разработан группой исследователей, возглавляемых Дж.П. Эккертом-младшим (J.P. Eckert, Jr.) из школы Мура в Университете штата Пенсильвания, которую посещал и Джон фон Нейман. Видимо, он просто был первым, кто упомянул эту идею в своей печатной работе, за что компьютерное сообщество и признало его как ее изобретателя.

В 1990-х годах и в начале XXI века представленные на коммерческом рынке CISC- и RISC-процессоры активно конкурировали между собой за главенствующее положение в секторе настольных ПК. Процессоры корпорации

Intel, использовавшиеся многими компаниями в изготавливаемых ими ПК, являлись примером ЦП с CISC-архитектурой. С другой стороны, процессоры PowerPC (разработанные альянсом корпораций Apple, IBM и Motorola) являли собой пример ЦП с RISC-архитектурой и использовались при производстве компьютеров Apple Macintosh. С течением времени стоимость изготовления CISC-процессоров радикально сократилась, вследствие чего процессоры корпорации Intel (и их аналоги от компании AMD — Advanced Micro Devices, Inc.) сейчас можно найти практически во всех настольных компьютерах и ноутбуках (даже Apple сейчас выпускает компьютеры, в которых используется продукция Intel).

Хотя CISC-архитектура успешно сохраняет свои позиции в секторе настольных ПК, ее характерной чертой является чрезмерный расход электроэнергии. В противоположность этому компания Advanced RISC Machine (ARM) разработала процессоры RISC-архитектуры, поставив основной целью снизить потребление ими электроэнергии. (Ранее компания Advanced RISC Machine носила название "Acorn Computers", а сейчас переименована в "ARM Holdings".) В результате разработанные компанией ARM процессоры, изготовленные целой группой ведущих компаний, включая Qualcomm и Texas Instruments, можно найти в игровых приставках, цифровых телевизорах, навигационных устройствах, бортовых автомобильных компьютерах, смартфонах и множестве других типов устройств цифровой бытовой электроники.

Независимо от выбора между RISC- и CISC-архитектурой, машинные команды можно разделить на три категории: команды передачи данных, арифметические и логические команды, а также команды управления.

Команды передачи данных

Группа команд этой категории включает те команды, при выполнении которых происходит перемещение данных из одного места в другое. На рис. 2.2 к этой группе относятся действия, выполняемые на этапах 1, 2 и 4. Следует отметить, что использование таких терминов, как *передача* или *перемещение* в определении этой группы команд, на самом деле является некорректным. Крайне редко встречаются ситуации, когда перемещаемые данные удаляются в месте их исходного расположения. Процедура выполнения команд передачи данных больше напоминает копирование информации с одного места в другое, а не обычное их перемещение. Следовательно, действия этой группы команд точнее было бы определить как *копирование* или *дублирование*.

Раз уж мы здесь коснулись вопросов терминологии, то следует указать, что для передачи данных между ЦП и основной памятью существуют специальные термины. Запрос на заполнение регистра общего назначения содержимым

Команды переменной длины

Из соображений упрощения излагаемого в этой книге материала в машинном языке, используемом для записи примеров в этой главе (и подробно описанном в приложении В), для всех команд принята фиксированная длина — два байта. В результате при выборке команды из основной памяти ЦП всегда извлекает содержимое двух соседних ячеек памяти и увеличивает значение своего счетчика команд на два. Такое постоянство упрощает задачу выборки команд, и эта особенность является характерной для RISC-процессоров. Однако машинный язык CISC-процессоров, как правило, включает команды разной длины. Например, в современных процессорах от Intel используются команды, длина которых может изменяться от одного до многих байтов в случае команд, длина которых зависит от точности представления чисел, с которыми работает данная команда. В ЦП с подобным машинным языком длина следующей выбираемой команды определяется по ее коду операции. Поэтому при выборке очередной команды ЦП сначала извлекает ее код операции, а затем, исходя из полученной комбинации битов, определяет, сколько еще байтов требуется считать из памяти для получения оставшейся части команды.

ячейки памяти обычно называют командой загрузки (LOAD), а запрос на передачу содержимого регистра в ячейку основной памяти — командой сохранения (STORE). На рис. 2.2 этапы 1 и 2 представляют собой команды загрузки, т.е. LOAD, а этап 4 представляет собой команду сохранения — STORE.

В следующую, очень важную, группу команд категории передачи данных входят команды взаимодействия с устройствами, функционирующими вне границ интерфейса *ЦП — основная память* (принтеры, клавиатуры, экраны дисплеев, дисковые накопители и т.д.). Поскольку эти команды отвечают за выполнение в машине операций ввода-вывода, они обычно называются **командами ввода-вывода** и в некоторых случаях помещаются в отдельную категорию. Однако в разделе 2.5 будет показано, что для выполнения операций ввода-вывода обычно используются те же команды, с помощью которых выполняется передача данных между ЦП и основной памятью машины. А это означает, что выделение данных команд в отдельную категорию следует считать неправомерным.

Арифметические и логические команды

Эта группа включает команды, которые передают в блок управления запросы на выполнение определенных действий арифметико-логического блока. На рис. 2.2 к этой категории относятся действия, выполняемые на этапе 3. Как следует из самого названия арифметико-логического блока, он также предусматривает выполнение группы операций, отличающихся от основных

арифметических действий. К ним относятся, например, обычные логические операции И (AND), ИЛИ (OR) и исключающее ИЛИ (XOR), которые мы уже рассматривали в главе 1. В этой главе мы обсудим эти операции более подробно.

Другая группа операций, реализованная в большинстве типов арифметико-логических блоков, состоит из команд, позволяющих перемещать содержимое регистров влево или вправо в пределах самих этих регистров. Такие операции называются операциями сдвига (SHIFT) или вращения (ROTATE), в зависимости от того, что происходит с битами, выходящими при перемещении содержимого регистра за его пределы. При операции сдвига (SHIFT) эти биты просто отбрасываются, а при операции вращения (ROTATE) биты, покидающие пределы регистра с одного конца, помещаются в освободившиеся позиции на другом конце регистра, поэтому в русскоязычной литературе подобная операция называется *циклическим сдвигом* — именно этим термином мы и будем пользоваться в дальнейшем.

Команды управления

Команды этой группы предназначены для управления ходом выполнения программы, а не обработки каких-либо данных. На рис. 2.2 к этой категории относятся действия, выполняемые на этапе 5, однако это очень простой пример. Данная категория включает много интересных команд, например группу команд перехода (JUMP) или ветвления (BRANCH). Они используются для перенаправления управляющего блока на выполнение команды, отличной от той, которая является очередной в выполняемой последовательности. Команды перехода реализуются в двух вариантах: команды **безусловного перехода** и команды **условного перехода**. К первому варианту относится команда типа "Пропустите все команды до этапа 5", а ко второму — команда типа "Если полученное число равно 0, то перейдите к этапу 5". Разница между ними состоит в том, что при выполнении команды условного перехода изменение последовательности произойдет только при выполнении указанного условия. В качестве примера можно привести последовательность команд (рис. 2.3), которая представляет собой реализацию алгоритма деления двух чисел. В этом примере этап 3 содержит команду условного перехода, предназначенную для предотвращения операции деления на нуль.

Этап 1. Загрузить (LOAD) в регистр число из основной памяти.

Этап 2. Загрузить (LOAD) в другой регистр еще одно число из основной памяти.

Этап 3. Если второе число равно нулю, перейти (JUMP) к этапу 6.

Этап 4. Разделить содержимое первого регистра на содержимое второго и записать результат в третий регистр.

Этап 5. Запомнить (STORE) содержимое третьего регистра в основной памяти.

Этап 6. Завершить (STOP) выполнение операции.

Рис. 2.3. Деление чисел, сохраняемых в основной памяти

Vole: пример простого машинного языка

Теперь давайте посмотрим, как можно закодировать команды в типичной вычислительной машине. Машину, которая используется в нашем примере, мы будем называть *Vole*, — она подробно описана в приложении B и схематично показана на рис. 2.4. Гипотетический процессор машины Vole имеет 16 регистров общего назначения и 256 ячеек основной памяти, каждая из которых имеет длину восемь бит. Для целей адресации присвоим регистрам номера от 0 до 15, а адреса ячеек основной памяти обозначим числами от 0 до 255. Для удобства будем полагать эти номера и адреса представленными в двоичной системе счисления, причем на письме эти достаточно длинные битовые комбинации будут сжаты в шестнадцатеричные числа. В результате регистры общего назначения будут иметь номера от `0x0` до `0xF`, а ячейки памяти — адреса от `0x00` до `0xFF`. (Вспомните, что в главе 1 мы приняли за правило всегда использовать префикс "`0x`" для обозначения шестнадцатеричных значений.)

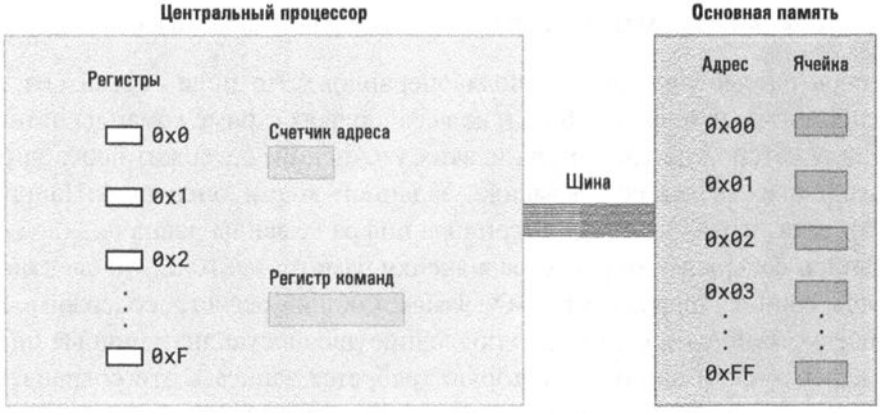


Рис. 2.4. Архитектура машины Vole, описанной в приложении B

Кодированное представление машинной команды обычно состоит из двух частей: поля **кода операции** (op-code — сокращение от "operation code") и **поля операндов**. Битовая комбинация, помещаемая в поле кода операции, определяет ту элементарную операцию (например, `STORE`, `SHIFT`, `XOR` или `JUMP`), выполнение которой предусматривается данной командой. Битовые комбинации в поле операндов предоставляют более детальную информацию о той операции, которая задана в поле кода операции. Например, при выполнении операции `STORE` информация в поле операндов указывает регистр, в котором содержатся предназначенные для сохранения данные, а также ту ячейку основной памяти, в которую эти данные должны быть записаны.

Весь машинный язык машины Vole включает только 12 основных команд. Каждая из них кодируется в 16 битах, представляемых в листингах четырьмя шестнадцатеричными цифрами (рис. 2.5). Код операции для каждой команды размещается в первых ее четырех битах и представляется одной шестнадцатеричной цифрой. Поэтому полный перечень команд машины Vole (см. приложение В) включает двенадцать базовых команд, коды операций которых представляются шестнадцатеричными цифрами от `0x1` до `0xC`. В частности, согласно таблице в приложении В, любая команда, код которой начинается с шестнадцатеричной цифры `0x3` (битовая комбинация `0011`), является командой сохранения `STORE`, а каждая команда, код которой начинается с шестнадцатеричного символа `0xA`, является командой циклического сдвига `ROTATE`.

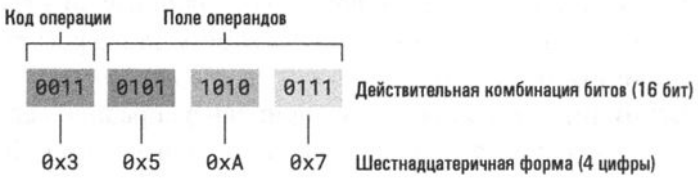


Рис. 2.5. Формат команды вычислительной машины Vole

А теперь рассмотрим формат поля операндов. Это поле состоит из трех шестнадцатеричных цифр (12 бит) и во всех случаях (кроме команды остановки `HALT`, для которой не требуется никаких уточнений) содержит необходимые дополнительные сведения о команде, заданной кодом операции. Например (рис. 2.6), если первая шестнадцатеричная цифра команды равна `0x3` (код операции записи содержимого регистра в ячейку памяти — `STORE`), то следующая шестнадцатеричная цифра команды указывает общий регистр, содержимое которого следует записать в память, а последние две шестнадцатеричные цифры задают адрес ячейки памяти, в которую требуется записать эти сохраняемые данные. Таким образом, команду `0x35A7` можно расшифровать как указание: "Сохранить комбинацию битов из регистра 5 в ячейку памяти с адресом `0xA7`". (Обратите внимание, как использование шестнадцатеричной нотации упрощает

нашу задачу. В действительности команда `0x35A7` представляет собой последовательность битов `0011010110100111`.)

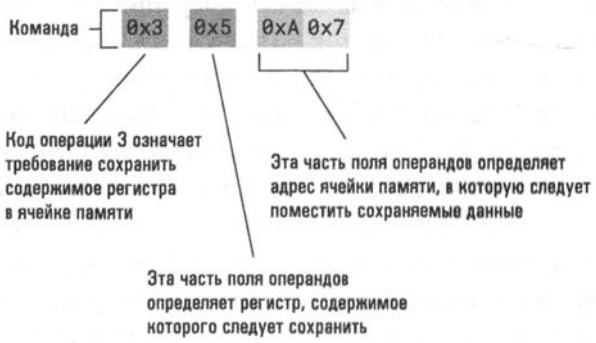


Рис. 2.6. Декодирование команды `0x35A7`

Основные положения для запоминания

- **Причина, по которой шестнадцатеричная (с основанием 16) нотация используется для представления цифровых данных, состоит в том, что в шестнадцатеричном представлении любого числа используется в четыре раза меньше цифр, чем в его двоичном представлении.**

Команда `0x35A7` также представляет собой убедительный пример того, почему емкость основной памяти компьютеров следует измерять в степенях двойки. Поскольку для определения адреса требуемой ячейки памяти в команде зарезервировано 8 битов, это поле позволяет обратиться ровно к 2^8 различным ячейкам памяти. Следовательно, нам надлежит построить для этой машины основную память с указанным количеством ячеек, адреса которых будут находиться в диапазоне от 0 до 255. Если в основной памяти будет больше ячеек, у нас не будет возможности написать команду, позволяющую обратиться к этим избыточным ячейкам, а если ячеек памяти будет меньше, появится возможность записать команду, которая будет обращаться к несуществующим ячейкам.

Рассмотрим еще один пример того, как поле операндов может использоваться для уточнения тех действий, которые должна выполнить команда, заданная своим кодом операции. Пусть код операции будет представлен шестнадцатеричной цифрой `0x7`, что указывает на логическую операцию `OR`, которую следует выполнить над содержимым двух регистров общего назначения. (Подробно о том, как операция `OR` выполняется над содержимым двух регистров, речь пойдет в разделе 2.4. На данный момент нас интересует только то, как соответствующая команда декодируется.) В данном случае первая шестнадцатеричная

цифра поля операндов задает регистр, в который следует поместить результат выполнения операции, а последние две цифры определяют регистры, над содержимым которых выполняется логическая операция OR. Следовательно, команду 0x70C5 можно расшифровать так: "Выполнить операцию OR над содержимым регистров 12 и 5, а полученный результат поместить в регистр 0".

В машинном языке Vole существуют две команды LOAD, между которыми имеется небольшое различие. Так, код операции 0x1 определяет команду загрузки в регистр общего назначения содержимого указанной ячейки основной памяти, тогда как код операции 0x2 относится к команде загрузки в регистр общего назначения заданного числового значения. Различие заключается в том, что поле операндов в команде первого типа содержит *адрес*, тогда как в команде второго типа поле операндов содержит *число*, т.е. ту битовую комбинацию, которую требуется загрузить в регистр.



Основные положения для запоминания

- **Интерпретация смысла последовательности битов зависит от того, как ее предполагается использовать.**

Также обратите внимание, что в машине Vole есть две команды сложения (ADD): одна — для сложения чисел в двоичном дополнительном коде, а другая — для сложения чисел в формате с плавающей точкой. Такое разделение обусловлено тем, что сложение чисел в двоичном дополнительном коде потребует от арифметико-логического блока выполнения совершенно иных действий, чем в случае сложения чисел в формате с плавающей точкой. Для регистров и ячеек основной памяти не существует каких-либо различий в отношении того, какого формата данные в них хранятся, — это всегда просто последовательность битов. Как именно интерпретируется эта последовательность битов, полностью определяется применяемой к ним *командой*. И наконец, в завершение этого раздела вашему вниманию предлагается рис. 2.7, на котором представлена закодированная версия той же последовательности команд, которая ранее была предложена на рис. 2.2. В данном случае предполагается, что суммируемые значения сохранены в двоичном дополнительном коде в ячейках памяти с адресами 0x6C и 0x6D, а полученную сумму требуется поместить в ячейку памяти с адресом 0x6E.



Основные положения для запоминания

- **Последовательность битов в разных контекстах может представлять собой данные различных типов.**

Коды команд	Расшифровка команд
0x156C	Загрузить в регистр 0x5 комбинацию битов, хранящуюся в ячейке памяти с адресом 0x6C
0x166D	Загрузить в регистр 0x6 комбинацию битов, хранящуюся в ячейке памяти с адресом 0x6D
0x5056	Сложить содержимое регистров 0x5 и 0x6, полагая, что это числа в двоичном дополнительном коде, и поместить результат в регистр 0x0
0x306E	Сохранить содержимое регистра 0x0 в ячейке памяти с адресом 0x6E
0xC000	Стоп. Завершение работы программы

Рис. 2.7. Закодированная на машинном языке версия последовательности команд, приведенной на рис. 2.2

2.2. Вопросы и упражнения

1. Почему термин *перемещение* следует считать некорректным для обозначения операции перемещения данных в машине из одного места в другое?
2. В тексте этого раздела команда перехода (JUMP) была записана так, что требуемое место передачи управления явно указывалось в ней с помощью имени (или номера этапа), например "Перейдите к этапу 6". Недостатком данного способа записи является то, что если имя (или номер этапа) адресуемой команды позднее будет изменено, потребуется найти и исправить все команды перехода на эту команду, имеющиеся в программе. Предложите другой способ записи команд перехода (JUMP), не содержащий явного указания имени адресуемой команды.
3. Как вы считаете, команда "Если 0 равен 0, то перейдите к этапу 7" является командой условного или безусловного перехода? Поясните свой ответ.
4. Запишите программу, приведенную как пример на рис. 2.7, в виде соответствующих битовых комбинаций.

5. Ниже представлены команды, записанные на машинном языке Vole. Приведите текстовую формулировку этих команд.

   а. `0x368A` **б.** `0xBADE` **в.** `0x803C` **г.** `0x40F4`

6. В чем состоит различие между командами `0x15AB` и `0x25AB`, записанными на машинном языке Vole?

7. Ниже дано текстовое представление нескольких машинных команд. Запишите эти команды на машинном языке Vole.

   а. Загрузить (LOAD) в регистр `0x3` шестнадцатеричное число `0x56`.

   б. Циклически сдвинуть (ROTATE) содержимое регистра `0x5` на три бита вправо.

   в. Выполнить операцию AND над содержимым регистров `0xA` и `0x5`, а ее результат поместить в регистр `0x0`.

2.3. Выполнение программы

Компьютер выполняет хранимую в его памяти программу посредством копирования по мере необходимости команд из основной памяти в ЦП. Как только команда попадает в ЦП, она декодируется, после чего выполняется. Порядок, в котором команды выбираются из памяти, соответствует порядку их размещения в памяти, за исключением случаев выполнения команды перехода (JUMP).

Чтобы понять общий процесс выполнения команд, сначала необходимо познакомиться двумя специализированными регистрами ЦП: **счетчиком адреса** и **регистром команд** (см. рис. 2.4). Регистр команд используется для хранения кода выполняемой в данный момент команды. Счетчик адреса содержит адрес команды, которая будет выполнена следующей, т.е. он предназначен для наблюдения за ходом выполнения программы.

ЦП работает в режиме постоянного повторения алгоритма, называемого **машинным циклом**, состоящего из трех этапов: *выборки*, *декодирования* и *выполнения* (рис. 2.8). На этапе выборки ЦП извлекает из основной памяти ту команду, адрес которой в данный момент находится в счетчике адреса. Поскольку каждая команда в нашей машине имеет длину два байта, процесс выборки включает считывание содержимого двух последовательных ячеек основной памяти. ЦП помещает считанную команду в регистр команд, а затем увеличивает значение в счетчике адреса на два, чтобы этот регистр содержал адрес следующей сохраненной в памяти команды. Таким образом, счетчик адреса оказывается полностью подготовленным к следующему этапу выборки.



Рис. 2.8. Схема машинного цикла

Поскольку команда уже находится в регистре команд, ЦП переходит к этапу ее декодирования. Эта процедура предусматривает разбиение поля операндов на соответствующие составляющие элементы, исходя из кода операции данной команды.

Затем ЦП выполняет команду посредством активизации соответствующей схемы, предназначенной для выполнения поставленной задачи. Например, если команда представляет собой операцию загрузки данных из основной памяти, ЦП отправляет соответствующие сигналы в основную память, ожидает, пока из памяти поступят требуемые данные, а затем помещает их в указанный регистр. Если же команда предусматривает выполнение арифметической операции, то ЦП активизирует соответствующую схему в арифметико-логическом блоке, передавая ей в качестве входных данных номера заданных в команде регистров, ожидает, пока арифметико-логический блок проведет вычисления, а затем помещает результат в указанный в команде регистр.

Когда обработка команды будет завершена, ЦП вновь начнет выполнение алгоритма машинного цикла с этапа выборки. Напомним, поскольку в конце предыдущего этапа выборки счетчик адреса был увеличен, он по-прежнему предоставляет блоку управления корректный адрес следующей выполняемой команды.

Особым случаем является выполнение команды перехода JUMP. Давайте рассмотрим выполнение команды с кодом `0xB258` (рис. 2.9), которая имеет следующий смысл: "Выполнить переход к команде, сохраняемой по адресу `0x58`,

если содержимое регистра `0x2` идентично содержимому регистра `0x0`". В этом случае этап выполнения в машинном цикле начинается со сравнения содержимого регистров `0x2` и `0x0`. Если оно различно, то этап выполнения этой команды завершается и начинается выполнение нового машинного цикла. Если же содержимое указанных регистров одинаково, то машина на данном этапе выполнения поместит в счетчик адреса значение `0x58`, в результате чего на этапе выборки *следующего* машинного цикла блок управления обнаружит в счетчике адреса значение `0x58` и следующей в регистр команд будет загружена, а затем выполнена команда, расположенная по этому адресу.

Сравнение производительности компьютеров

Приступив к поиску подходящего персонального компьютера, вы обнаружите, что для сравнения производительности различных вычислительных машин часто используется такой параметр, как частота сигнала синхронизации (тактовая частота компьютера). **Тактовая частота** задается компьютерными часами, которые представляют собой генератор электрических колебаний, импульсы которого используются для согласования всех действий в компьютере: чем выше частота импульсов синхронизации, тем быстрее машина выполняет введенные в нее задания. Частота сигналов синхронизации измеряется в герцах (сокращенно — Гц); 1 Гц соответствует выполнению одного цикла (или появлению одного импульса) за секунду. Как правило, тактовая частота в обычном компьютере колеблется в диапазоне от сотен тысяч мегагерц (МГц) в более старых моделях до нескольких гигагерц (ГГц). Здесь 1 МГц равен одному миллиону герц, а один ГГц — 1000 МГц или одному миллиарду герц.

К сожалению, центральные процессоры производятся многими фирмами, каждая из которых устанавливает собственный объем работы, выполняемой за один цикл синхронизации. По этой причине сравнение производительности машин по тактовой частоте их центральных процессоров, имеющих разные конструкции, следует считать некорректным. Если необходимо сравнить производительности компьютера, в котором установлен процессор от Intel, и компьютера, в котором используется ЦП от ARM, то разумнее будет выполнить измерение показателей производительности с помощью специальных тестовых заданий (**benchmark**). В этом случае сравнивается реальная скорость выполнения каждым из компьютеров одного и того же задания, называемого *тестовым*. Подобрав тестовое задание, которое может служить эталоном для некоторого типа приложений, мы будем иметь инструмент сравнения, позволяющий получить осмысленную оценку коммерческих продуктов, представленных в различных сегментах рынка.

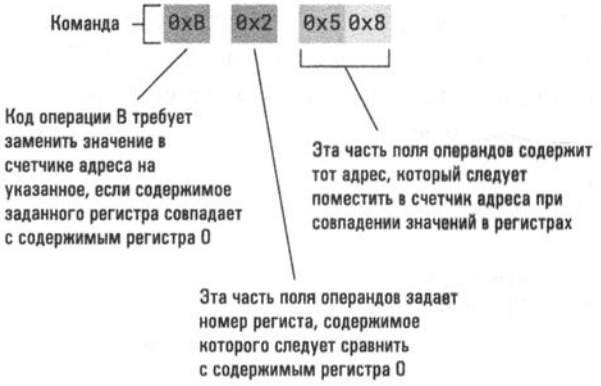


Рис. 2.9. Расшифровка команды `0xB258`

Обратите внимание, что если команда будет иметь вид `0xB058`, то решение о том, следует ли изменить значение в счетчике команд, будет приниматься на основе результатов сравнения содержимого регистра `0x0` с содержимым регистра, опять же, `0x0`. Поскольку это один и тот же регистр, его содержимое всегда будет равно самому себе. А это означает, что по команде, имеющей вид `0xB0XY`, всегда будет выполняться переход к команде, расположенной по адресу `0xXY`, независимо от того, что именно находится в регистре 0. Иначе говоря, такая команда воспринимается машиной как команда *безусловного перехода*.

Пример выполнения программы

Давайте проследим за выполнением всех машинных циклов, предусматриваемых программой, текст которой представлен на рис. 2.7. В этой программе два числа выбираются из основной памяти, суммируются, а затем результат записывается в указанную ячейку основной памяти. Прежде всего необходимо разместить нашу программу где-либо в памяти машины. Например, будем считать, что текст этой программы занесен в последовательные ячейки памяти, начиная с адреса `0xA0`. Если записать программу таким способом, то можно заставить машину выполнить ее, поместив адрес первой команды программы (`0xA0`) в счетчик адреса и запустив машину в работу (рис. 2.10).

Свою работу ЦП начинает с извлечения из основной памяти команды, записанной по адресу `0xA0`, и помещает ее код (`0x156C`) в регистр команд (рис. 2.11, *а*). Напомним, что в нашей машине длина команды составляет шестнадцать битов (2 байта), а это значит, что выбираемая команда должна занимать две ячейки памяти с адресами `0xA0` и `0xA1`. Конструкция ЦП разработана с учетом этой особенности, поэтому он выбирает содержимое обеих ячеек и помещает данные в регистр команд, размер которого составляет 16 бит. Затем ЦП добавляет

число 2 к значению в счетчике адреса, чтобы содержимое этого регистра представляло собой адрес следующей команды (рис. 2.11, *б*). По завершении этапа выборки первого машинного цикла в счетчике адреса и регистре команд будут содержаться следующие данные:

```
Счетчик адреса: 0xA2
Регистр команд: 0x156C
```

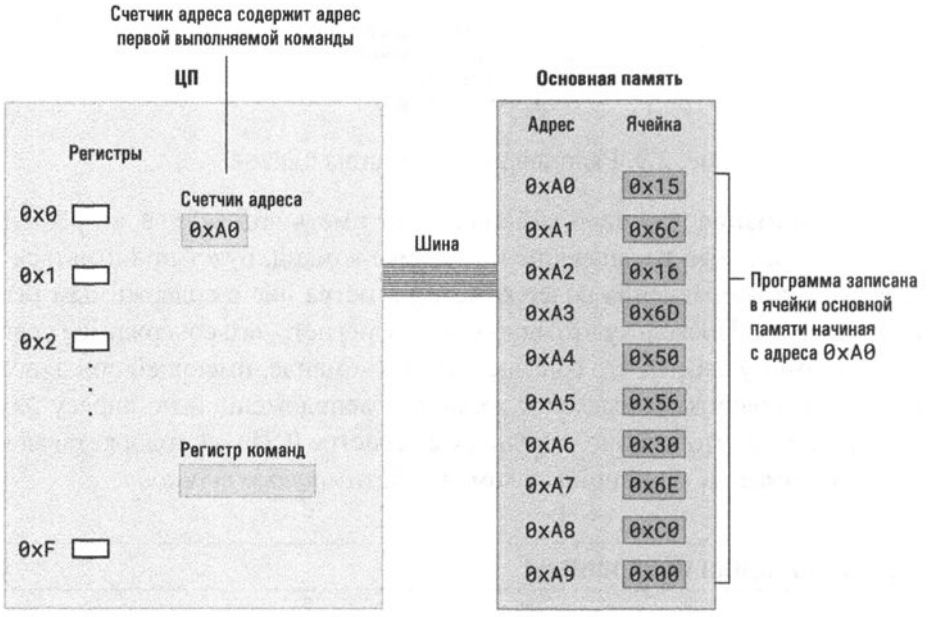
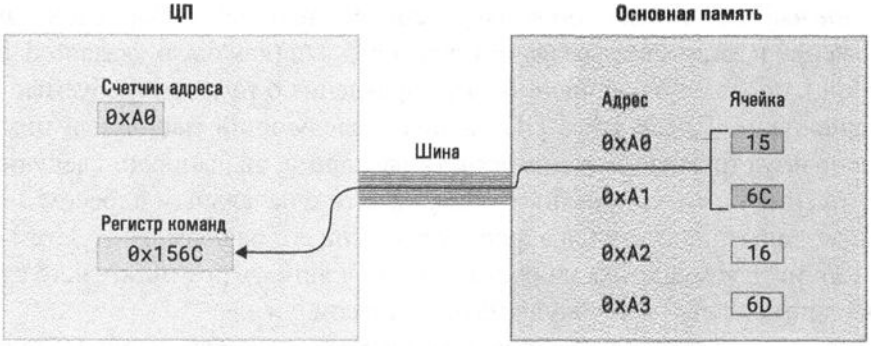


Рис. 2.10. Программа, представленная на рис. 2.7, записана в основную память и готова к выполнению

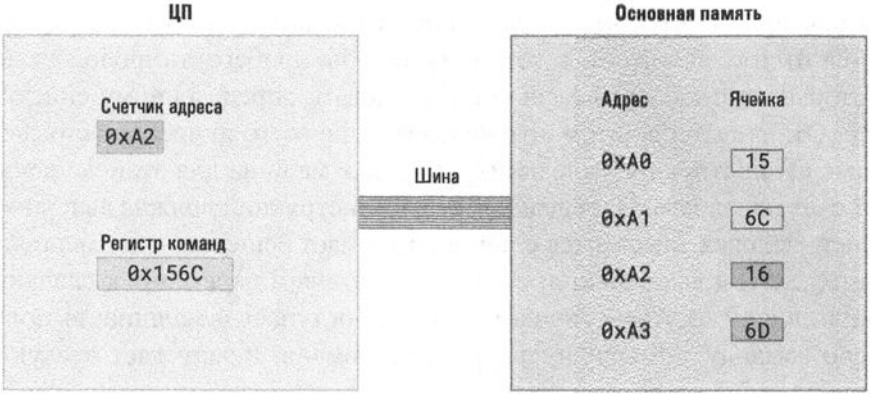
На следующем этапе ЦП анализирует команду, помещенную в регистр команд, и приходит к заключению, что это команда загрузки в регистр `0x5` содержимого ячейки памяти с адресом `0x6C`. Загрузка осуществляется на этапе выполнения данного машинного цикла, после чего блок управления начинает новый машинный цикл.

Этот цикл начинается с выборки команды `0x166D` из двух ячеек памяти начиная с адреса `0xA2`. ЦП помещает эту команду в регистр команд и увеличивает значение счетчика адреса, после чего оно становится равным `0xA4`. После завершения очередного этапа выборки в счетчике адреса и регистре команд будут следующие данные:

```
Счетчик адреса: 0xA4
Регистр команд: 0x166D
```



а) В начале выполнения этапа выборки команда, записанная в ячейках памяти начиная с адреса 0xA0, извлекается из памяти и помещается в регистр команд



б) Далее содержимое счетчика команд увеличивается таким образом, чтобы он указывал на начало следующей команды

Рис. 2.11. Выполнение этапа выборки первого машинного цикла

Далее ЦП декодирует команду `0x166D` и определяет, что в регистр `0x6` необходимо загрузить содержимое ячейки памяти с адресом `0x6D`, после чего выполняет эту команду, и в регистре `0x6` в конечном счете оказывается требуемое значение.

Поскольку в данный момент счетчик адреса имеет значение `0xA4`, ЦП считывает следующую команду, которая начинается с указанного адреса. В результате в регистр команд загружается значение `0x5056`, а счетчик адреса получает новое значение — `0xA6`. Блок управления декодирует содержимое этой команды и выполняет ее посредством активизации электронной схемы сложения чисел в дополнительном двоичном коде, указав этой схеме использовать как входные регистры `0x5` и `0x6`.

На этапе выполнения данной команды арифметико-логический блок выполняет сложение и записывает результат в регистр `0x0` (как было указано блоком управления), после чего сообщает блоку управления о том, что требуемые действия выполнены. После этого ЦП начинает следующий машинный цикл. И вновь, используя текущее значение в счетчике адреса, он выбирает следующую команду (`0x306E`) из двух смежных ячеек памяти с начальным адресом `0xA6` и увеличивает значение счетчика адреса, установив его равным `0xA8`. Затем считанная из памяти команда декодируется и выполняется, в результате чего сумма двух чисел помещается в ячейку памяти с адресом `0x6E`.

Следующая выбираемая команда расположена в памяти начиная с адреса `0xA8`. После ее извлечения значение счетчика адреса увеличивается до `0xAA`. Содержимое регистра команд (`0xC000`) расшифровывается как команда останова. В результате работа машины останавливается на этапе выполнения, и это означает, что выполнение программы завершено.

В заключение скажем, что выполнение хранимой в памяти программы не отличается от того процесса, к которому мог бы прибегнуть любой человек, перед которым поставлена задача четко следовать определенному списку инструкций. Отличие лишь в том, что человек чаще всего отмечает в списке выполненные им инструкции галочками, тогда как машина для этой же цели использует счетчик адреса. Определив, какая из инструкций должна выполняться следующей, человек знакомится с ней и принимает решение, что именно нужно сделать, а затем выполняет требуемые действия и переходит к следующей инструкции в списке. Аналогичным образом поступает и машина: выполняет очередную команду, помещенную в регистр команд, и запускает следующий цикл выборки.

Программы или данные

В основной памяти компьютера можно одновременно разместить множество программ, выделив для каждой различные области памяти. Тем, какая из этих программ начнет выполняться при запуске машины, можно легко управлять, просто установив соответствующим образом исходное значение счетчика адреса.

Однако не следует забывать, что данные также содержатся в основной памяти и кодируются с помощью нулей и единиц, поэтому машина сама по себе не может установить, что именно является данными, а что — программой. Если в счетчике адреса вместо адреса требуемой программы будет установлен адрес данных, то ЦП не сможет предпринять никаких иных действий, кроме как рассматривать битовые комбинации данных так, как если бы они были командами, и попытаться их выполнить. Полученный результат непредсказуем и будет зависеть от того, с какими именно данными работала машина.

Тем не менее нельзя сказать, что мы поступаем неверно, используя и для программ, и для данных одинаковый тип представления в памяти. В действительности это дает большие преимущества, поскольку, благодаря такому подходу, одна программа может работать с другими программами (и даже с самой собой) как с обычными данными. Например, можно представить себе программу, которая в результате взаимодействия с окружающей средой изменяет саму себя, получая, таким образом, возможность обучаться. Или другой пример — программа, которая пишет и выполняет другие программы, используя их как средства решения поставленной перед ней задачи.

2.3. Вопросы и упражнения

1. Предположим, что в памяти машины Vole в ячейках с адресами от `0x00` до `0x05` содержатся приведенные ниже битовые комбинации.

Адрес	Содержимое
`0x00`	`0x14`
`0x01`	`0x02`
`0x02`	`0x34`
`0x03`	`0x17`
`0x04`	`0xC0`
`0x05`	`0x00`

   Если запустить машину, предварительно установив в счетчике адреса значение `0x00`, то какая битовая комбинация окажется в ячейке памяти с адресом `0x17`, когда машина остановится?

2. Предположим, что в памяти машины Vole в ячейках с адресами от `0xB0` до `0xB8` содержатся приведенные ниже битовые комбинации.

Адрес	Содержимое
`0xB0`	`0x13`
`0xB1`	`0xB8`
`0xB2`	`0xA3`
`0xB3`	`0x02`
`0xB4`	`0x33`
`0xB5`	`0xB8`
`0xB6`	`0xC0`
`0xB7`	`0x00`
`0xB8`	`0x0F`

   а. Если в начале работы в счетчике адреса будет находиться значение `0xB0`, то какая битовая комбинация будет содержаться в регистре `0x3` после выполнения первой команды?

   б. Какая битовая комбинация будет находиться в ячейке памяти с адресом `0xB8` после выполнения команды останова?

3. Предположим, что в памяти машины Vole в ячейках с адресами от `0xA4` до `0xB1` содержатся приведенные ниже битовые комбинации.

Адрес	Содержимое
`0xA4`	`0x20`
`0xA5`	`0x00`
`0xA6`	`0x21`
`0xA7`	`0x03`
`0xA8`	`0x22`
`0xA9`	`0x01`
`0xAA`	`0xB1`
`0xAB`	`0xB0`
`0xAC`	`0x50`
`0xAD`	`0x02`
`0xAE`	`0xB0`
`0xAF`	`0xAA`
`0xB0`	`0xC0`
`0xB1`	`0x00`

Отвечая на следующие вопросы, исходите из того, что в начале работы счетчик адреса содержит значение `0xA4`.

а. Какое значение будет находиться в регистре `0x0` после первого выполнения команды, расположенной в ячейке с адресом `0xAA`?

б. Что будет находиться в регистре `0x0` после второго выполнения команды, расположенной в ячейке с адресом `0xAA`?

в. Сколько раз должна быть выполнена команда, расположенная в ячейке с адресом `0xAA`, прежде чем машина Vole остановится?

4. Предположим, что в памяти машины Vole в ячейках с адресами от `0xF0` до `0xF9` содержатся приведенные ниже битовые комбинации.

Адрес	Содержимое
`0xF0`	`0x20`
`0xF1`	`0xC0`
`0xF2`	`0x30`
`0xF3`	`0xF8`
`0xF4`	`0x20`
`0xF5`	`0x00`
`0xF6`	`0x30`
`0xF7`	`0xF9`
`0xF8`	`0xFF`
`0xF9`	`0xFF`

Если в начале работы машины счетчик адреса содержит значение `0xF0`, то какими будут действия машины, когда она достигнет команды, записанной в ячейке с адресом `0xF8`?

2.4. Арифметические и логические команды

Как говорилось ранее, группа арифметических и логических команд состоит из таких команд, которые требуют выполнения некоторых арифметических операций, логических операций или операций сдвига. В этом разделе мы познакомимся с этими командами более подробно.

Логические операции

В первой главе логические операции AND (И), OR (ИЛИ) и XOR (исключающее ИЛИ) были представлены как операции, которые комбинируют значения двух входных двоичных разрядов в целях получения одного двоичного разряда на выходе. Эти операции могут быть расширены, т.е. могут рассматриваться как операции, комбинирующие значения двух **строк битов** для получения одной строки битов на выходе, что достигается посредством применения соответствующей базовой операции к отдельным позициям в строках. Например, результат применения операции AND к строкам битов 10011010 и 11001001 будет следующим.

```
    10011010
AND 11001001
    10001000
```

Другими словами, при выполнении этой строковой операции результат операции AND для двух битов в каждой позиции исходных строк просто записывается в этой же позиции строки результата. Аналогичным образом при выполнении операций OR и XOR с теми же входными строками битов будут получены следующие результаты.

```
   10011010        10011010
OR 11001001    XOR 11001001
   11011011        01010011
```

Операции AND чаще всего используются для помещения нулей в некоторую часть битовой комбинации (не затрагивая при этом другую ее часть). Например, давайте посмотрим, что произойдет, если байт 00001111 использовать в качестве первого операнда логической операции AND. Даже не имея никакой информации о втором операнде, можно сразу же сделать вывод, что четыре старших бита строки результата всегда будут равны 0. Более того, можно также заранее утверждать, что четыре младших бита строки результата будут копиями соответствующих битов второго операнда, что непосредственно подтверждается следующим примером.

```
    00001111
AND 10101010
    00001010
```

Подобное применение операции AND является примером процедуры, называемой **маскированием**. Первый операнд, или **маска**, определяет ту часть второго операнда, которая будет способна оказать воздействие на результат операции. В случае операции AND результатом маскирования будет частичная копия второго операнда, в которой нули заполняют те позиции, которые не подлежат дублированию. Одним из типичных примеров использования операции AND в подобном контенте может быть маскирование всех битов, представляющих красную компоненту цвета пикселя в изображении, — в результате она полностью аннулируется и в цвете данного пикселя остаются только синяя и зеленая компоненты. Подобные преобразования широко используются как один из инструментов, предоставляемых в программах обработки изображений.

Помимо обработки изображений, операции AND будут полезны и при работе с различными иными **битовыми отображениями**, например со строками битов, в которых каждый бит представляет наличие или отсутствие определенного объекта. Примером может служить 52-разрядная битовая строка, в которой каждый бит ассоциируется с некоторой игральной картой. Данное представление может использоваться для описания карт, которые были сданы игроку в покер. В этой строке битов в единичном состоянии будут находиться только те пять битов, которые ассоциируются с картами, полученными игроком при раздаче, все остальные биты будут иметь значения 0. Аналогичная 52-разрядная битовая строка, в которой уже 13 битов будут равны 1, может представлять отдельного игрока в бридж. В то же время в ином приложении 32-разрядная битовая строка может представлять 32 вида мороженого, выпускаемого изготовителем.

Предположим, что восьмиразрядная ячейка памяти используется как битовое отображение и мы хотим убедиться, что объект, представленный третьим битом со старшего конца, имеется в наличии. Для этого достаточно выполнить операцию AND, использовав в качестве операндов весь байт битового отображения и маску 00100000. В результате будет получен байт, содержащий все нули тогда и только тогда, когда третий со старшего конца бит исходного отображения равен 0. В этом случае в программе можно предпринять необходимые действия посредством помещения после данной команды AND соответствующей команды условного перехода. Если же третий со старшего конца бит растрового изображения равен 1, а нам требуется изменить его значение на 0 без изменения состояния других битов, достаточно выполнить операцию AND, используя в качестве операндов весь байт с битовым отображением и маску 11011111, а затем записать результат в ту ячейку, в которой хранился исходный байт битового отображения.

Если операция AND может использоваться для дублирования части битовой строки с помещением нулей во все биты другой ее части, то операция OR может

применяться для дублирования части строки с помещением во все ее оставшиеся биты *единиц*. Для этого также используется определенная маска, но на этот раз позиции, значения которых должны быть продублированы, отмечаются в ней нулями, а единицами заполняются все остальные позиции, не подлежащие дублированию. Приведем пример. При выполнении операции OR с любым байтом и битовой комбинацией `11110000` будет получен результат, в котором четыре старших бита всегда будут содержать единицы, тогда как младшие биты будут просто копиями битов исходного операнда, что демонстрируется следующим примером.

```
   11110000
OR 10101010
   11111010
```

Таким образом, если операция AND с маской `11011111` может использоваться для того, чтобы поместить значение `0` в третий со старшего конца бит некоторого восьмиразрядного битового отображения, то операция OR с маской `00100000` может применяться для установки этого же бита в единицу.

Операция XOR чаще всего используется для создания строки дополнения к некоторой строке битов, — этого результата можно достичь, если применить эту операцию к байту и маске из всех единиц. Например, обратите внимание на взаимосвязь между вторым операндом и строкой результата в следующем примере.

```
   11111111
XOR 10101010
   01010101
```

Как видите, операция XOR между любым байтом и байтом, содержащим все `1`, дает в результате байт дополнения к исходному байту. Такие операции могут, например, использоваться для инвертирования всех битов в представлении изображения в виде растра в формате RGB. Результатом будет "инвертирование" цветов изображения, при котором каждый светлый оттенок заменяется темным и наоборот.

В языке машины Vole (см. приложение B) коды операций `0x7`, `0x8` и `0x9` используются для обозначения логических операций OR, AND и XOR соответственно. В каждом случае предполагается, что соответствующая логическая операция выполняется над содержимым двух заданных регистров, а результат помещается в третий регистр, номер которого также указывается в команде. Например, команда `0x7ABC` может быть расшифрована следующим образом: "Выполнить операцию OR над содержимым регистров `0xB` и `0xC` и поместить результат в регистр `0xA`".

Операции сдвига

Операции сдвига и циклического сдвига (вращения) позволяют перемещать биты в регистре и часто используются для решения проблем выравнивания. Классификация этих операций производится по направлению движения (вправо или влево), а также с учетом того, является ли сдвиг циклическим. В рамках этой классификации существует множество различных вариантов, для обозначения которых используется смешанная терминология. Давайте бегло ознакомимся с основными принципами, положенными в ее основу.

Возьмем для примера некоторый байт и сдвинем его содержимое на один бит вправо. На правом конце байта крайний бит выйдет за его пределы и будет потерян, тогда как на левом конце образуется пустое место, в которое потребуется ввести некоторое значение. Что произойдет с удаляемым битом и что будет вставлено в освободившуюся позицию — именно это и определяет различия между множеством разнообразных операций сдвига. Одним из возможных решений является помещение бита, удаляемого с одного конца байта, в пустую позицию на другом его конце. В результате мы получим **циклический сдвиг**, который иногда называют вращением. Если выполнить циклический сдвиг байта вправо восемь раз подряд, то можно получить ту же битовую комбинацию, которая существовала вначале.

Другим вариантом решения является удаление бита, выходящего за пределы байта, и помещение в освободившиеся позиции исключительно значения `0`. Подобный вариант называют **логическим сдвигом**. Этот вариант сдвига влево можно использовать для умножения значения байта в дополнительном двоичном коде на число 2. В любом случае сдвиг двоичных цифр влево означает умножение значения на 2, подобно тому как и аналогичный сдвиг десятичных цифр означает умножение на десять. Кроме того, сдвинув двоичную строку вправо, можно выполнить деление ее значения на число 2. Однако в этом случае нужно обязательно сохранить тот знаковый бит, который используется в данной нотации. Для этого часто используется такой вариант сдвига вправо, при котором освободившаяся позиция (а это чаще всего и будет знаковый бит) всегда заполняется тем значением, которое в ней находилось до операции сдвига. Сдвиги, которые не изменяют значения знакового бита, иногда называют **арифметическими**.

Из всего разнообразия теоретически возможных команд сдвига в языке машины Vole (см. приложение B) присутствует только операция циклического сдвига вправо, которой присвоен код операции `0xA`. В данном случае первая шестнадцатеричная цифра в поле операндов определяет регистр, содержимое которого подвергается операции циклического сдвига, а оставшаяся часть поля операндов задает количество двоичных позиций, на которое осуществляется сдвиг. Следовательно, команду `0xA501` можно расшифровать следующим

образом: "Циклически сдвинуть содержимое регистра `0x5` вправо на `1` бит". Поэтому, если в регистре `0x5` исходно содержалось значение `0x65`, то после выполнения данной команды в нем будет содержаться значение `0xB2` (рис. 2.12). (Вы можете поэкспериментировать с тем, как прочие команды сдвига могут быть реализованы посредством определенных комбинаций команд машинного языка Vole. Например, поскольку регистры в машине Vole имеют длину 8 бит, циклический сдвиг вправо на 3 бита даст тот же самый результат, что и циклический сдвиг влево на 5 бит.)



Рис. 2.12. Циклический сдвиг (`Rotate`) комбинации битов `0x65` на один бит вправо

Арифметические операции

Несмотря на то что выше уже шла речь об арифметических операциях сложения, вычитания, умножения и деления, в этом вопросе все же необходимо расставить все точки над i. Во-первых, как уже говорилось, операция вычитания может быть выполнена с помощью сложения и отрицания. Более того, умножение — это всего лишь многократно повторенное сложение, а деление — многократно повторенное вычитание. (Шесть деленное на два равно трем, поскольку двойку из шестерки можно вычесть ровно три раза.) По этой причине некоторые малогабаритные ЦП имеют в своем наборе команд только операции сложения или, возможно, сложения и вычитания.

Кроме того, следует напомнить, что существует множество различных вариантов любой арифметической операции. Речь об этом уже шла выше в связи с операциями сложения, которые включены в набор команд машины Vole. При операциях сложения операнды могут быть представлены в двоичном

дополнительном коде, и тогда операция их сложения будет выполняться, как обычное поразрядное двоичное суммирование. Если же операнды будут представлены как числа в формате с плавающей точкой, то при суммировании сначала потребуется выделить мантиссу каждого из чисел, после чего эти значения нужно будет сдвинуть вправо или влево в зависимости от значения в поле порядка. Затем следует проверить знаковые биты и выполнить операцию сложения, а полученный результат вновь перевести в формат с плавающей точкой. Как видите, хотя обе описанные выше операции считаются операциями сложения, действия машины по их выполнению будут существенно различаться.

2.4. Вопросы и упражнения

1. Выполните приведенные ниже операции.

а.	01001011	**б.**	100000011	**в.**	11111111
AND	10101011	AND	11101100	AND	00101101
г.	01001011	**д.**	10000011	**е.**	11111111
OR	10101011	OR	11101100	OR	00101101
ж.	01001011	**з.**	100000011	**и.**	11111111
XOR	10101011	XOR	11101100	XOR	00101101

2. Предположим, что требуется выделить средние 4 бита в байте посредством помещения нулей в оставшиеся его четыре бита, не изменяя значения четырех средних битов. Какую маску и какую операцию следует использовать в этом случае?
3. Предположим, что нужно заменить значение средних 4 битов байта на обратное, не изменяя при этом значения оставшихся четырех битов. Какую маску и какую операцию следует использовать в этом случае?
4. **а.** Предположим, что была выполнена операция XOR с первыми двумя битами некоторой строки битов, а затем эта операция последовательно выполнялась с очередным результатом и следующим битом строки. Как общий результат связан с количеством единиц в исходной строке битов?

   б. Какое отношение приведенная выше задача имеет к определению требуемого значения бита четности при кодировании сообщения?
5. Иногда удобнее использовать логическую операцию вместо числовой. Например, логическая операция AND комбинирует значения двух битов по тому же принципу, что и операция умножения. Какая

логическая операция почти идентична операции сложения двух битов? Какие отличия существуют между этими операциями?

6. Какую логическую операцию и в сочетании с какой маской следует использовать для преобразования строчных букв в прописные в коде ASCII? А в случае преобразования прописных букв в строчные?
7. Каков будет результат при выполнении циклического сдвига вправо на три позиции для каждой из следующих битовых комбинаций?

   а. `01101010` **б.** `00001111` **в.** `01111111`
8. Каков будет результат при циклическом сдвиге влево на одну позицию для каждой из следующих битовых комбинаций, представленных в шестнадцатеричной системе счисления? Свой ответ предоставьте также в шестнадцатеричном виде.

   а. `0xAB` **б.** `0x5C` **в.** `0xB7` **г.** `0x35`
9. На сколько разрядов влево нужно циклически сдвинуть строку из 8 бит, чтобы результат был эквивалентен циклическому сдвигу этой же строки вправо на три бита?
10. Какая комбинация битов будет представлять сумму значений `01101010` и `11001100`, если полагать, что эти значения являются числами в дополнительном двоичном коде? А если считать, что это числа, представленные в формате с плавающей точкой, речь о котором шла в главе 1?
11. Используя язык машины Vole (см. приложение В), напишите программу, которая поместит единицу в старший бит ячейки памяти с адресом `0xA7`, оставив значения всех остальных битов этой ячейки без изменения.
12. Напишите для машины Vole программу, которая скопирует значения четырех средних битов ячейки памяти с адресом `0xE0` в младшие четыре бита ячейки памяти с адресом `0xE1`, при этом в старшие четыре бита этой ячейки должны быть занесены нули.

2.5. Взаимодействие с другими устройствами

Основная память и ЦП образуют центральное звено компьютера. В этом разделе мы рассмотрим, как это центральное звено, на которое мы здесь будем ссылаться как на "компьютер", взаимодействует с различными периферийными устройствами, такими как устройства массовой памяти, принтеры, клавиатура, мышь, экран дисплея, цифровая фотокамера и даже другие компьютеры.

Взаимодействие через управляющее устройство

Взаимодействие между машиной и другими устройствами обычно осуществляется через промежуточное устройство, называемое **контроллером**. Если в качестве примера взять персональный компьютер, то контроллер может представлять собой некоторую электронную схему, размещенную на материнской плате компьютера, либо быть выполнен в виде дополнительной платы (карты) расширения, которая вставляется в разъем на основной (материнской) плате компьютера при подключении к нему нового периферийного устройства. В любом случае контроллеры либо подключаются непосредственно к устройствам, установленным в корпусе компьютера, либо с помощью кабелей соединяются с внешними устройствами через промежуточные разъемы, установленные на задней стенке корпуса компьютера.

Задача контроллера состоит в преобразовании сообщений и данных, которыми обмениваются компьютер и периферийное устройство, в тот формат, который будет совместим с внутренними характеристиками самого компьютера и подключенного к нему устройства. Как правило, каждый контроллер обеспечивает взаимодействие с определенным видом устройств, поэтому прежде довольно часто вместе с новым периферийным устройством приходилось покупать и новый контроллер. Однако по мере того, как компьютеры получали все большее распространение, а количество доступных периферийных устройств неуклонно возрастало, стандарты связи постепенно совершенствовались с целью позволить одному контроллеру обеспечивать взаимодействие с устройствами многих типов. Последовательное появление все более и более быстрых версий стандартов **USB** (Universal Serial Bus — *универсальная последовательная шина*), **HDMI** (High Definition Multimedia Interface — *интерфейс мультимедиа высокого разрешения*) и **DisplayPort** (*порт дисплея*) является ярким примером неуклонного совершенствования стандартов, получивших в результате широкое распространение для подключения периферийных устройств к компьютерам. Так, единственный USB-контроллер сейчас можно использовать в качестве интерфейса между компьютером и широким набором всевозможных USB-устройств. (Теоретически к нему можно подключить до 127 устройств одновременно, или $2^7 - 1$.) Сегодня список представленных на рынке устройств, оснащенных USB-интерфейсом, включает мыши, принтеры, сканеры, внешние дисковые накопители, устройства флеш-памяти, цифровые фотокамеры, акустические системы, смарт-часы, смартфоны, планшеты и множество других, еще более экзотических устройств.

Взаимодействие каждого контроллера с компьютером обеспечивается посредством его подключения к той же шине, которая соединяет ЦП компьютера и его основную память (рис. 2.13). В результате контроллер получает

возможность непрерывно наблюдать за сигналами, посылаемыми из ЦП машины и основной памяти, а также отправлять по шине собственные сигналы.

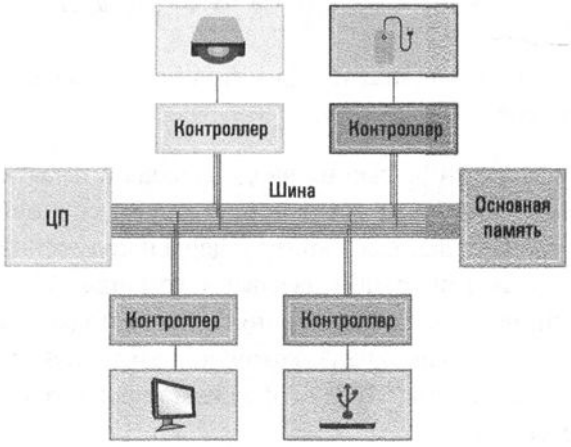


Рис. 2.13. Подключение контроллеров к шине компьютера

При таком подходе ЦП получает возможность взаимодействовать с подключенными к шине контроллерами точно таким же образом, как с основной памятью. Для отправки комбинации битов контроллеру эта комбинация сначала помещается в один из регистров общего назначения процессора, а затем ЦП выполняет команду, подобную команде `STORE`, которая позволяет "записать" эту комбинацию в контроллер. Аналогичным образом для получения комбинации битов от контроллера следует воспользоваться командой, подобной команде `LOAD`.

В некоторых типах компьютеров передача данных в контроллеры и из контроллеров осуществляется посредством тех же самых кодов операций `LOAD` и `STORE`, которые предусмотрены для взаимодействия с основной памятью. В подобных случаях каждый контроллер проектируется так, чтобы реагировать исключительно на ссылки на уникальный диапазон адресов памяти, тогда как основная память строится так, чтобы игнорировать ссылки на эти адреса. В результате, когда ЦП посылает сообщение на шину для записи данных в эти особые ячейки памяти, их получает соответствующий контроллер, а не основная память компьютера. Аналогичным образом, когда ЦП пытается считать данные из этих ячеек памяти, например с помощью команды `LOAD`, он получает битовую комбинацию не из основной памяти, а из соответствующего контроллера. Такая схема взаимодействия называется **отображением ввода-вывода в память**, поскольку различные устройства ввода-вывода компьютера представлены для ЦП как определенные ячейки памяти (рис. 2.14).

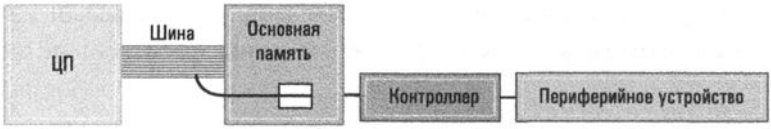


Рис. 2.14. Концептуальная схема метода отображения ввода-вывода в память

Альтернативой методу отображения ввода-вывода в память является такой подход, при котором в машинном языке присутствуют специальные коды операции для прямой передачи данных в контроллеры и получения данных от них. Команды с такими кодами операции называют командами ввода-вывода. Например, если в машинном языке Vole принять такой подход, то можно включить в него команду, подобную `0xF5A3`, которую в этом случае можно было бы расшифровать как: "Сохранить (STORE) содержимое регистра `0x5` в контроллере с идентификатором `0xA3`".

Прямой доступ к памяти

Поскольку контроллер подключен непосредственно к шине компьютера, он мог бы самостоятельно осуществлять взаимодействие с основной памятью на протяжении тех наносекунд, когда шина не используется центральным процессором. Подобный тип доступа контроллера к основной памяти называется **прямым доступом к памяти** (*Direct Memory Access* — **DMA**) и является важным средством повышения производительности компьютера. Например, чтобы считать с диска данные из определенного сектора, ЦП может отсылать его контроллеру представленные в виде битовых комбинаций запросы, требующие отыскать этот сектор, прочитать из него данные и поместить их в указанный блок ячеек основной памяти. Пока контроллер будет выполнять затребованную операцию считывания и записывать полученные данные в основную память с использованием механизма DMA, ЦП может продолжать обработку других заданий. А это означает, что в одно и то же время будут выполняться два разных действия: ЦП будет выполнять программу, а контроллер будет обеспечивать передачу данных между дисковым устройством и основной памятью компьютера. Такой подход позволяет избежать простоя вычислительных ресурсов во время выполнения относительно медленного процесса передачи данных.

Однако механизм DMA оказывает и определенный отрицательный эффект, поскольку при этом увеличивается количество взаимодействий, осуществляемых через шину компьютера. Битовые комбинации должны перемещаться между ЦП и основной памятью, между ЦП и каждым из контроллеров, а также между каждым из контроллеров и основной памятью компьютера. Координация всей этой деятельности, осуществляемой через шину компьютера, является

важнейшей задачей конструирования. Даже в самых лучших проектных решениях центральная шина может превратиться в источник помех в работе компьютера, возникающих из-за того, что ЦП и контроллеры соревнуются между собой за доступ к шине. Это затруднение известно как **узкое место архитектуры фон Неймана**, поскольку является следствием базовой архитектуры компьютеров, предложенной в свое время фон Нейманом, в которой ЦП извлекает свои инструкции из памяти по центральной шине.

Подтверждение установления связи

Передача данных между двумя компонентами компьютера редко бывает односторонним действием. На первый взгляд, может показаться, что принтер является устройством, которое способно только получать данные, однако в действительности он также может отсылать данные обратно в компьютер. В самом деле, компьютер способен подготавливать символы и передавать их принтеру намного быстрее, чем принтер может их печатать. Если компьютер будет передавать данные на принтер вслепую, то принтер быстро начнет отставать, что неизбежно приведет к потере данных. Поэтому такой процесс, как печать документа, предусматривает постоянный двусторонний диалог, известный как **подтверждение** (или квитирование), в процессе которого компьютер и периферийное устройство обмениваются информацией о текущем состоянии устройства и координируют свои действия.

Такой диалог часто предусматривает использование **слова состояния** устройства, т.е. определенной битовой комбинации, которая генерируется периферийным устройством и отсылается его контроллеру. Биты в слове состояния отражают текущее состояние устройства. Например, если устройство — принтер, то единичное значение младшего бита слова состояния может указывать на то, что в принтере закончилась бумага, тогда как следующий бит в этом слове будет определять, готов ли принтер к приему очередной порции информации. Еще какой-нибудь бит в слове состояния в единичном состоянии может сигнализировать о том, что в принтере замялась бумага. В зависимости от выбранной системы контроллер либо сам реагирует на подобную информацию о состоянии устройства, либо передает ее на обработку в ЦП. В любом случае использование слова состояния обеспечивает механизм, посредством которого поддерживается взаимодействие компьютера с периферийным устройством.

Основные типы соединений

Существуют два основных типа соединений между устройствами компьютера: параллельный и последовательный. Указанные термины определяют способ, который используется для передачи отдельных битов в комбинации по

отношению один к другому. При **параллельной передаче** несколько битов передаются одновременно, каждый по отдельной "линии связи". Такой способ позволяет повысить скорость передачи данных, однако требует относительно сложных соединений. В качестве примера таких соединений можно привести внутреннюю шину компьютера, в которой множество проводников или жил используется для того, чтобы обеспечить одновременную передачу больших блоков данных и других сигналов.

В противоположность этому при **последовательной передаче** в каждый момент по соединению передается только один бит данных. В результате соединение получается более простым в сравнении с параллельным и это является основной причиной его популярности. Примерами последовательных систем передачи данных являются стандарты USB и FireWire, обеспечивающие надежную и достаточно высокоскоростную передачу данных на небольшие расстояния, порядка нескольких метров. Для организации последовательной передачи данных на бо́льшие расстояния (в пределах квартиры, офиса или целого здания) чаще всего выбирают стандарт Ethernet (раздел 4.1) с использованием проводных или беспроводных соединений.

Для обеспечения связи между компьютерами на больши́е расстояния долгое время использовались преимущественно традиционные телефонные линии. Эти акустические каналы связи состояли из единственного провода, по которому звуковые сигналы передавались последовательно, один за другим, и компьютерные системы просто унаследовали от телефонных принцип последовательной передачи данных. Передача цифровых данных через подобные линии связи осуществляется посредством преобразования исходных битовых комбинаций в звуковые сигналы с помощью специальных устройств, называемых **модемами** (сокращение от *модулятор-демодулятор*). Далее полученные звуковые сигналы последовательно, один за другим, отсылаются по телефонным линиям, после чего на приемной стороне вновь преобразуются другим модемом в битовые комбинации.

Для организации более быстрой связи на дальние расстояния по традиционным телефонным линиям был предложен метод, получивший название **DSL** (*Digital Subscriber Line* — цифровая абонентская линия), в котором преимущества извлекались из того факта, что существующие телефонные линии в действительности способны работать с более широким диапазоном частот, чем тот, который используется при традиционном голосовом общении. Хотя метод DSL имел большой успех и получил очень широкое распространение, телефонные компании очень быстро обновили свои системы и перешли на использование оптоволоконных линий, обеспечивающих гораздо более эффективную реализацию цифровых линий связи в сравнении с обычными телефонными линиями.

Кабельные модемы — это еще один конкурирующий с прочими подход, при котором осуществляются модуляция и демодуляция последовательностей

битов для их передачи по кабельным телевизионным сетям. Многие провайдеры сетевых услуг сейчас широко используют эту технологию для одновременной передачи абонентам телевизионных сигналов с высоким разрешением и предоставления доступа к компьютерным сетям, используя для этой цели оптоволоконные либо обычные коаксиальные кабели.

Спутниковые линии связи с использованием высокочастотных радиоканалов обеспечивают доступ к компьютерным сетям даже в самых удаленных местах, где отсутствует доступ к любым скоростным телефонным линиям или сетям кабельного телевидения.

Скорость передачи данных

Скорость, с которой биты передаются от одного вычислительного компонента к другому, измеряется в **битах в секунду** (бит/с). Широкое распространение также получили такие единицы измерения, как **Кбит/с** (килобит в секунду, равный 1000 бит/с), **Мбит/с** (мегабит в секунду, равный миллиону бит/с) и **Гбит/с** (гигабит в секунду, равный миллиарду бит/с). Обратите внимание на использование для измерений в этом случае именно *битов*, а не байтов, т.е. 1 Кбайт в секунду в действительности эквивалентен 8 Кбит/с.

В каждом случае максимальная скорость передачи данных зависит от типа используемой линии связи и выбранной технологии. Для коротких расстояний чаще всего используются стандарты USB 3.0 и Thunderbolt, обеспечивающие скорость передачи данных на уровне многих гигабитов в секунду, чего вполне достаточно для большинства приложений мультимедиа. Именно по этой причине в сочетании с удобством использования и относительно невысокой стоимостью они получили широкое распространение в качестве средства для подключения к домашним компьютерам всевозможных периферийных устройств, таких как принтеры, внешние дисковые накопители, цифровые фотокамеры и т.д.

В сочетании с **мультиплексированием** (*уплотнением* канала, т.е. передачей с использованием кодирования или перемешивания данных одновременно нескольких логических информационных каналов по одному физическому каналу связи) и методами сжатия данных обычные телефонные линии позволяют достичь скорости передачи до 57,6 Кбит/с. Однако этого слишком мало для современных потребностей мультимедиа- и интернет-приложений, например для передачи потокового видео высокого разрешения с таких сайтов, как YouTube или Megogo. Для воспроизведения музыкальных записей в формате MP3 требуется скорость передачи данных не менее 64 Кбит/с, тогда как для воспроизведения видео даже самого низкого качества необходима скорость передачи данных уже на уровне единиц мегабитов в секунду. Именно по этой причине сначала DSL, а затем кабельные и спутниковые линии связи, обеспечивающие

скорость передачи на уровне многих мегабитов в секунду, пришли на смену традиционным акустическим телефонным линиям.

Максимальное значение скорости передачи данных в конкретном случае зависит от типа используемой линии связи и технологии, применяемой для организации передачи данных. Это максимальное значение часто приблизительно приравнивают к **пропускной способности** канала связи, хотя понятие "пропускная способность" подразумевает скорее емкость канала связи, а не его пропускную способность. Другими словами, когда говорят, что канал связи имеет высокую пропускную способность (или является широкополосным), то имеют в виду, что этот канал позволяет передавать биты с высокой скоростью, а также способен одновременно передавать большое количество информации.

2.5. Вопросы и упражнения

1. Предположим, что машина Vole использует механизм отображения ввода-вывода в память, а адрес `0xB5` определяет местоположение порта принтера, в который должны передаваться данные для вывода на печать.
   - **а.** Если регистр `0x7` содержит код ASCII буквы "A", какая команда машинного языка может быть использована для вывода этой буквы на печать?
   - **б.** Если наша машина способна выполнять миллион операций в секунду, то сколько раз за одну секунду этот символ может быть послан принтеру для вывода на печать?
   - **в.** Если принтер может напечатать пять стандартных страниц текста в минуту, то успеет ли он вывести на печать все символы, посланные ему при условиях, указанных в предыдущем пункте?
2. Предположим, что жесткий диск персонального компьютера вращается со скоростью 3000 оборотов в минуту. Каждая дорожка этого диска содержит 16 секторов, а каждый сектор — 1024 байта информации. Какая приблизительно скорость передачи данных потребуется для линии связи между дисководом и контроллером диска, если контроллер будет получать с дисковода биты непосредственно после их считывания по мере вращения диска?
3. Сколько времени займет передача романа, занимающего 300 страниц печатного текста, представленного символами в 16-битовых кодах Unicode, если передача данных будет осуществляться со скоростью 54 Мбит/с?

Многоядерные процессоры

Поскольку развитие технологий предоставляет возможность размещать все больше и больше электронных компонентов на одном кремниевом кристалле (чипе), физические расстояния между компонентами компьютера становятся все меньше и меньше. Например, единственный чип может содержать и ЦП, и основную память. Это пример подхода SoC (*System-on-a-Chip* — система на кристалле), при котором основной целью является обеспечить полный набор необходимых аппаратных средств в единственном кристалле, который затем можно будет использовать как абстрактный инструмент при проектировании систем более высокого уровня. В других случаях на один кристалл помещаются многие копии одной и той же схемы. Этот исторически более поздний подход изначально появился в форме кристаллов, содержащих несколько независимых копий вентиля или, возможно, триггера. Сегодняшний уровень этого искусства позволяет разместить на одном кристалле сразу несколько ЦП. Такая архитектура положена в основу создания чипов, получивших название "многоядерные процессоры". Эти устройства могут включать два и более центральных процессоров, размещенных на одном кристалле, вместе с разделяемой ими кеш-памятью. Такие чипы упрощают конструирование MIMD-систем и в наше время получили самое широкое распространение в домашних компьютерах, ноутбуках, планшетах и смартфонах.

2.6. Манипулирование данными в программе

Одной из важнейших функций компьютерных языков программирования, таких как Python, является ограждение тех, кто их использует, от необходимости утомительного описания всех деталей работы с компонентами машины на ее нижнем уровне. Вы только что завершили изучение бо́льшей части этой главы, где речь шла как раз о самых низких уровнях обработки данных в процессорах компьютеров. Поэтому будет очень поучительно познакомиться с важнейшими деталями написания сценариев на языке Python, которые избавляют программиста от необходимости беспокоиться обо всем этом.

Как будет подробно рассказано в главе 6, операторы языков программирования высокого уровня для своего выполнения должны быть отображены в соответствующий набор машинных команд самого низкого уровня. При этом единственный оператор языка Python может отображаться в единственную машинную команду, а может и в десятки, а то и в сотни машинных команд — в зависимости от сложности оператора и эффективности команд машинного языка. Различные версии интерпретатора языка Python, в сочетании с определенными элементами программного обеспечения операционной системы компьютера, обеспечивают практическую реализацию процесса подобного отображения

для каждого конкретного типа процессоров, используемых в компьютерах. По этой причине программистам на языке Python нет необходимости знать, на процессоре какого именно типа, RISC или CISC, будет выполняться их сценарий.



Основные положения для запоминания

- **Операторы языков программирования обычно должны быть предварительно переведены (оттранслированы) в команды другого языка (более низкого уровня) для выполнения на компьютере.**

Можно привести много примеров операторов языка Python, которые достаточно точно соответствуют основным машинным командам современных компьютеров или даже простейшей машины Vole, подробно описанной в приложении B. Кроме того, представление в языке Python целых чисел и чисел с плавающей запятой очень напоминает представление операндов команд `ADD` в нашей простой машине. Присвоение значений переменным, безусловно, предполагает использование команд `LOAD`, `STORE` и `MOVE` по той же самой схеме. Язык Python снимает с нас необходимость беспокоиться о том, какие регистры процессора будут использоваться, и берет на себя выбор тех машинных команд, которые будут применены для выполнения наших инструкций. Мы не сможем увидеть, что находится в регистре команд или счетчике адреса и узнать адреса используемых ячеек памяти, тем не менее сценарии на языке Python всегда выполняются последовательно, оператор за оператором, точно таким же образом, как программы на машинном языке.



Основные положения для запоминания

- **Операторы программ выполняются последовательно.**

Логические операции и операции сдвига

Логические операции и операции сдвига могут выполняться с любыми типами числовых данных, но поскольку обычно они оперируют отдельными битами данных, проще всего иллюстрировать работу этих операторов на данных в двоичном представлении. В языке Python, где префикс '`0x`' используется для

определения шестнадцатеричных значений, префикс '0b' аналогичным образом можно использовать для определения двоичных значений[1].

```
x    = 0b00110011
mask = 0b00001111
```

Обратите внимание, что в действительности не будет никакой разницы в результате при присвоении переменной x десятичного значения 51 (что выглядит как 110011 в двоичном представлении) или 0x33 (это то же число 51, но представленное в шестнадцатеричной нотации) или при присвоении переменной mask десятичного значения 15 (что есть 1111 в двоичной нотации) или 0x0F (т.е 15 в шестнадцатеричном представлении). Нотация, которую мы используем для представления целого числа в операторе присвоения языка Python, не оказывает никакого влияния на то, как это число будет храниться в компьютере; она определяет лишь то, как люди, читающие программу, будут себе его представлять.

Ниже приведены встроенные операторы языка Python, предназначенные для выполнения каждой из побитовых логических операций, обсуждавшихся в разделе 2.4.

```
print(0b00000101 ^ 0b00000100)  # Печатать 5 XOR 4, что равно 1
print(0b00000101 | 0b00000100)  # Печатать 5 OR  4, что равно 5
print(0b00000101 & 0b00000100)  # Печатать 5 AND 4, что равно 4
```

Теперь у нас появилась возможность повторить на языке Python все примеры, приведенные выше, в разделе 2.4.

```
                                #       10011010
print(0b10011010 & 0b11001001)  #   AND 11001001
                                #       10001000
                                #       10011010
print(0b10011010 | 0b11001001)  #   OR  11001001
                                #       11011011
                                #       10011010
print(0b10011010 ^ 0b11001001)  #   XOR 11001001
                                #       01010011
```

Во всех приведенных выше случаях интерпретатор языка Python выведет результаты в формате, принимаемом для выходных данных по умолчанию, т.е. в десятичном представлении. Если вы хотите, чтобы результат был выведен в двоичном представлении, используйте встроенную функцию, предназначенную для преобразования целых чисел в строку из нулей и единиц, где каждый символ соответствует одному разряду в двоичном представлении числа.

[1] Этот синтаксис является одним из недавних дополнений, принимаемых по мере развития языка Python. Чтобы повторить приведенные здесь примеры, убедитесь, что вы используете версию языка не ниже Python 3.

```
print(bin(0b10011010 & 0b11001001))  # Печатает "0b10001000"
print(bin(0b10011010 | 0b11001001))  # Печатает "0b11011011"
print(bin(0b10011010 ^ 0b11001001))  # Печатает "0b1010011"
```

Поскольку новые версии языка Python могут использовать произвольное количество цифр для представления чисел, ведущие нули на печать не выводятся. Следовательно, в третьей строке вывода выше будет выведено только семь цифр, а не восемь.

Встроенные операторы языка Python для выполнения операций логического сдвига представляют собой сдвоенные символы "больше чем" и "меньше чем", что визуально указывает на направление сдвига. При этом операнд справа от этих символов задает количество двоичных позиций (битов), на которое осуществляется сдвиг.

```
print(0b00111100 >> 2)     # Печатает  "15", что есть 0b00001111
print(0b00111100 << 2)     # Печатает "240", что есть 0b11110000
```

Помимо сдвига последовательности битов (маски) влево или вправо, операторы побитового сдвига могут использоваться для умножения (сдвиг влево) или деления (сдвиг вправо) целых чисел на соответствующие степени числа 2.

Управляющие конструкции

В машинном языке, обсуждавшемся выше в этой главе, группа команд управления представляла собой инструмент для перехода от одной части программы к другой. В языках программирования высокого уровня, подобных Python, такие возможности обеспечиваются тем, что принято называть **управляющими конструкциями** — синтаксическими шаблонами, позволяющими выражать алгоритмы более лаконично. Одним из примеров таких конструкций является оператор `if`, позволяющий при выполнении программы пропустить сегмент кода, если значение заданной в сценарии булевой величины не является истиной.

```
if (water_temp > 60):
  print('Вода в ванне слишком горячая!')
```

Интуитивно понятно, что этот фрагмент кода на языке Python будет отображен в последовательность машинных команд, выполняющих сравнение значения в переменной `water_temp` с целым значением `60`, оба из которых были предварительно загружены в регистры. Команда условного перехода обеспечит пропуск последовательности машинных команд, необходимых для выполнения встроенного оператора `print()`, если значение переменной `water_temp` будет больше `60`.

Другая управляющая конструкция представляет собой цикл `while`, который позволяет выполнять сегмент кода многократно, пока выполняется некоторое условие.

```
while (n < 10):
  print(n)
  n = n + 1
```

Полагая, что переменная `n` имеет исходное значение, меньшее `10`, этот цикл будет продолжать выводить значение переменной `n`, а затем увеличивать ее значение на единицу до тех пор, пока оно не станет равным или больше `10`.

Более подробно мы будем рассматривать эти и другие управляющие конструкции в главе 5 и последующих главах. А на данный момент мы сосредоточим внимание на механизме, который позволит нам передать управление в другую часть программы, выполнить требуемые действия, а затем вернуться в ту точку программы, откуда был выполнен переход.

Функции

Ранее мы уже познакомились с тремя встроенными операторами языка Python, синтаксис которых отличается от используемого в арифметических и логических операторах. Обращение к операторам `print()`, `str()` и `bin()` осуществляется указанием присвоенных им имен, а не знаков операции, и предусматривает использование скобок, в которые помещаются их операнды.

Все эти конструкции являются примерами особых средств языка Python, называемых **функциями**. В математике термин "функция" обычно используется для описания алгебраических взаимосвязей, как, например, в выражении "$f(x) = x^2 + 3x + 4$". Увидев такое определение функции, мы понимаем, что в последующих строках выражение "$f(5)$" будет означать, что значение 5 должно быть подставлено во всех тех местах, где в определении функции $f()$ встречается параметр x. Следовательно, $f(5) = 5^2 + 3*5 + 4 = 25 + 15 + 4 = 44$. Эта абстракция позволяет использовать выражение множество раз без необходимости дублировать его при каждом упоминании. В языках программирования функции представляют собой очень схожие конструкции, позволяющие нам присвоить имя некоторой последовательности операторов, которые должны выполняться с учетом переданного к началу ее выполнения значения заданного параметра или параметров. Исходя из того способа, которым эти языковые конструкции отображаются в команды машинного языка низшего уровня, появление имени функции в выражении или операторе принято называть **вызовом функции**. В языках программирования механизм вызова функции является важным сокращением, позволяющим существенно уменьшить сложность кода программы.



Основные положения для запоминания

- **Функция — это именованная группа операторов языка программирования.**
- **Функции являются многократно используемыми абстракциями на уровне программы.**
- **Использование функций позволяет уменьшить сложность написания и сопровождения программ.**

Выражения `print()` и `bin()` в приведенных выше примерах являют собой примеры вызова функций, которые указывают интерпретатору языка Python на необходимость выполнить последовательность операций, сохраненную под указанным именем функции, а затем вернуться в точку вызова и продолжить работу, воспользовавшись предоставленными функцией результатами ее выполнения. Синтаксис вызова функции предусматривает указание ее имени, непосредственно за которым следует открывающая скобка, а за ней — **значения аргументов** функции, за которыми должна следовать закрывающая скобка. Значения аргументов функции будут переданы ей в качестве значений параметров, указанных в определении функции. В этой записи очень важно соблюдать четкое сопоставление каждой открывающей скобке соответствующей закрывающей скобки; если не придерживаться этого правила, интерпретатор Python укажет на синтаксическую ошибку, что является характерным промахом для начинающих. При каждом вызове функции можно указывать различные значения аргументов, передаваемые ей в качестве параметров, и этот механизм обеспечивает возможность многократного использования программного кода функции с различными значениями ее параметров без необходимости дублирования этого кода в разных местах программы.



Основные положения для запоминания

- **Параметры обеспечивают возможность передачи различных значений в качестве входных данных для функции при каждом ее вызове из программы.**
- **Параметры позволяют обобщить метод решения задачи, что делает возможным вместо дублирования кода использовать в программе функции.**

С этого момента мы будем строго следовать соглашению об указании скобок при каждом упоминании имени функции языка Python, как в данном случае: `print()`. Это позволит четко отличать названия функций от имен переменных и других элементов программы.

Функции могут иметь различный вид, помимо того, с которым вы уже познакомились. Так, некоторые функции могут принимать более одного аргумента, например функция `max()`:

```
x = 1034
y = 1056
z = 2078
biggest = max(x, y, z)
print(biggest)              # Печатает "2078"
```

При наличии у функции нескольких аргументов все они разделяются запятыми и заключаются в одну пару скобок. Некоторые функции **возвращают** значение, а это означает, что вызов функции сам по себе может выступать как часть более сложного выражения либо являться правой частью оператора присваивания. Примерами функций, которые возвращают значение, являются упоминавшиеся выше `max()` (использовалась в приведенном выше примере) и `bin()` (принимает целое число и возвращает соответствующую строку нулей и единиц). Другие функции не возвращают никакого значения и обычно используются как самостоятельные операторы; примером такой функции является `print()`. Функции, не возвращающие значения, иногда называют **void-функциями** или **процедурами**, хотя в языке Python не делается различий в их синтаксисе. Не имеет смысла присваивать результат выполнения void-функции какой-либо переменной, как в следующем выражении, поскольку она ничего не возвращает (*void* — пустой, несуществующий).

```
x = print('hello world!')  # переменной x присвоено значение None
```

Тем не менее такая запись в языке Python не является ошибкой, а стало быть, здесь есть тонкое отличие от ситуации, когда переменной `x` вообще не присваивается никакого значения.



Основные положения для запоминания

- **Процедуры имеют имена и могут иметь параметры и возвращаемые значения.**

Каждая из функций, с которыми вы познакомились на текущий момент, является одной из нескольких десятков встроенных функций собственно языка Python, но помимо этого, существуют обширные библиотеки с дополнительными наборами функций, которые можно использовать при написании более сложных сценариев. Модуль некоторой библиотеки в языке Python может содержать множество полезных функций, в применении которых обычно нет надобности, но которые, безусловно, можно будет вызвать при появлении такой необходимости. Вот как это делается.

```
# Вычисление длины гипотенузы прямоугольного треугольника
import math

sideA = 3.0
sideB = 4.0

# Вычисление длины третьей стороны на основании теоремы Пифагора
hypotenuse = math.sqrt(sideA**2 + sideB**2)
print(hypotenuse)
```

В этом примере оператор `import` предупреждает интерпретатор языка Python о том, что в сценарии имеются ссылки на библиотеку с именем "`math`", которая является одной из библиотек в стандартном наборе библиотечных модулей, поставляемых вместе со средой разработки языка Python. Функция `sqrt()`, определение которой дано в модуле библиотеки `math`, возвращает квадратный корень числа, переданного ей в качестве аргумента при вызове. В нашем случае этот аргумент был задан как выражение, вычисляющее сумму квадратов значений переменных `sideA` и `sideB`. Обратите внимание, что вызов библиотечной функции должен включать как имя модуля библиотеки ("`math`"), так и имя самой функции ("`sqrt`"), разделяемых точкой.

Модуль `math` языка Python включает десятки полезных математических функций, в том числе логарифмические, тригонометрические и гиперболические, а также определения некоторых важнейших констант, таких как `math.pi`.

Помимо встроенных модулей библиотек, язык Python предоставляет необходимый синтаксис для написания сценариев, в которых можно определить собственные функции. Мы рассмотрим несколько простых примеров определения функции в конце этого раздела и познакомимся с более сложными вариантами функций в последующих главах.

Ввод-вывод

В предыдущих примерах фрагментов программного кода и сценариев для вывода результатов мы использовали встроенную функцию языка Python `print()`. Многие языки программирования включают схожие механизмы для организации ввода и вывода, предоставляя программистам удобные абстракции для пересылки данных в либо из процессора компьютера. В действительности эти встроенные процедуры ввода-вывода напрямую взаимодействуют с контроллерами и периферийными устройствами, разговор о которых шел в предыдущем разделе.

До этого момента ни один из наших примеров сценариев еще не требовал организации ввода данных от пользователя. Самый простой способ получить в

сценарии данные от пользователя состоит в использовании встроенной функции языка Python input().

```
echo = input('Пожалуйста, введите строку для вывода: ')
print(echo * 3)
```

Функция input() принимает необязательный аргумент, представляющий собой строку приглашения ко вводу для ожидающего пользователя. При выполнении приведенный выше сценарий выведет текстовое сообщение "Пожалуйста, введите строку для вывода:" и перейдет в режим ожидания, пока пользователь введет какие-либо данные. Когда он нажмет клавишу <Enter>, сценарий присвоит введенную пользователем строку символов (без символа <Enter>) в качестве значения строковой переменной echo. Второй оператор сценария три раза выведет полученную строку на экран. (Вспомните, что оператор '*' используется для повторения строкового значения.)

Получив возможность запрашивать у пользователя входные данные, давайте перепишем наш сценарий вычисления длины гипотенузы с целью получения длины двух сторон от пользователя вместо того, чтобы жестко кодировать эти значения в операторах присваивания.

```
# Вычисление длины гипотенузы прямоугольного треугольника
import math

# Ввод длин двух сторон треугольника, первая попытка
sideA = input('Длина стороны A? ')
sideB = input('Длина стороны B? ')
# Вычисление длины гипотенузы по теореме Пифагора
hypotenuse = math.sqrt(sideA**2 + sideB**2)

print(hypotenuse)
```

При запуске на выполнение этот сценарий выведет пользователю сообщение "Длина стороны A?" и перейдет в режим ожидания завершения ввода. Предположим, что пользователь ввел "3" и нажал клавишу <Enter>. Сценарий выведет сообщение "Длина стороны B?" и вновь перейдет к ожиданию завершения ввода. Предположим, что в этот раз пользователь ввел "4" и нажал клавишу <Enter>. И здесь интерпретатор языка Python неожиданно прервет выполнение сценария и выведет сообщение:

```
hypotenuse = math.sqrt(sideA**2 + sideB**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Подобное сообщение об ошибке очень легко получить в языке с динамическим определением типа данных, подобном языку Python. Дело в том, что наша процедура вычисления длины гипотенузы, которая прекрасно работала в

предыдущей версии сценария, уже не может работать в том же виде и выдает сообщение об ошибке, когда используемые в ней значения вводятся пользователем. Возникшая проблема характеризуется словами 'TypeError', указывающими на тот факт, что интерпретатор языка Python не знает, как можно возвести в квадрат текущее значение переменной sideA, поскольку это значение в данной версии сценария представляет собой строку символов, а не целое число, как это было в предыдущей версии сценария. Как видите, проблема вызвана не тем оператором, в котором вычисляется длина гипотенузы, а предыдущими операторами, когда значения переменных sideA и sideB были получены от функции input(). Это также типичная ошибка, с которой можно столкнуться при обучении любым языкам программирования. Далее в сообщении интерпретатор Python предпринимает попытку указать строку сценария, вызвавшую появление проблемы, но реальная причина ошибки на самом деле имеет место в сценарии выше этой строки.

В фрагменте кода с троекратным выводом введенной строки было очевидно, что значение, присваиваемое переменной echo, должно быть строкой символов, введенных пользователем. Точно таким же образом функция input() ведет себя и в сценарии вычисления длины гипотенузы, даже несмотря на то, что программист в данном случае простодушно предполагал ввод числового значения. Двоичное представление символа "4" в коде ASCII или UTF-8 отличается от представления целого числа 4 в дополнительном двоичном коде, поэтому наш сценарий на языке Python должен предусматривать явное преобразование введенного значения из одного представления в другое, прежде чем можно будет приступить к вычислениям, выполняемым с целочисленными значениями.

К счастью, другая встроенная функция языка Python предоставляет нам эту возможность, — функция int(), которая при вызове предпринимает попытку преобразовать переданный ей аргумент в целочисленное представление. Если это ей не удается, выводится соответствующее сообщение об ошибке.

В нашем сценарии имеется по крайней мере три места, где можно воспользоваться функцией int() для устранения возникшей в нем ошибки. Мы можем вызвать ее еще до присвоения введенных функцией input() значений переменным. Мы можем добавить в сценарий новые строки специально для выполнения требуемого преобразования значений с помощью этой функции. Наконец, мы можем выполнить преобразование непосредственно перед возведением значения в квадрат в выражении, указанном при вызове функции math.sqrt(). Измененный сценарий будет выглядеть так, как показано ниже, — здесь используется первый вариант из трех, рассмотренных выше. Остальные два я оставляю в качестве упражнения для читателя.

```
# Вычисление длины гипотенузы прямоугольного треугольника
import math
```

```
# Ввод длин двух сторон треугольника с преобразованием в целое
sideA = int(input('Длина стороны A? '))
sideB = int(input('Длина стороны B? '))
# Вычисление длины гипотенузы по теореме Пифагора
hypotenuse = math.sqrt(sideA**2 + sideB**2)

print(hypotenuse)
```

Измененный вариант сценария работает так, как ему положено, и может использоваться для расчета длины гипотенузы множества прямоугольных треугольников без необходимости внесения в него каких-либо изменений, как это требовалось для исходной версии сценария, в которой ввод данных не использовался.

И как последний штрих сообщу, что функция `int()` выполняет преобразование, тщательно анализируя переданный ей строковый аргумент и интерпретируя его как число. Если исходная строка будет представлять собой число, но не целое, например "3.14", функция `int()` просто отбросит дробную часть и вернет только целую часть исходного значения. Такое поведение является операцией *усечения*, а не округления, как мог бы ожидать непосвященный пользователь.

Сходные функции преобразования есть в языке Python для всех прочих стандартных типов данных.

Программа Marathon Training Assistant

Приведенный ниже полный сценарий на языке Python призван продемонстрировать многие концепции, обсуждавшиеся в этом разделе. Поскольку популярность тренировочных пробежек на различные расстояния постоянно возрастает, многие из энтузиастов этого метода оздоровления разрабатывают для себя сложные графики тренировок с целью подготовки к суровым испытаниям, неизбежным при беге на марафонские дистанции. Этот сценарий призван помочь тем бегунам, кто хочет рассчитать, как долго будет проходить его тренировка, исходя из намеченной дистанции и темпа бега. При заданных темпе (количество минут и секунд, за которые пробегается одна миля) и общей дистанции пробежки в этом сценарии вычисляется прогнозируемое количество времени, необходимое для тренировки, а также предлагается удобный способ вычисления скорости в милях в час. На рис. 2.15 приведен пример конкретных данных, в котором в каждой строке в первых трех столбцах представлены входные данные, а в последних трех столбцах выведены предполагаемые результаты. Обратите внимание, что различные реализации интерпретаторов языка Python могут выводить иное количество десятичных знаков после точки, не совпадающее с тем, которое присутствует в значениях скорости на рис. 2.15, не округленных до целых чисел.

Время на 1 милю				Общее время пробежки	
Минуты	Секунды	Мили	Скорость (миль/ч)	Минуты	Секунды
9	14	5	6.49819494584	46	10
8	0	3	7.5	24	0
7	45	6	7.74193548387	46	30
7	25	1	8.08988764044	7	25

Рис. 2.15. Пример данных о тренировках по бегу

```
# Сценарий Marathon training assistant.
import math

# Эта функция преобразует количество минут и секунд просто в секунды.
def total_seconds(min, sec):
  return min * 60 + sec

# Эта функция вычисляет скорость в милях в час при заданном
# времени (в секундах) за которое будет пробегаться одна миля.
def speed(time):
  return 3600 / time

# Предложение пользователю ввести темп бега и дистанцию.
pace_minutes = int(input(Минуты на 1 милю? '))
pace_seconds = int(input('Секунды на 1 милю? '))
miles = int(input('Всего миль? '))

# Вычисление и вывод расчетной скорости.
mph = speed(total_seconds(pace_minutes, pace_seconds))
print('Ваша скорость бега будет')
print(mph)

# Вычисление общего предполагаемого времени пробежки.
total = miles * total_seconds(pace_minutes, pace_seconds)
elapsed_minutes = total // 60
elapsed_seconds = total % 60

print('Общее время пробежки составит')
print(elapsed_minutes)
print(elapsed_seconds)
```

В приведенном выше сценарии используются как встроенные функции языка Python (`input()`, `int()` и `print()`), так и функции, определенные пользователем

(speed() и total_seconds()). Определение пользовательской функции начинается с ключевого слова def, за которым следуют имя определяемой функции и список параметров, которые должны быть ей переданы при вызове на выполнение. Следующая строка с отступом представляет собой тело функции и выражает те действия, которые функция будет выполнять. В последующих главах вы познакомитесь с примерами функций, тело которых будет включать более одного оператора. Ключевое слово return отмечает выражение, которое должно быть вычислено для получения результата выполнения данной функции.

Код определяемых пользователем функций всегда находится в начале сценария, но управление им будет передано только тогда, когда при выполнении сценария будет достигнута строка, в которой вызов данной функции является частью некоторого выражения. Также обратите внимание на то, как в этом сценарии вызовы функции вложены один в другой. Результат выполнения функции input() немедленно передается в качестве аргумента функции int(), а результат выполнения функции int(), в свою очередь, присваивается переменной. Аналогичным образом результат выполнения функции total_seconds() немедленно передается как аргумент функции speed(), результат выполнения которой присваивается переменной mph. В каждом из этих случаев было бы вполне допустимо в отдельном операторе вызвать первую функцию и присвоить результат ее выполнения некоторой дополнительной переменной, а затем в следующем операторе вызвать следующую функцию, используя в качестве аргумента значение этой дополнительной переменной. Однако выше вам была предложена именно компактная форма как более краткая и не требующая использования нескольких дополнительных переменных для хранения промежуточных результатов вычислений.

Если в начале работы сценария ввести значения 7 минут и 45 секунд для темпа и 6 миль в качестве дистанции, то сценарий выведет следующие результаты расчетов:

```
Ваша скорость бега будет
7.74193548387
Общее время пробежки составит
46
30
```

Формат представления выходных данных в этом сценарии остается достаточно примитивным. Здесь не достает указания используемых единиц измерения (7.74193548387 миль/час и 46 минут, 30 секунд), на выходе представлено количество знаков после точки, не имеющее смысла для столь простых вычислений, а количество выводимых строк слишком велико, — было бы вполне достаточно всего двух. Доработку сценария с учетом этих замечаний мы оставляем в качестве упражнения для читателей.

2.6. Вопросы и упражнения

1. В примере сценария расчета длины гипотенузы задаваемый размер длин сторон ограничен целыми числами, но на выходе длина гипотенузы представлена числом с плавающей точкой. Почему? Доработайте сценарий так, чтобы он выводил целое значение.
2. Доработайте сценарий расчета длины гипотенузы так, чтобы на входе использовались значения с плавающей точкой, без усечения вводимых значений. Какой вариант вы считаете более подходящим — целочисленную версию из предыдущего вопроса или версию, использующую числа с плавающей точкой?
3. Встроенная функция языка Python `str()` предназначена для конвертирования числового значения ее аргумента в соответствующую строку символов, а символ '+', как вы помните, можно использовать для конкатенации строковых значений в единую строку. Используйте эти средства для модификации сценария Marathon training assistant таким образом, чтобы его выходные данные имели более удобочитаемый вид, например:

   ```
   Ваша скорость бега будет 7.74193548387 миль/ч
   Общее время пробежки составит 46 мин, 30 с
   ```

4. Воспользуйтесь встроенной функцией языка Python `bin()` для написания сценария, который будет вводить целые десятичные числа и выводить их двоичное представление в виде последовательности единиц и нулей.
5. Операция `XOR` часто используется как для эффективного вычисления контрольных сумм (см. раздел 1.9), так и для шифрования (см. раздел 4.5). Напишите на языке Python простой сценарий, который будет вводить числа и выводить результат применения к ним операции `XOR` с использованием маски в виде комбинации единиц и нулей, например такой, как `0x55555555`. Таким образом, сценарий будет "зашифровывать" вводимые числа, представляя их на выходе в виде чисел, казалось бы, совершенно не связанных с исходными. Однако при повторном выполнении и вводе в качестве входных данных зашифрованных значений на выходе будут получены исходные числа.
6. Проведите исследования в отношении возможных ошибочных ситуаций, которые могут возникнуть при вводе значений, не соответствующих ожиданиям сценария. Например, что произойдет, если вы введете все нули в качестве входных данных для сценария вычисления длины гипотенузы или для сценария Marathon training assistant? А что произойдет, если вводить отрицательные числа или строку произвольных символов вместо чисел?

2.7. Другие типы архитектуры компьютеров

Для расширения кругозора целесообразно будет рассмотреть некоторые варианты архитектуры построения компьютеров, отличающиеся от той архитектуры, речь о которой шла в предыдущих разделах этой главы.

Конвейерная обработка

Скорость прохождения электронных импульсов по проводам не превышает скорости света. Поскольку скорость света составляет около 30 см/нс (одна миллиардная часть секунды), потребуется не менее двух наносекунд, чтобы блок управления центрального процессора выбрал команду из ячейки памяти, которая находится от него на расстоянии около 30 см. (Запрос на считывание должен поступить в схемы основной памяти, для чего потребуется не менее одной наносекунды. После этого выбранная команда должна быть доставлена в блок управления, что также потребует не менее одной наносекунды.) Следовательно, чтобы выбрать и выполнить команду, машине потребуется несколько наносекунд, а это означает, что увеличение скорости выполнения команд прямо связано с проблемой его миниатюризации.

Однако увеличение скорости выполнения программы — это не единственный способ повысить производительность компьютеров. Истинной целью в этом случае является повышение их **пропускной способности**. Этот термин означает общее количество работы, которое машина способна выполнить за определенный период времени.

Приведем пример того, как можно повысить пропускную способность компьютера без увеличения скорости выполнения команд, используя подход, называемый **конвейерной обработкой**, — прием, когда выполнение этапов машинного цикла может перекрываться во времени. Например, во время этапа выполнения одной из команд для следующей команды уже может выполняться этап выборки, а это означает, что выполнение более одной команды одновременно осуществляется по принципу конвейера, т.е. каждая из них будет находиться на разной стадии выполнения. В результате общая пропускная способность компьютера увеличится, причем без повышения скорости выборки и выполнения каждой отдельной команды. (Естественно, когда машина достигнет команды перехода (JUMP), все преимущества от предварительной выборки и выполнения последующих команд будут утрачены, так как в действительности потребуется выполнение совершенно других команд, которых в данное время на "конвейере" нет.)

Конструкции современных процессоров оставляют далеко позади рассмотренный выше простейший пример конвейерной обработки. Сейчас они часто

способны выбирать сразу несколько команд за одно и то же время, а также реально выполнять больше одной команды одновременно, если только их действия не являются взаимозависимыми.

Многопроцессорные машины

Использование конвейерного режима можно рассматривать как первый шаг в направлении реализации **параллельной обработки**, предусматривающей одновременное выполнение сразу нескольких действий. Однако параллельная обработка требует использования нескольких устройств обработки данных, что приводит к необходимости создания многопроцессорных или **многоядерных** машин.

По этому принципу разработано подавляющее большинство современных компьютеров. Один подход предусматривает подключение к одним и тем же ячейкам основной памяти нескольких устройств обработки данных, каждое из которых напоминает обычный центральный процессор однопроцессорной машины. В такой конфигурации процессоры могут работать независимо, координируя свои действия посредством обмена сообщениями через общие ячейки памяти. Например, когда один процессор получает большое и сложное задание, он может записать программу для выполнения части этого задания в общем поле памяти, а затем отослать другому процессору запрос на ее выполнение. В результате мы получим машину, в которой разные последовательности команд выполняют обработку разных наборов данных. Подобная архитектура носит название **MIMD** (*multiple instruction stream, multiple-data stream* — множество потоков команд с множеством потоков данных). Очевидно, что она является противоположной по отношению к традиционной архитектуре компьютеров, называемой **SISD** (*single instruction stream, single-data stream* — один поток команд и один поток данных).

Еще одним вариантом архитектуры многопроцессорных компьютеров является такое соединение процессоров между собой, которое позволяет им одновременно выполнять одну и ту же последовательность команд, но с разными наборами данных. Этот вариант носит название архитектуры **SIMD** (*single instruction stream, multiple-data stream* — один поток команд и множество потоков данных). Машины этого типа лучше всего подходят для выполнения таких приложений, в которых один и тот же алгоритм обработки применяется к отдельным наборам схожих элементов, составляющих один большой блок данных. Еще один подход к реализации параллельной обработки заключается в конструировании больших машин как некоего конгломерата из машин меньшего размера, каждая из которых имеет собственную память и центральный

процессор. В подобной архитектуре каждая малая машина связана со своими соседями; в результате задача, поставленная перед всей системой, может быть разделена на элементарные задания, распределяемые между отдельными машинами. Таким образом, если задача, поставленная перед одной внутренней машиной, может быть разделена на несколько подзадач, то эта машина может "попросить" соседние машины выполнить все эти подзадачи параллельно. В результате вся задача в целом может быть выполнена в многопроцессорной машине намного быстрее, чем в однопроцессорной.

2.7. Вопросы и упражнения

1. Еще раз вернемся к вопросу 3 из раздела 2.3. Если в машине будет применяться обсуждавшаяся выше технология конвейерной обработки, то какая команда попадет на "конвейер", когда будет выполняться команда, расположенная в ячейке с адресом `0xAA`? При каких условиях конвейерная обработка на данном этапе программы не будет давать никаких преимуществ?
2. Какие противоречия должны быть разрешены при запуске программы, приведенной в вопросе 4 из раздела 2.3, в машине с конвейерной обработкой данных?
3. Предположим, что два "центральных" процессора подсоединены к одному и тому же полю основной памяти и выполняют различные программы. Более того, в одно и то же время одному из процессоров требуется прибавить единицу к содержимому некоторой ячейки памяти, а другому необходимо отнять единицу от содержимого этой же ячейки. (Общий результат этих действий должен быть таким, чтобы содержимое данной ячейки осталось неизменным.)
   а. Опишите последовательность действий, которая приведет к тому, что в результате их выполнения содержимое данной ячейки будет на единицу меньше исходного значения.
   б. Опишите, какая последовательность действий приведет к тому, что в результате их выполнения содержимое данной ячейки будет на единицу больше исходного значения.

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. **а.** В чем регистры общего назначения и ячейки основной памяти схожи между собой?

   б. Чем различаются регистры общего назначения и ячейки основной памяти?

2. Дайте ответы на следующие вопросы по машинному языку Vole (см. приложение B).

   а. Запишите команду `0x2304` как строку из 16 битов.

   б. Запишите код операции команды `0xB2A5` как строку из 4 битов.

   в. Запишите поле операндов команды `0xB2A5` как строку из 12 битов.

3. Предположим, что блок данных записан в ячейках памяти машины Vole с адресами от `0x98` и до `0xA2` включительно. Сколько ячеек памяти содержит этот блок? Перечислите их адреса.

4. Каким будет значение в счетчике адреса машины Vole непосредственно после выполнения команды `0xB0CD`?

5. Предположим, что ячейки памяти с адресами от `0x00` до `0x05` в машине Vole содержат следующие битовые комбинации.

Адрес	Содержимое
`0x00`	`0x22`
`0x01`	`0x11`
`0x02`	`0x32`
`0x03`	`0x02`
`0x04`	`0xC0`
`0x05`	`0x00`

   Исходя из предположения, что исходно счетчик адреса содержал значение `0x00`, запишите содержимое счетчика адреса, регистра команд и ячейки памяти по адресу `0x02` в конце фазы выборки каждого машинного цикла до тех пор, пока машина не остановится.

6. Предположим, что в машинной памяти записаны три числа — x, y и z. Опишите последовательность действий (загрузка значений из памяти в регистры, сохранение результатов в памяти и т.д.), необходимых для вычисления суммы $x + y + z$. А какая последовательность действий потребуется для вычисления значения выражения $(2x) + y$?

7. Ниже приведено несколько команд на машинном языке Vole, описанном в приложении B. Дайте текстовое описание этих команд.

   а. `0x7123` **б.** `0x40E1` **в.** `0xA304` **г.** `0xB100` **д.** `0x2BCD`

8. Предположим, что в некотором машинном языке поле кода операции имеет длину 4 бита. Сколько различных машинных команд может существовать в этом языке? Что можно сказать по этому поводу, если длина поля кода операции будет увеличена до 6 бит?

9. Запишите приведенные ниже команды на машинном языке Vole, описанном в приложении B.

   а. Загрузить (LOAD) в регистр `0x6` шестнадцатеричное число `0x77`.

   б. Загрузить (LOAD) в регистр `0x7` содержимое ячейки памяти с адресом `0x77`.

   в. Выполнить переход (JUMP) к команде, расположенной по адресу `0x24`, если содержимое регистра `0x0` будет равно содержимому регистра `0xA`.

   г. Выполнить циклический сдвиг (ROTATE) содержимого регистра `0x4` на три бита вправо.

   д. Выполнить операцию AND над содержимым регистров `0xE` и `0x2`, поместив результат операции в регистр `0x1`.

10. Перепишите программу, представленную на рис. 2.7, полагая, что суммируемые значения представлены в формате с плавающей точкой, а не в двоичном дополнительном коде.

11. Распределите на три категории приведенные ниже команды (записанные на машинном языке Vole), исходя из того, изменит ли выполнение команды содержимое ячейки памяти с адресом `0x3C`, позволит ли выполнение команды считать содержимое ячейки памяти с адресом `0x3C` или же данная команда никак не зависит от содержимого ячейки памяти с адресом `0x3C`.

    а. `0x353C` **б.** `0x253C` **в.** `0x153C`

    г. `0x3C3C` **д.** `0x403C`

12. Предположим, что в машине Vole ячейки памяти с адресами от `0x00` до `0x03` содержат следующие битовые комбинации.

Адрес	Содержимое
`0x00`	`x26`
`0x01`	`x55`
`0x02`	`xC0`
`0x03`	`x00`

    а. Дайте текстовую формулировку первой команды.

    б. Если в начале работы машины содержимое счетчика адреса будет равно `0x00`, какая битовая комбинация окажется в регистре `0x6`, когда машина выполнит команду останова?

13. Предположим, что в машине Vole ячейки памяти с адресами от `0x00` до `0x02` содержат следующие битовые комбинации.

Адрес	Содержимое
0x00	0x12
0x01	0x21
0x02	0x34

а. Какой будет первая выполненная команда, если машина начнет работу со счетчиком адреса, равным 0x00?

б. Какой будет первая выполненная команда, если машина начнет работу со счетчиком адреса, равным 0x01?

14. Предположим, что в машине Vole ячейки памяти с адресами от 0x00 до 0x05 содержат следующие битовые комбинации.

Адрес	Содержимое
0x00	0x12
0x01	0x02
0x02	0x32
0x03	0x42
0x04	0xC0
0x05	0x00

Дайте ответы на следующие вопросы, полагая, что когда машина начинает работу, в ее счетчике адреса находится значение 0x00.

а. Сформулируйте текстовое описание каждой команды.

б. Какая битовая комбинация будет находиться в ячейке памяти с адресом 0x42, после того как машина выполнит команду останова?

в. Какая битовая комбинация будет находиться в счетчике адреса, когда машина выполнит команду останова?

15. Предположим, что в машине Vole ячейки памяти с адресами от 0x00 до 0x09 содержат следующие битовые комбинации.

Адрес	Содержимое
0x00	0x1C
0x01	0x03
0x02	0x2B
0x03	0x03
0x04	0x5A
0x05	0xBC
0x06	0x3A
0x07	0x00
0x08	0xC0
0x09	0x00

Будем считать, что машина начинает работу со счетчиком адреса, равным 0x00.

а. Какое значение будет находиться в ячейке памяти с адресом 0x00, когда машина выполнит команду останова?

б. Какая битовая комбинация будет находиться в счетчике адреса, когда машина выполнит команду останова?

16. Предположим, что в машине Vole ячейки памяти с адресами от `0x00` до `0x07` содержат следующие битовые комбинации.

Адрес	Содержимое
`0x00`	`0x2B`
`0x01`	`0x07`
`0x02`	`0x3B`
`0x03`	`0x06`
`0x04`	`0xC0`
`0x05`	`0x00`
`0x06`	`0x00`
`0x07`	`0x23`

а. Перечислите адреса ячеек памяти, содержащие программу, которая будет выполнена, если машина начнет работу со счетчиком адреса, равным `0x00`.

б. Перечислите адреса ячеек памяти, которые используются для хранения данных.

17. Предположим, что в машине Vole ячейки памяти с адресами от `0x00` до `0x0D` содержат следующие битовые комбинации.

Адрес	Содержимое
`0x00`	`0x20`
`0x01`	`0x04`
`0x02`	`0x21`
`0x03`	`0x01`
`0x04`	`0x40`
`0x05`	`0x12`
`0x06`	`0x51`
`0x07`	`0x12`
`0x08`	`0xB1`
`0x09`	`0x0C`
`0x0A`	`0xB0`
`0x0B`	`0x06`
`0x0C`	`0xC0`
`0x0D`	`0x00`

Будем считать, что машина начинает работу со счетчиком адреса, равным `0x00`.

а. Какая битовая комбинация будет находиться в регистре `0x0`, когда машина выполнит команду останова?

б. Какая битовая комбинация будет находиться в регистре `0x1`, когда машина выполнит команду останова?

в. Какая битовая комбинация будет находиться в счетчике адреса, когда машина выполнит команду останова?

18. Предположим, что в машине Vole ячейки памяти с адресами от `0xF0` до `0xFD` содержат следующие (шестнадцатеричные) битовые комбинации.

Адрес	Содержимое
`0xF0`	`0x20`
`0xF1`	`0x00`
`0xF2`	`0x22`
`0xF3`	`0x02`
`0xF4`	`0x23`
`0xF5`	`0x04`
`0xF6`	`0xB3`
`0xF7`	`0xFC`
`0xF8`	`0x50`
`0xF9`	`0x02`
`0xFA`	`0xB0`
`0xFB`	`0xF6`
`0xFC`	`0xC0`
`0xFD`	`0x00`

Если машина начнет работу со счетчиком адреса, имеющим значение `0xF0`, какое значение будет находиться в регистре `0x0`, когда машина выполнит команду останова, расположенную в ячейке с адресом `0xFC`?

19. Если машина Vole выполняет каждую команду за одну микросекунду (миллионная доля секунды), сколько времени займет выполнение программы, приведенной в задании 18?

20. Предположим, что в машине Vole ячейки памяти с адресами от `0x20` до `0x28` содержат следующие битовые комбинации.

Адрес	Содержимое
`0x20`	`0x12`
`0x21`	`0x20`
`0x22`	`0x32`
`0x23`	`0x30`
`0x24`	`0xB0`
`0x25`	`0x21`
`0x26`	`0x24`
`0x27`	`0xC0`
`0x28`	`0x00`

Будем считать, что машина начинает работу со счетчиком адреса, равным `0x20`.

а. Какие комбинации битов будут находиться в регистрах `0x0`, `0x1` и `0x2`, когда машина выполнит команду останова?

б. Какая комбинация битов будет находиться в ячейке памяти с адресом `0x30`, когда машина выполнит команду останова?

в. Какая комбинация битов будет находиться в ячейке памяти с адресом `0xB0`, когда машина выполнит команду останова?

21. Предположим, что в машине Vole ячейки памяти с адресами от `0xAF` до `0xB1` содержат следующие битовые комбинации.

Адрес	Содержимое
`0xAF`	`0xB0`
`0xB0`	`0xB0`
`0xB1`	`0xAF`

Что произойдет, если машина начнет работу со счетчиком адреса, равным `0xAF`?

22. Предположим, что в машине Vole в ячейках памяти с адресами от `0x00` до `0x05` содержатся следующие (шестнадцатеричные) битовые комбинации.

Адрес	Содержимое
`0x00`	`0x25`
`0x01`	`0xB0`
`0x02`	`0x35`
`0x03`	`0x04`
`0x04`	`0xC0`
`0x05`	`0x00`

Когда машина выполнит команду останова, если она начнет работу при содержимом счетчика адреса `0x00`?

23. Для каждого приведенного ниже задания напишите на машинном языке Vole небольшую программу, предназначенную для его выполнения. Исходите из того, что каждая программа будет записана в памяти начиная с адреса `0x00`.

а. Переместите значение, сохраняемое в ячейке памяти с адресом `0xD8`, в ячейку с адресом `0xB3`.

б. Поменяйте местами значения, записанные в ячейках с адресами `0xD8` и `0xB3`.

в. Если значение, записанное в ячейке памяти с адресом `0x44`, равно `0x00`, запишите значение `0x01` в ячейку памяти с адресом `0x46`, в противном случае запишите в ячейку `0x46` значение `0xFF`.

24. Одной из популярных среди любителей компьютеров игр является *core wars* (война сердечников). (Термин *core* использовался в устаревших технологиях создания основной памяти, в которых нули и единицы представлялись магнитными полями в небольших кольцевых сердечниках из магнитного материала.) Игра проводится между двумя программами, выступающими одна против другой, которые записаны по разным адресам общей памяти компьютера. Предполагается, что компьютер постоянно переключается с одной программы на другую, т.е. выполняет одну команду первой программы, после чего немедленно выполняет одну команду второй программы и т.д. Цель каждой программы — уничтожение другой

программы посредством записи посторонних данных поверх ее текста, сохраняемого в основной памяти. Однако ни одна из программ не знает места расположения другой программы.

а. Напишите игровую программу на машинном языке Vole, реализующую стратегию обороны, т.е. имеющую минимально возможный размер.

б. Напишите игровую программу на машинном языке Vole, стратегия которой будет построена на стремлении избежать любых нападений другой программы посредством перемещения самой себя по различным адресам основной памяти. Говоря точнее, напишите программу, начинающуюся с адреса `0x00`, которая при выполнении должна копировать себя в ячейки памяти, начиная с адреса `0x70`, а затем передавать управление по адресу `0x70`.

в. Модернизируйте программу из п. *б*, чтобы она продолжила свое перемещение по новым адресам. В частности, пусть программа переместит себя по адресу `0x70`, затем — по адресу `0xE0` (= `0x70` + `0x70`), а затем — по адресу `0x60` (= `0x70` + `0x70` + `0x70`) и т.д.

25. Напишите на машинном языке Vole программу, которая будет вычислять сумму чисел, представленных в формате с плавающей точкой и сохраняемых в ячейках с адресами `0xA0`, `0xA1`, `0xA2` и `0xA3`. Результат должен быть записан в ячейку с адресом `0xA4`.

26. Предположим, что в машине Vole ячейки памяти с адресами от `0x00` до `0x05` содержат следующие битовые комбинации (шестнадцатеричное представление).

Адрес	Содержимое
`0x00`	`0x20`
`0x01`	`0xC0`
`0x02`	`0x30`
`0x03`	`0x04`
`0x04`	`0x00`
`0x05`	`0x00`

Что произойдет, если машина начнет работу со счетчиком адреса, имеющим значение `0x00`?

27. Что произойдет, если в машине Vole ячейки памяти с адресами `0x08` и `0x09` будут содержать битовые комбинации `0xB0` и `0x08` соответственно и машина начнет работу со счетчиком адреса, имеющим значение `0x08`?

28. Предположим, что приведенная ниже программа написана на машинном языке Vole и записана в память машины начиная с ячейки с адресом `0x30` (шестнадцатеричное представление). Какое задание будет выполнено программой после ее завершения?

```
0x2003
0x2101
0x2200
0x2310
0x1400
0x3410
0x5221
0x5331
0x3239
0x333B
0xB248
0xB038
0xC000
```

29. Опишите этапы выполнения машиной Vole команды с кодом операции `0xB`. Представьте свой ответ в виде набора инструкций, указывающих центральному процессору, что и как нужно делать.

***30.** Опишите этапы выполнения машиной Vole команды с кодом операции `0x5`. Представьте свой ответ в виде набора инструкций, указывающих центральному процессору, что и как нужно делать.

***31.** Опишите этапы выполнения машиной Vole команды с кодом операции `0x6`. Представьте свой ответ в виде набора инструкций, указывающих центральному процессору, что и как нужно делать.

***32.** Предположим, что регистры `0x4` и `0x5` в машине Vole содержат битовые комбинации `0x3A` и `0xC8` соответственно. Какая комбинация окажется в регистре `0x0` после выполнения следующих команд?

а. `0x5045` **б.** `0x6045` **в.** `0x7045`

г. `0x8045` **д.** `0x9045`

***33.** Напишите на машинном языке Vole программы для выполнения приведенных ниже заданий.

а. Скопируйте битовую комбинацию, записанную в ячейке памяти с адресом `0x44`, в ячейку с адресом `0xAA`.

б. Присвойте четырем младшим битам ячейки памяти с адресом `0x34` значение `0`, не изменяя при этом значения остальных ее битов.

в. Скопируйте четыре младших бита из ячейки памяти с адресом `0xA5` в четыре младших бита ячейки с адресом `0xA6`, не изменяя при этом значения остальных битов ячейки с адресом `0xA6`.

г. Скопируйте четыре младших бита из ячейки памяти с адресом `0xA5` в четыре старших бита этой же ячейки. (В результате первые четыре бита в ячейке с адресом `0xA5` будут идентичны ее последним четырем битам.)

*34. Выполните указанные операции.

а. 111001 AND 101001	**б.** 000101 AND 101010	**в.** 001110 AND 010101
г. 111011 AND 110111	**д.** 111001 OR 101001	**е.** 010100 OR 101010
ж. 00100 OR 010101	**з.** 101010 OR 110101	**и.** 111001 XOR 101001
к. 000111 XOR 101010	**л.** 010000 XOR 010101	**м.** 111111 XOR 110101

*35. Укажите значение маски и тип логической операции, необходимые для выполнения указанных ниже действий.

а. Поместите значение `1` в четыре старших бита восьмибитовой комбинации, не изменяя состояния других ее битов.

б. Определите двоичное дополнение для старшего бита восьмибитовой ячейки памяти без изменения состояния остальных ее битов.

в. Получите двоичное дополнение для комбинации из восьми битов.

г. Поместите значение `0` в младший бит восьмибитовой комбинации, не изменяя состояния остальных ее битов.

д. Поместите значение `1` во все биты восьмибитовой комбинации, кроме самого старшего, оставив его значение неизменным.

е. Отфильтруйте всю зеленую компоненту пикселей растрового изображения в формате RGB, если для их сохранения используются 8 средних битов в 24-битовом формате представления данных о цветности.

ж. Инвертируйте все биты в 24-битовом представлении цвета пикселей растрового изображения в формате RGB.

з. Установите равными единице все биты в 24-битовом представлении цвета пикселей растрового изображения в формате RGB, что будет соответствовать "перекрашиванию" этих пикселей в белый цвет.

*36. Напишите и протестируйте краткие сценарии на языке Python, в которых будут реализованы ответы на все пункты предыдущего задания.

*37. Укажите тип логической операции (вместе с соответствующей маской), которая при применении к входной восьмибитовой строке даст на выходе строку из всех нулей тогда и только тогда, когда входная строка будет иметь значение `10000001`.

*38. Напишите и протестируйте краткий сценарий на языке Python, в котором будет реализован ответ на предыдущее задание.

*39. Укажите последовательность логических операций (вместе с соответствующими масками), которые при ее применении к входной восьмибитовой строке дадут на выходе строку из всех нулей тогда и только тогда, когда входная строка будет начинаться и заканчиваться битом со значением `1`. Во всех остальных случаях выходная строка должна будет содержать по крайней мере один единичный бит.

*40. Напишите и протестируйте сценарий на языке Python, в котором будет реализован ответ на предыдущее задание.

*41. Каков будет результат выполнения операции циклического сдвига на четыре бита влево для следующих битовых комбинаций?

а. `10101` **б.** `11110000` **в.** `001`

г. `101000` **д.** `00001`

*42. Каким будет результат выполнения циклического сдвига на два бита вправо для следующих байтов, представленных в шестнадцатеричной нотации (ответы также должны быть представлены в шестнадцатеричной нотации).

а. `0x3F` б. `0x0D` **в.** `0xFF` г. `0x77`

*43. **а.** Какая единственная команда в машинном языке Vole может быть использована для выполнения циклического сдвига на 5 битов вправо содержимого регистра `0xB`?

б. Какая единственная команда в машинном языке Vole может быть использована для выполнения циклического сдвига влево на 2 бита содержимого регистра `0xB`?

*44. Напишите программу на машинном языке Vole, которая изменит содержимое ячейки памяти с адресом `0x8C` на обратное. (То есть конечная последовательность битов в ячейке с адресом `0x8C`, прочитанная слева направо, будет соответствовать исходной последовательности битов в этой ячейке, прочитанной справа налево.)

*45. Напишите программу на машинном языке Vole, которая будет вычитать значение, сохраненное в ячейке с адресом `0xA1`, из значения, сохраненного в ячейке с адресом `0xA2`, и записывать результат в ячейку с адресом `0xA0`. Будем считать, что все значения будут представлены в двоичном дополнительном коде.

*46. Данные в формате видео высокого разрешения предполагают передачу 30 кадров в секунду, и при этом каждый кадр имеет разрешение 1920 × 1080 пикселей, цвета которых закодированы с использованием 24 битов на пиксель. Можно ли отправить несжатые данные этого формата через последовательное соединение по стандарту USB 1.1? А через последовательное соединение стандарта USB 2.0? Что можно сказать об использовании соединения

стандарта USB 3.0? (*На заметку*: максимальная скорость передачи данных для последовательных соединений по стандартам USB 1.1, USB 2.0 и USB 3.0 составляет 12 Мбит/с, 480 Мбит/с и 5 Гбит/с соответственно.)

***47.** Предположим, что человек вводит с клавиатуры по 40 слов в минуту, причем каждое слово состоит из пяти символов. Если машина каждую микросекунду (миллионная доля секунды) выполняет 500 команд, то сколько команд выполнит эта машина, пока с клавиатуры будут введены два последовательных символа?

***48.** Сколько битов в секунду должна передавать клавиатура в компьютер, чтобы успевать за пользователем, выполняющим ввод со скоростью 40 слов в минуту? (Предположим, что каждый символ шифруется в коде ASCII и имеет бит четности, а каждое слово состоит из шести символов.)

***49.** Предположим, что машина Vole для работы с принтером использует метод отображения ввода-вывода в память. Также предположим, что адрес `0xFF` используется для передачи символов в принтер, а адрес `0xFE` — для получения информации о состоянии принтера. В частности, будем считать, что младший бит ячейки с адресом `0xFE` определяет готовность принтера получить еще один символ (значение `0` означает неготовность, а значение `1` — готовность получить очередной символ). Напишите на машинном языке программу, начинающуюся с адреса `0x00`, которая будет ожидать, пока принтер сообщит о готовности принять очередной символ, а затем отправит ему символ, представленный битовой комбинацией в регистре `0x5`.

***50.** Напишите программу на машинном языке Vole, которая будет помещать нулевые значения во все ячейки основной памяти с адресами от `0xA0` до `0xC0` и одновременно будет достаточно небольшого размера, чтобы помещаться в ячейках памяти с адресами от `0x00` до `0x13`.

***51.** Предположим, что машина, имеющая на жестком диске 200 Гбайт свободной памяти, принимает данные по широкополосной линии связи со скоростью 15 Мбит/с. Сколько времени ей понадобится, чтобы заполнить данными все свободное место на жестком диске?

***52.** Предположим, что спутниковая система используется для получения потока последовательности данных со скоростью 250 Кбит/с. Если вспышка атмосферных помех продлится 6,96 с, то сколько битов данных будет ею затронуто?

***53.** Предположим, что имеется 32 процессора, каждый из которых способен определить сумму двух многозначных чисел за миллионную долю секунды. Опишите, как применить методы параллельной обработки, чтобы обеспечить вычисление суммы 64 чисел всего за шесть миллионных

долей секунды. Сколько времени потребуется одному процессору для определения этой суммы?

***54.** Кратко опишите основные различия между CISC- и RISC-архитектурами.

***55.** Опишите два подхода к увеличению пропускной способности машины.

***56.** Объясните, как среднее значение для некоторого набора чисел может быть вычислено быстрее в многопроцессорной машине, чем в машине, имеющей только один процессор.

***57.** Напишите на языке Python и протестируйте сценарий, который будет вводить число в формате с плавающей точкой, представляющее радиус круга, а затем выполнять необходимые расчеты и выводить длину его окружности и площадь.

***58.** Напишите на языке Python и протестируйте сценарий, который будет вводить строку символов и целое число, а затем выводить эту строку символов столько раз, сколько указано этим целым числом.

***59.** Напишите на языке Python и протестируйте сценарий, который будет вводить два числовых значения в формате с плавающей точкой, представляющих длины двух сторон прямоугольного треугольника, а затем выполнять необходимые расчеты и выводить длину его гипотенузы, периметр и площадь.

СОЦИАЛЬНЫЕ И ОБЩЕСТВЕННЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны разобраться, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. Предположим, что фирма — изготовитель компьютеров разработала новую архитектуру машины. В какой степени данной фирме может быть позволено обладать правом собственности на эту архитектуру? Какая политика в подобных случаях будет оптимальной с точки зрения общества?
2. В какой-то мере 1923 год можно считать годом рождения так называемой *модели планируемого устаревания*. Именно в этом году компания General Motors, возглавляемая Альфредом Слоэном (Alfred Sloan), ввела в автомобильной индустрии понятие модели года. Идея состояла в том, чтобы повысить уровень продаж посредством изменения моды, а не изготовления

лучшей машины. Часто цитируется следующее высказывание Слоэна: "Мы хотим заставить вас быть неудовлетворенными имеющейся у вас сейчас моделью машины, чтобы вы покупали новые". В какой мере эта коммерческая уловка применяется в современной компьютерной индустрии?

3. Мы часто думаем о том, как компьютерная технология изменила наше общество. Однако многие считают, что эта технология часто была сдерживающим фактором для распространения многих нововведений только потому, что поддерживала старые системы и позволяла им не только выжить, но и укрепить свои позиции. Как вы думаете, осталось бы неизменным влияние правительства США на жизнь общества, если бы компьютерных технологий не существовало? Какую степень влияния имела бы сейчас централизованная власть, если бы компьютерные технологии были для нее недоступны? Как бы мы жили, лучше или хуже, при отсутствии компьютерных технологий?
4. Можно ли считать этичным для личности утверждение, что ей не нужно знать какие-либо подробности об устройстве и принципах работы машины, поскольку есть люди, которые занимаются разработкой таких машин, их обслуживанием и решением всех проблем, возникающих при их эксплуатации? Будет ли предоставленный вами ответ отличаться в зависимости от того, о чем будет идти речь — о компьютере, автомобиле, атомной электростанции или тостере?
5. Предположим, что фирма-изготовитель выпустила на рынок компьютерную микросхему, а позднее обнаружила в ее конструкции недоработку. В результате она решает не отзывать уже выпущенные схемы и сохраняет сведения о дефекте в секрете, мотивируя это тем, что ни одна из выпущенных микросхем не используется в таких приложениях, в которых наличие подобной недоработки может иметь какие-либо последствия. Принесло ли кому-либо вред такое решение фирмы? Является ли решение фирмы-изготовителя обоснованным, если никто не пострадал и принятие такого решения защитило фирму от больших убытков и, возможно, от необходимости увольнения работников?
6. Помогает ли развитие технологии успешнее лечить заболевания сердца или, наоборот, ведет к сидячему образу жизни, который является серьезной причиной сердечно-сосудистых заболеваний?
7. Легко представить себе те финансовые или навигационные катастрофы, которые могут произойти в результате арифметических просчетов, вызванных ошибками усечения значения или переполнения. А какими, по вашему мнению, будут последствия ошибок в системах запоминания

изображений (например, тех, которые используются при зондировании или определении медицинского диагноза)?

8. Компания ARM Holdings — это небольшая компания, проектирующая процессоры для широкого диапазона бытовых электронных приборов. Она не изготавливает процессоры, а лишь лицензирует свои разработки другим фирмам — изготовителям полупроводниковых устройств (таким, как Qualcomm, Samsung и Texas Instruments), которые отчисляют им как владельцам патента определенную сумму за каждую изготовленную единицу продукции. Такая бизнес-модель получила широкое распространение на рынке бытовой электроники по причине высокой стоимости проведения новых исследований и разработок в области процессоров для специализированных компьютеров. Сегодня процессоры компании ARM работают более чем в 95% всех сотовых телефонов (а не только смартфонов), более чем в 40% всех цифровых фотокамер и примерно в 25% цифровых телевизоров. Более того, ARM-процессоры можно найти в планшетах, аудиоплеерах, игровых приставках, электронных книгах, навигационных системах. И этот перечень можно продолжить. Принимая все это во внимание, склонны ли вы рассматривать компанию ARM Holdings как монополиста? Почему "да" или почему "нет"? Поскольку в современном обществе бытовые электронные устройства играют все возрастающую роль, является ли зависимость в этом вопросе от некоей малоизвестной компании положительным фактором или такое положение дел может стать причиной возникновения определенных проблем?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Carpinelli J.D. *Computer Systems Organization and Architecture.* — Boston, MA: Addison-Wesley, 2001.
2. Comer D.E. *Essentials of Computer Architecture.* — Upper Saddle River, NJ: Prentice-Hall, 2005.
3. Dandamudi S.P. *Guide to RISC Processors for Programmers and Engineers.* — New York: Springer, 2005.
4. Furber S. *ARM System-on-Chip Architecture*, 2nd ed. — Boston, MA: AddisonWesley, 2000.
5. Hamacher V.C., Vranesic Z.G., Zaky S.G. *Computer Organization*, 5th ed. — New York: McGraw-Hill, 2002.
6. Knuth D.E. *The Art of Computer Programming*, Vol. 1, 3rd ed. — Boston, MA: Addison-Wesley, 1998. (Имеется русский перевод этой книги: Кнут Д.Э.

Искусство программирования. Т. 1. Получисленные алгоритмы, 3-е изд. — М.: Издательский дом "Вильямс", 2000.)

7. Murdocca M.J., Heuring V.P. *Computer Architecture and Organization: An Integrated Approach*. — New York: Wiley, 2007.
8. Stallings W. *Computer Organization and Architecture*, 9th ed. — Upper Saddle River, NJ: Prentice-Hall, 2012.
9. Tanenbaum A. S. *Structured Computer Organization*, 6th ed. — Upper Saddle River, NJ: Prentice-Hall, 2012.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Столлингс В. *Структурная организация и архитектура компьютерных систем*, 5-е изд. — М.: Издательский дом "Вильямс", 2002.

Глава 3

В этой главе мы познакомимся с операционными системами, представляющими собой сложные пакеты программ, координирующих всю деятельность внутри машины, а также контролирующих все ее взаимодействие с внешним миром. Именно операционная система компьютера превращает весь набор его сложнейшего оборудования в полезный инструмент. Наша задача — разобраться, что именно делает операционная система в компьютере и как она это делает. Знание этого совершенно необходимо для того, чтобы стать достаточно знающим пользователем компьютера, понимающим, какими будут результаты его действий.

Операционные системы

3.1. ЭВОЛЮЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

3.2. АРХИТЕКТУРА ОПЕРАЦИОННЫХ СИСТЕМ
- Обзор программного обеспечения
- Компоненты операционной системы
- Запуск операционной системы

3.3. КООРДИНАЦИЯ ДЕЙСТВИЙ МАШИНЫ
- Понятие процесса
- Управление процессами

*3.4. ОРГАНИЗАЦИЯ КОНКУРЕНЦИИ МЕЖДУ ПРОЦЕССАМИ
- Семафоры
- Взаимная блокировка

3.5. БЕЗОПАСНОСТЬ
- Атаки снаружи
- Атаки изнутри

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

Операционной системой (ОС) называют программное обеспечение, управляющее всей работой компьютера. Она предоставляет средства, с помощью которых пользователь может сохранять и считывать файлы, обеспечивает интерфейс, пользуясь которым пользователь может потребовать выполнения программ, и формирует окружение, необходимое для выполнения этих программ.

Вероятно, наиболее известным примером операционной системы является ОС Windows, многочисленные версии которой, созданные корпорацией Microsoft, получили самое широкое распространение в среде персональных компьютеров. Другим примером широко распространенной операционной системы является UNIX, которая обычно используется в более крупных компьютерных системах, но ее достаточно часто можно найти и на ПК. В действительности ОС UNIX была использована как базовая также при создании двух других популярных операционных систем: Mac OS, которая является операционной системой, которую корпорация Apple устанавливает на широком диапазоне своих машин с общим названием "Mac", и ОС Solaris, которая была разработана компанией Sun Microsystems (теперь принадлежащей корпорации Oracle). Еще одним примером операционной системы, которую можно найти как на больших, так и на малых машинах, является Linux. Эта ОС изначально была разработана энтузиастами-компьютерщиками на некоммерческой основе, а сейчас является доступной через многочисленные коммерческие источники, включая корпорацию IBM.

Для рядовых пользователей компьютеров различия между существующими операционными системами являются по большей части чисто косметическими. Но от компьютерщиков-профессионалов смена операционной системы может потребовать весьма значительных изменений в используемом инструментарии, а также смены тех ключевых положений, которыми они руководствуются при распространении и сопровождении результатов своей работы. Но как бы там ни было, по своей сути все ведущие операционные системы предназначены, прежде всего, для решения одних и тех же проблем, с которыми специалисты компьютерных наук сталкивались на протяжении более чем половины столетия.

3.1. Эволюция операционных систем

Современные операционные системы являются сложными, комплексными пакетами программ, выросшими из очень скромных начал. В 1940- и 1950-х годах компьютеры были недостаточно гибкими и эффективными. Каждая машина занимала большое помещение. Выполнение программ требовало значительной предварительной подготовки оборудования: установки магнитных лент, загрузки перфокарт в устройство чтения перфокарт, установки переключателей и т.д. Запуск каждой программы, называемый **заданием** (*job*), производился по

отдельности: машину готовили к выполнению программы, а затем программа выполнялась, после чего все ленты, перфокарты и тому подобное следовало убрать, чтобы можно было начать подготовку к выполнению следующей программы. Если нескольким пользователям нужно было работать на одной и той же машине, то предварительно составлялось специальное расписание, позволяющее им зарезервировать машинное время. На протяжении отведенного времени машина находилась полностью в распоряжении пользователя. Сеанс обычно начинался с подготовки машины, после чего следовало выполнение самой программы. Чаще всего сеанс заканчивался лихорадочными усилиями пользователя сделать что-то еще ("Это займет только одну минуту!"), в то время как следующий пользователь с нетерпением ожидал, когда он сможет приступить к подготовке машины для своей задачи.

В подобной ситуации операционные системы создавались как средство для упрощения подготовки программ и ускорения перехода от одного задания к другому. Первоначальной идеей создания подобных систем было отделение пользователя от оборудования, что позволяло избавиться от постоянного потока людей, входящих и выходящих из помещения, в котором находилась машина. С этой целью была введена должность оператора компьютера, задача которого состояла в выполнении всех операций непосредственно на оборудовании. Каждый пользователь должен был предоставить оператору программу вместе с необходимыми данными и специальными указаниями о ее требованиях, а затем вернуться за результатами. Оператор загружал полученные материалы в массовую память машины, откуда операционная система могла отправить их на выполнение. Это было началом пакетной обработки — метода выполнения заданий посредством их предварительного объединения в единый пакет, который затем выполнялся без дальнейшего взаимодействия с пользователем.

В системах пакетной обработки задания, ожидающие своего выполнения в массовой памяти, образуют **очередь заданий** (рис. 3.1). **Очередь** — это средство организации памяти, когда сохраняемые в ней объекты (в нашем случае — задания) упорядочены по принципу **"Первым вошел — первым вышел"** (FIFO — *First-in, First-out*). Иначе говоря, объекты покидают очередь в том же порядке, в котором в нее поступают. На самом деле большинство очередей задач не следует в точности принципу FIFO, так как большая часть операционных систем поддерживает установку приоритетов задач. В результате выполнение находящегося в очереди задания может быть отодвинуто на более поздний срок заданием с более высоким приоритетом.

В ранних системах пакетной обработки каждое задание сопровождалось рядом инструкций, описывающих шаги, необходимые для подготовки машины к выполнению этого конкретного задания. Эти инструкции кодировались на языке управления заданиями (JCL — *Job Control Language*) и помещались вместе

с заданием в очередь задач. Когда задание выбиралось для выполнения, операционная система распечатывала эти инструкции на принтере, чтобы оператор мог их прочитать и выполнить. Такое взаимодействие между операционной системой и пользователем компьютера в некоторой степени сохранилось и до сих пор. Свидетельством тому являются сообщения об ошибках, такие как "Нет доступа к сети" или "Принтер не отвечает", которые операционные системы ПК выводят пользователям на экран.

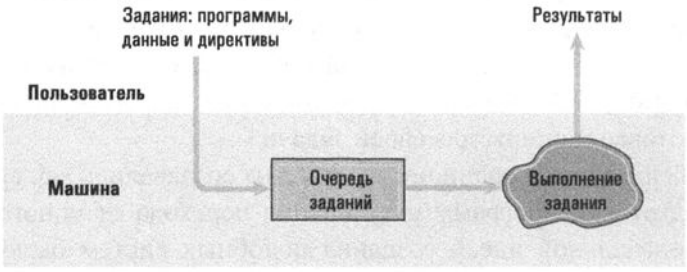


Рис. 3.1. Пакетная обработка

Главным недостатком традиционной пакетной обработки с участием оператора компьютера является то, что пользователь лишен возможности взаимодействовать с программой с того момента, как она передана оператору на выполнение. Такой подход допустим для приложений, в которых все данные и процедуры определены заранее (например, при подготовке платежных ведомостей). Однако он неприемлем, если пользователю необходимо взаимодействовать с программой в процессе ее выполнения. Примером могут служить системы резервирования (мест, билетов), в которых информация о резервировании и отменах должна быть доступна сразу же после поступления; системы обработки текстов, поддерживающие интерактивное создание и обновление документов; а также компьютерные игры, в которых взаимодействие с машиной является основным элементом игры.

Для решения этих проблем были созданы новые операционные системы, которые позволяли выполнять программы, ведущие диалог с пользователем, работающим за удаленным терминалом или рабочей станцией. Такой режим функционирования называется **интерактивной обработкой** (рис. 3.2). (Терминал включал по меньшей мере электронную пишущую машинку, на которой пользователь мог вводить свои инструкции и читать сообщения, напечатанные на бумаге компьютером. Хотя современные ПК, ноутбуки и смартфоны являются значительно более мощными устройствами в сравнении с компьютерами и терминалами прежних времен, ирония состоит в том, что мы по-прежнему часто используем их просто для взаимодействия с куда более мощными облачными сервисами Интернета.)

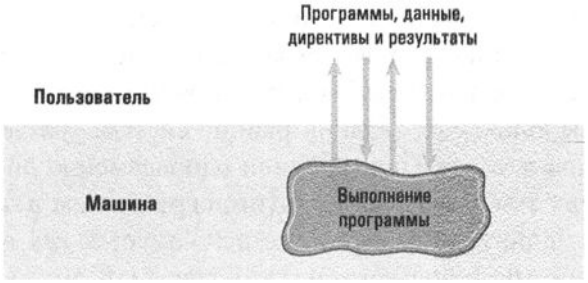


Рис. 3.2. Интерактивная обработка

Важнейшим условием для успешной интерактивной обработки является то, что действия компьютера должны быть достаточно быстрыми, чтобы координировать их с потребностями пользователя, а не заставлять его подстраиваться под установленное для нее расписание. Задача обработки платежной ведомости вполне может быть запланирована с учетом количества времени, которое потребуется компьютеру на ее выполнение, а вот работа с текстовым процессором будет очень неудобной, если машина не будет немедленно отвечать пользователю после ввода им очередного символа. В некотором смысле компьютер здесь вынужден выполнять задания в жестко установленный срок, — это процесс, который получил название **обработка в реальном времени**, поскольку именно в таком режиме выполняемые машиной действия должны координироваться с происходящими в ее окружении событиями.

Если бы от интерактивных систем требовалось обслуживать в каждый момент времени только одного пользователя, то обработка в реальном времени не вызывала бы никаких проблем. Однако вычислительные машины в 1960- и 1970-х годах стоили дорого, поэтому каждая машина должна была обслуживать несколько пользователей. В свою очередь, вполне типичной была ситуация, когда одновременно нескольким пользователям требовалось получить с удаленных терминалов доступ к машине в интерактивном режиме, в результате чего выполнение работы в реальном времени становилось уже затруднительным. Если в такой многопользовательской среде операционная система будет придерживаться правила строго поочередного выполнения заданий, то только один из пользователей сможет получить удовлетворительное обслуживание в реальном времени.

Решением этой проблемы стала разработка операционных систем, способных успешно обслуживать одновременно нескольких пользователей с помощью процесса, получившего название **разделение времени**. Этот метод заключается в разделении машинного времени на интервалы, или *кванты*, с последующим ограничением времени непрерывного выполнения каждой программы одним квантом времени за один раз. В конце каждого интервала выполнение текущего

задания временно приостанавливается и во время следующего кванта выполняется новое. При быстром чередовании заданий подобным образом создается иллюзия, что несколько заданий выполняется в машине одновременно. В зависимости от типа выполняемых заданий ранние системы разделения времени позволяли обслуживать в реальном времени одновременно до 30 пользователей. Такой режим работы назывался **мультипрограммным режимом**. В наши дни работа в мультипрограммном режиме используется как в персональных компьютерах, так и в многопользовательских системах, хотя в первом случае такой принцип работы принято называть **мультизадачным режимом**.

С появлением многопользовательских операционных систем с разделением времени типичная вычислительная система стала представлять собой большой центральный компьютер, соединенный с многочисленными рабочими станциями. С помощью рабочих станций пользователи могли непосредственно взаимодействовать с компьютером, находясь при этом за пределами машинного зала вычислительного комплекса, вместо того чтобы направлять оператору компьютера свои заявки на выполнение той или иной работы. Часто используемые программы хранились непосредственно в устройствах массовой памяти компьютера, и операционные системы были спроектированы так, чтобы уметь вызывать эти программы на выполнение по запросам пользователей, поступающим с рабочих станций. С этого времени роль оператора компьютера как промежуточного звена между пользователями и компьютером начала постепенно терять свое значение.

В наши дни операторы компьютеров уже практически исчезли, особенно на тех участках, где работа ведется на персональных компьютерах. Теперь все обязанности по работе с ПК возложены непосредственно на его пользователя. Даже самые крупные компьютерные системы сейчас работают практически без присмотра. В действительности с течением времени операторы компьютеров уступили свое место системным администраторам, которые теперь управляют компьютерными системами. Их задачи — получение и установка нового оборудования и программного обеспечения, гарантия соблюдения установленных в системе требований, таких как выдача новых учетных записей и установка ограничений на использование массовой памяти отдельными пользователями, а также координация усилий по решению возникающих в системе проблем. Как видите, о ручном управлении работой машин уже не может быть и речи.

В двух словах можно сказать, что операционные системы прошли долгий путь от простых программ, которые последовательно считывали и выполняли указанные задания, до сложнейших комплексных систем, которые координируют процессы разделения времени, обеспечивают хранение и доступ к программам и файлам данных в устройствах массовой памяти компьютеров и непосредственно отвечают на любые запросы и указания, поступающие от пользователей.

Однако эволюция операционных систем все еще продолжается. Появление многопроцессорных машин потребовало создания операционных систем, реализующих процессы разделения времени и мультизадачности посредством назначения разных заданий разным процессорам, одновременно с обеспечением режима разделения времени в каждом отдельном процессоре. Подобные операционные системы должны справляться с такими проблемами, как **балансировка загрузки** (load balancing) — динамическое распределение задач между процессорами таким образом, чтобы все они эффективно использовались, а также **масштабирование** (*scaling*) — разбиение задачи на количество подзадач, совместимое с количеством доступных процессоров.

Более того, появление компьютерных сетей, в которых множество машин, разбросанных на большие расстояния, соединено в единую систему, привело к созданию программных комплексов, предназначенных для координации сетевой активности. В результате технология компьютерных сетей (подробно эта тема будет рассмотрена в главе 4) во многих отношениях расширила требования к операционным системам, выдвинув на передний план проблему управления работой множества ресурсов у разных пользователей, работающих на различных машинах, вместо управления работой одного отдельного компьютера.

Что находится внутри смартфона?

По мере того как сотовые телефоны становились все более и более мощными, появилась возможность предложить их владельцам дополнительные функции и инструменты, выходящие далеко за рамки обычной голосовой телефонной связи. Сейчас типичный смартфон можно использовать для отправки текстовых сообщений, посещения веб-сайтов, определения маршрута движения, просмотра контента мультимедиа... Короче говоря, его возможности уже мало чем отличаются от возможностей обычного ПК, а кое в чем и превосходят его. В результате смартфоны требуют наличия операционной системы, которая будет не только контролировать использование ограниченных ресурсов оборудования смартфона, но и обеспечивать необходимую функциональную поддержку быстро расширяющегося рынка прикладного программного обеспечения для смартфонов. Битва за доминирование на рынке операционных систем для смартфонов обещает быть ожесточенной и, скорее всего, будет вестись за то, на базе какой системы можно будет реализовать наиболее впечатляющие функциональные возможности по наилучшей цене. На рынке операционных систем для смартфонов главными игроками сейчас являются корпорация Google и ее ОС Android, корпорация Apple и ее ОС для iPhone, компания Research In Motion и ее ОС BlackBerry, а также корпорация Microsoft и ее ОС Windows Phone.

Совершенно иным направлением исследований в области операционных систем является ситуация управления работой таких устройств, которые предназначены для выполнения конкретных задач, например медицинского оборудования, бортовых компьютеров различных средств передвижения, устройств бытовой техники, сотовых телефонов или других компактных вычислительных устройств. Компьютерные системы в таких устройствах называют **встроенными системами**. Операционные системы встроенных компьютеров часто должны следить за экономным расходом заряда батарей, отвечать жестким требованиям работы в режиме реального времени или непрерывно функционировать при минимальном или вообще отсутствующем контроле со стороны человека. Успехи в этой области отмечены такими системами, как VxWORKS, разработанной компанией Wind River Systems и использованной в американских марсоходах, получивших названия Spirit и Opportunity; операционными системами Windows CE (также известной как Pocket PC), разработанной корпорацией Microsoft, и Palm OS, разработанной компанией PalmSource, Inc., предназначенными для использования в компактных вычислительных устройствах и средствах передвижения.

3.1. Вопросы и упражнения

1. Приведите примеры очередей. В каждом случае укажите любые ситуации, способные нарушить FIFO-структуру очереди.
2. Какие из приведенных ниже ситуаций требуют обработки в реальном времени?
   - **а.** Печать почтовых этикеток с адресами.
   - **б.** Компьютерная игра.
   - **в.** Отображение на экране смартфона цифр телефонного номера по мере их набора на клавиатуре.
   - **г.** Выполнение программы, предсказывающей состояние экономики в будущем году.
   - **д.** Воспроизведение аудиозаписей в формате MP3.
3. В чем состоят различия между встроенными системами и персональными компьютерами?
4. Каковы различия между режимом с разделением времени и многозадачностью?

3.2. Архитектура операционных систем

Для понимания архитектуры типичной операционной системы полезно сначала получить представление о полном спектре программного обеспечения, используемого в стандартной компьютерной системе, и только после перейти к знакомству с самой операционной системой.

Обзор программного обеспечения

Мы начнем наш обзор программного обеспечения, которое можно обнаружить в типичной компьютерной системе, с представления общей схемы его классификации. В подобных классификациях близкие элементы программного обеспечения зачастую помещаются в различные классы, подобно тому как введение часовых поясов заставляет близких соседей устанавливать свои часы с разницей в час, хотя моменты заката и восхода у них почти совпадают. Более того, в случае с классификацией программного обеспечения динамичность самого предмета и отсутствие признанных авторитетов в этой области часто имеют следствием противоречивость используемой терминологии. Например, в операционной системе Windows от Microsoft имеются группы так называемых "вспомогательных программ" и "средств администрирования", содержащих по нашей классификации как программы из класса прикладных, так и программы из класса утилит. Поэтому приводимую ниже классификацию следует рассматривать скорее как средство, дающее некоторую точку опоры в сложном предмете, а не как констатацию всеми признанного факта.

Первым делом разделим программное обеспечение на две общие категории: **прикладное программное обеспечение** и **системное программное обеспечение** (рис. 3.3). Прикладное программное обеспечение включает программы, предназначенные для решения задач, вытекающих из специфических особенностей использования данной машины. Машина, используемая при инвентаризации в промышленной компании, будет иметь набор прикладных программ, существенно отличающийся от того, который будет иметь машина, используемая в работе инженером-электриком. Примером прикладного программного обеспечения являются электронные таблицы, системы баз данных, настольные издательские системы, системы разработки программ и игры.

В отличие от прикладного программного обеспечения системное программное обеспечение выполняет задачи, общие для всех вычислительных систем в целом. Фактически системное программное обеспечение формирует среду, в которой функционирует прикладное программное обеспечение, аналогично тому, как государственная инфраструктура (правительство, финансовые институты, государственные учреждения, дороги и т.д.) создает фундамент, на котором ее граждане основывают свой индивидуальный стиль жизни.

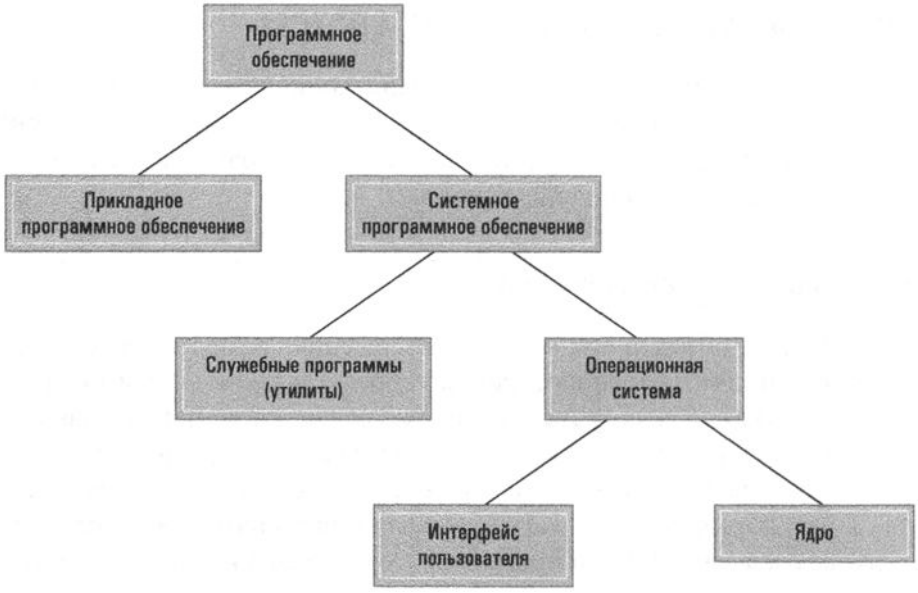


Рис. 3.3. Классификация программного обеспечения

Внутри класса системного программного обеспечения также есть две категории: одна — собственно операционная система, другая — элементы программного обеспечения, объединяемые понятием **обслуживающие программы** или **утилиты**. Большую часть установленных в системе обслуживающих программ составляют программы, предназначенные для выполнения действий, необходимых для успешного функционирования компьютера, но еще не включенные в операционную систему. В некотором смысле обслуживающие программы объединяют элементы программного обеспечения, расширяющие (или, возможно, настраивающие) возможности операционной системы. Например, обычно операционная система сама по себе не предоставляет средств форматирования диска или копирования файлов с жесткого диска на флеш-накопитель, поэтому данные функции обеспечиваются обслуживающими программами. К другим видам обслуживающих программ относятся программы сжатия и распаковки данных, программы для воспроизведения контента мультимедиа, программное обеспечение для осуществления сетевых соединений.

Предоставление определенных функциональных возможностей с помощью обслуживающих программ упрощает настройку системного программного обеспечения в соответствии с требованиями конкретной установки в сравнении с тем случаем, когда они являются частью собственно операционной системы. Действительно, вовсе не являются исключением компании или независимые пользователи, модифицирующие или расширяющие возможности утилит, поставляемых вместе с операционной системой их компьютеров.

К сожалению, различие между прикладным и обслуживающим программным обеспечением весьма условно. С нашей точки зрения, различие заключается в том, является ли данный пакет частью инфраструктуры программного обеспечения. Таким образом, новое приложение может превратиться в утилиту, если оно становится одной из основных сервисных программ. Будучи на стадии исследовательского проекта, программное обеспечение для общения через Интернет рассматривалось как прикладное программное обеспечение, тогда как сегодня такие инструменты имеют основополагающее значение для большинства компьютеров и, следовательно, должны классифицироваться как обслуживающие программы.

Linux

Для энтузиастов, желающих поэкспериментировать с внутренними компонентами операционной системы, существует ОС Linux. Эта операционная система изначально была разработана Линусом Торвальдсом (Linus Torvalds) еще в то время, когда он был студентом университета в городе Хельсинки. Это некоммерческий продукт, и потому данную операционную систему можно получить бесплатно вместе с документацией и исходным текстом программ (подробности — в главе 6). Благодаря свободе доступа к предоставляемому исходному коду она стала весьма популярной среди тех, для кого компьютер — хобби, а также среди студентов, изучающих операционные системы, и программистов вообще. Более того, ОС Linux признана в мире как одна из наиболее надежных операционных систем, существующих на сегодняшний день. По этой причине несколько компаний в настоящее время предлагают пакетные и рыночные версии Linux в легко доступной форме, и эти продукты успешно конкурируют на рынке с общепризнанными коммерческими операционными системами. Дополнительные сведения о Linux можно найти на сайте `http: //www.linux.org`.

Различие между обслуживающим программным обеспечением и операционной системой является довольно расплывчатым. В частности, антимонопольное законодательство в США и Западной Европе строилось на вопросах относительно того, являются ли такие элементы, как браузеры и медиаплееры, обязательными компонентами операционных систем Microsoft, или это утилиты, которые Microsoft включила только из соображений повышения конкурентоспособности.

Компоненты операционной системы

Теперь мы поговорим о тех компонентах, которые, собственно, и образуют операционную систему. Прежде всего, для выполнения действий, запрошенных пользователем компьютера, операционная система должна иметь возможность общаться с этим пользователем. Ту часть ОС, которая обеспечивает интерфейс операционной системы с пользователями, часто называют **интерфейсом пользователя**. Более ранние версии интерфейса пользователя, носившие название **оболочка**, обеспечивали взаимодействие пользователя с ОС посредством текстового интерфейса с использованием клавиатуры и экрана монитора. Более современные операционные системы решают эту задачу посредством **графического пользовательского интерфейса** (GUI — *Graphical User Interface*) в котором различные объекты, которыми манипулирует пользователь, например файлы и программы, представлены на экране монитора графически, в виде небольших рисунков или пиктограмм. Такой подход позволяет пользователям вводить команды, воспользовавшись для этого одним из нескольких доступных устройств ввода. Так, компьютерную мышь с двумя или более кнопками можно использовать для выбора щелчком или перетаскивания объектов на экране. Вместо мыши могут использоваться и другие указательные устройства различного назначения, например, трекболы или же стилусы, которые часто используются художниками для рисования и в некоторых случаях необходимы для работы с компактными графическими устройствами. В последние десятилетия очень широкое распространение получили устройства с высококачественными сенсорными экранами, позволяющими манипулировать отображаемыми на них объектами непосредственно пальцем. Более того, хотя в современных графических интерфейсах используются пока лишь двухмерные проекционные системы изображений, объектами текущих исследований ученых и инженеров являются трехмерные интерфейсы, предоставляющие пользователям возможность общаться с компьютерами с помощью *трехмерных* проекционных систем, тактильных сенсорных устройств и систем воспроизведения объемного звука.

Хотя интерфейс пользователя в операционной системе играет важную роль в определении доступной на данной машине функциональности, он, тем не менее, является всего лишь посредником между пользователем и сердцем самой операционной системы (рис. 3.4). Различие между интерфейсом пользователя и внутренними элементами операционной системы подчеркивается тем фактом, что некоторые операционные системы разрешают пользователю выбрать наиболее удобный для него тип интерфейса. Например, пользователи операционной системы UNIX могут выбрать одну из оболочек, включая Bourne, C или Korn, либо графический интерфейс пользователя под названием “X11”. Ранние

версии Microsoft Windows также представляли собой всего лишь приложения графического интерфейса пользователя, которые можно было загрузить из командной строки операционной системы MS-DOS. Программу командной оболочки DOS `cmd.exe` до сих пор можно найти как одну из утилит даже в последних версиях Windows, хотя этот интерфейс практически никогда на используется рядовыми пользователями. Аналогичным образом в ОС X от Apple сохранилась утилита командной оболочки `Terminal`, указывающая на то, что дальним предком этой системы была ОС UNIX.

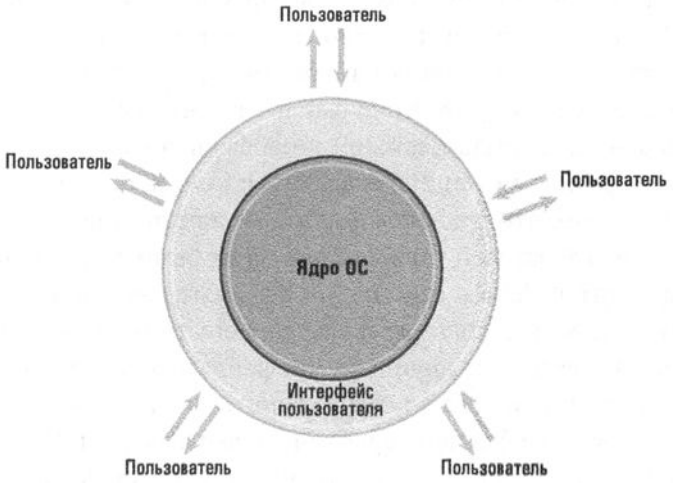


Рис. 3.4. Графический интерфейс выступает посредником между пользователями и ядром операционной системы

Важным компонентом современных графических оболочек является **программа управления окнами**, которая распределяет отдельные блоки пространства экрана, называемые окнами, и отслеживает, какое приложение ассоциируется с каждым из этих окон. Когда приложение намеревается отобразить что-нибудь на экране, оно сообщает об этом программе управления окнами, и она размещает предоставленное изображение в окне, соответствующем данному приложению. В свою очередь, когда пользователь нажимает кнопку мыши, именно программа управления окнами определяет положение указателя мыши на экране и уведомляет соответствующее приложение об этом действии пользователя. Программа управления окнами также отвечает за то, что обобщенно называют "стилем" графического интерфейса, и большинство таких программ предлагают широкий диапазон возможностей его настройки. А вот пользователи ОС Linux даже имеют возможность *выбрать* желаемую программу управления окнами, и наиболее популярными вариантами в этом случае являются KDE и Gnome.

В отличие от интерфейса пользователя операционной системы ее внутренняя часть обычно называется **ядром**, которое включает компоненты программного обеспечения, выполняющие основные функции, необходимые для управления работой всего оборудования компьютера. Одним из этих компонентов является **программа управления файлами**, в задачу которой входит координация использования устройств массовой памяти машины. Точнее говоря, эта программа поддерживает записи обо всех файлах, содержащихся в массовой памяти, включая информацию о том, где каждый из файлов находится, каким пользователям разрешен доступ к различным файлам и какой объем массовой памяти может быть использован для записи новых и расширения уже имеющихся файлов. Эти записи хранятся на каждом отдельном носителе внешней памяти, содержащем соответствующие файлы, так что каждый раз, когда этот носитель устанавливается пользователем и переводится в режим готовности к работе, диспетчер файлов может считывать их с него и, соответственно, получать информацию о том, что хранится на этом конкретном носителе.

Для удобства пользователей большинство программ управления файлами разрешает объединять файлы в группы, называемые **каталогами** или **папками**. Такой подход позволяет пользователям размещать свои файлы так, как им это удобно, помещая связанные друг с другом файлы в один каталог. Более того, каталоги могут содержать в себе другие каталоги, называемые подкаталогами, что позволяет создавать из файлов иерархические структуры. Например, пользователь может создать каталог `Записи`, который будет включать подкаталоги `Финансы`, `Медицина` и `Хозяйство`. В каждом подкаталоге будут размещаться файлы, относящиеся к соответствующей категории. (Пользователи операционной системы Windows могут попросить диспетчер файлов графически представить текущую конфигурацию доступных им папок, запустив на выполнение программу-утилиту `Проводник` (Windows Explorer).)

Цепочка, описывающая иерархию вложения каталогов внутрь других каталогов, называется **путем доступа**. Пути доступа чаще всего представляются посредством последовательного перечисления названий каталогов в соответствии с иерархией их вложения друг в друга, при этом названия каталогов разделяются *символом косой черты* — слешем "/". Например, запись `животные/доисторические/динозавры` представляет путь доступа, начинающийся с каталога `животные`, в который вложен подкаталог `доисторические`, в котором, в свою очередь, находится подкаталог `динозавры`. (В случае операционной системы Windows при записи пути доступа вместо символа косой черты используется *символ обратной косой черты* "\", т.е. тот же путь доступа в этом случае будет выглядеть как `животные\доисторические\динозавры`.)

Любой доступ к файлу со стороны других компонентов программного обеспечения предоставляется и контролируется программой управления файлами.

Процедура получения доступа к файлу начинается с запроса к программе управления файлами (этот запрос выполняется в рамках процедуры открытия файла). Если программа управления файлами разрешает доступ, то она предоставляет информацию, необходимую для поиска файла и работы с ним.

Другой компонент ядра операционной системы представляет собой набор **драйверов устройств**, т.е. элементов программного обеспечения, взаимодействующих с контроллерами устройств (или же непосредственно с устройствами) в целях выполнения различных операций в периферийных устройствах, подключенных к машине. Каждый драйвер устройства специально разрабатывается для конкретного типа устройства (например, принтера, дисковода или монитора). Он преобразует поступающие запросы в последовательность команд выполнения отдельных физических операций, которые необходимо выполнить устройству, связанному с этим драйвером. Например, драйвер устройства для принтера содержит программное обеспечение для считывания и декодирования слова состояния данного конкретного принтера, а также всех других данных, используемых в процедуре подтверждения связи. Следовательно, другим программным компонентам нет необходимости иметь дело с подобными техническими деталями, чтобы вывести что-либо на печать. Вместо этого при необходимости вывести данные на принтер другие компоненты могут просто полагаться на программное обеспечение драйвера данного устройства, — пусть этот драйвер сам позаботится обо всех необходимых деталях. В результате разработка других элементов программного обеспечения может вестись независимо от специфических особенностей конкретных устройств. Все это позволяет создать обобщенную операционную систему, которая будет легко настраиваться на использование любых периферийных устройств с помощью простой процедуры установки соответствующих драйверов.

Еще один компонент ядра операционной системы — **программа управления памятью**, которая решает задачу координации использования машиной ее основной памяти[1]. В среде, где машина выполняет только одно задание в каждый момент времени, обязанности этой программы минимальны. В этом случае необходимая текущему заданию программа помещается в основную память, выполняется, а затем заменяется программой для последующего задания. Однако в многопользовательской среде или в среде с многими задачами, когда машина должна обрабатывать множество запросов, поступающих в одно и то же время, у программы управления памятью появляются обширные обязанности. В этой ситуации в основной памяти одновременно должно находиться множество программ и блоков данных. Следовательно, программа управления памятью должна отыскать и выделить необходимые области памяти для

[1] Доступную пользователям часть основной памяти машины также часто называют оперативной памятью, что указывает на ее высокое быстродействие в сравнении с внешней памятью компьютера.

удовлетворения возникающих потребностей, а также следить за тем, чтобы действия каждой программы были ограничены исключительно выделенным ей пространством памяти. Более того, поскольку в процессе работы программы не только начинают свое выполнение, но и заканчивают его, программа управления памятью также должна отслеживать информацию о тех участках памяти, которые уже освободились.

Задача программы управления памятью еще больше усложняется, когда требуемый объем основной памяти превышает ее реально существующий в компьютере объем. В этом случае программа управления памятью может создать иллюзию увеличения объема памяти путем перемещения программ и данных из основной памяти в массовую и обратно. Этот иллюзорный объем памяти называется **виртуальной памятью**. Предположим, что выполняемым программам требуется 8 Гбайт основной памяти, а в наличии имеется только 4 Гбайт. Чтобы создать иллюзию большего объема памяти, программа управления памятью прежде всего резервирует на жестком диске 4 Гбайт свободного места, где предполагается разместить содержимое тех 4 Гбайт основной памяти из 8 требуемых, которым не хватает места непосредственно в памяти компьютера. Далее этот участок памяти делится на фрагменты, называемые **страницами**, которые обычно имеют размер в несколько килобайтов, и в дальнейшем в каждую страницу записывается некоторый фрагмент данных из памяти, которому в ней не хватило места и который не предполагается использовать непосредственно в данный момент. В дальнейшем программа управления перемещает эти страницы назад и вперед между основной памятью и массовой памятью на диске, организуя этот процесс таким образом, чтобы все страницы, которые необходимы программам в данный конкретный момент времени, действительно присутствовали в 4 Гбайт основной памяти машины, а неиспользуемые размещались на диске. В результате компьютер получает возможность работать так, как если бы у него действительно было 8 Гбайт основной памяти.

Кроме упомянутых выше, в состав ядра операционной системы входят еще два дополнительных компонента — **планировщик** и **диспетчер**, речь о которых пойдет в следующем разделе. Сейчас мы только отметим, что в системах с разделением времени планировщик определяет последовательность выполняемых действий, а диспетчер контролирует распределение временных квантов для них.

Запуск операционной системы

Мы уже знаем, что в процессе функционирования операционная система формирует инфраструктуру, необходимую для других программных компонентов, но мы еще не рассматривали вопрос, как операционная система начи-

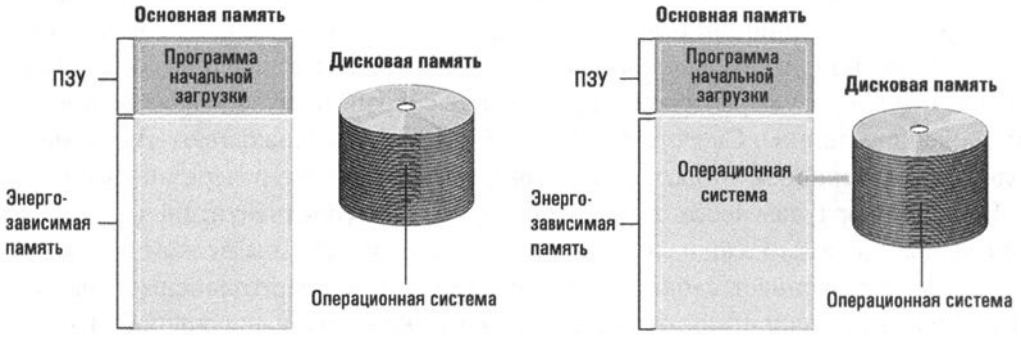
нает свою работу. Запуск операционной системы осуществляется с помощью процесса, называемого **начальной загрузкой** или просто **загрузкой системы** (*booting*), который выполняется при каждом включении машины. В этом процессе операционная система переносится с устройства массовой памяти (где она хранится постоянно) в основную память машины (которая, по существу, является пустой при включении машины). Чтобы сделать первый шаг к пониманию этого процесса и осознанию того, почему его необходимо выполнять на первом этапе включения машины, вновь обратимся к структуре ЦП компьютера.

Центральный процессор машины разработан таким образом, что при его включении выполняемая им программа каждый раз стартует с определенного, наперед заданного адреса. Следовательно, именно в этом месте основной памяти ЦП ожидает найти первую команду, которую требуется выполнить. Тогда теоретически все, что от нас требуется, — это обеспечить сохранение программы операционной системы в памяти, начиная с этого адреса. Однако по техническим причинам основная память компьютера, как правило, создается по таким технологиям, которые не позволяют сохранять ее содержимое после выключения машины. Следовательно, необходимо восстанавливать требуемое содержимое основной памяти всякий раз, когда компьютер перезапускается.

Короче говоря, нам необходимо, чтобы программа (предпочтительно — операционная система) находилась в основной памяти машины всякий раз, когда компьютер начинает свою работу, но содержимое энергозависимой памяти компьютера полностью утрачивается при каждом его выключении. Чтобы решить эту дилемму, небольшая часть основной памяти компьютера, где ЦП ожидает найти свою начальную программу, создается по особой, энергонезависимой технологии. Такая память называется **постоянной** (постоянное запоминающее устройство — **ПЗУ**), поскольку содержимое ее ячеек можно прочесть, но нельзя изменить. В качестве аналогии хранение последовательностей битов в ПЗУ можно представить себе как набор из крошечных выключателей: одни из них включены (это единицы), а другие выключены (это нули), хотя в действительности используемые технологии намного сложнее. Если говорить точнее, то большинство ПЗУ в современных компьютерах строится на использовании флеш-технологии (а это означает, что они не являются ПЗУ в строгом понимании этого слова, т.е. их содержимое изменить можно, но лишь при определенных условиях).

В компьютерах общего назначения программа, называемая **программой начальной загрузки** (*bootstrap*), постоянно хранится в ПЗУ машины. Именно эта программа автоматически запускается всякий раз при включении компьютера. Она предписывает ЦП считать данные из заранее определенного участка массовой памяти в энергозависимую основную память (рис. 3.5). Современные

программы начальной загрузки позволяют считывать в основную память код программ операционной системы из различных источников. Например, во встроенных системах, таких как смартфоны, операционная система извлекается из специальной (энергонезависимой) флеш-памяти, тогда как в случае небольших рабочих станций в вычислительных комплексах крупных компаний или университетов операционная система может быть скопирована с удаленного сервера по сети. Как только программы операционной системы будут помещены в основную память, программа начальной загрузки потребует от ЦП выполнить команду перехода в данную область памяти. В результате стартуют программы ядра, и операционная система начинает контролировать дальнейшую деятельность машины. Общий процесс выполнения программы начальной загрузки и последующего запуска операционной системы называют **загрузкой** компьютера.



Этап 1. Машина начинает выполнять программу начальной загрузки, находящуюся в постоянной памяти. Операционная система находится в массовой (внешней) памяти

Этап 2. Программа начальной загрузки выдает указание поместить операционную систему в основную память, а затем передает ей управление

Рис. 3.5. Процесс начальной загрузки

Вы можете спросить, а почему не использовать в компьютерах ПЗУ такого объема, чтобы в него можно было записать всю операционную систему и считывать что-нибудь с устройств массовой памяти уже не требовалось. Хотя такой подход вполне допустим для встраиваемых устройств с небольшими по размеру операционными системами, выделение больших блоков основной памяти на энергонезависимое хранилище в компьютерах общего назначения оказывается неэффективным при текущем уровне развития технологий. Более того, сами операционные системы часто подвергаются различным обновлениям из соображений обеспечения безопасности и включения в них поддержки новых и улучшенных версий драйверов устройств для новейшего оборудования. Хотя вполне возможно обновить операционную систему и программу начальной загрузки, записанные в ПЗУ (эта процедура называется **обновлением прошивки**

Встроенное программное обеспечение (firmware)

Помимо программы начальной загрузки, в ПЗУ персонального компьютера хранится большой набор подпрограмм, обеспечивающих выполнение необходимых процедур ввода-вывода, таких как получение данных с клавиатуры, отображение сообщений на экране дисплея, считывание данных с устройств массовой памяти. Записанное в ПЗУ, таком как флеш-память, это программное обеспечение не является навсегда прошитым в кремнии микросхем (как аппаратная часть машины), но при этом не является и столь легко изменяемым, как остальная часть программного обеспечения, сохраняемого на устройствах массовой памяти. Для обозначения этой промежуточной группы в отношении возможности изменения был выбран термин прошивка (firmware). Подпрограммы прошивки используются в начале загрузки операционной системы, пока операционная система не получит возможности взять управление доступом к устройствам на себя. Так, они могут использоваться для взаимодействия с пользователем еще до того, как собственно начнется процедура загрузки операционной системы, а также выводить сообщения об ошибках, возникших в процессе выполнения загрузки. Широко распространенные системы прошивок включают BIOS (Basic Input/Output System — базовая система ввода-вывода), долгое время использовавшуюся на ПК, более новую EFI (Extensible Firmware Interface — интерфейс расширяемой прошивки), Sun's Open Firmware (теперь это продукт компании Oracle) и CFE (Common Firmware Environment — общая среда прошивки), используемую в многочисленных встраиваемых устройствах.

(**firmware**) технологические ограничения делают внешнюю массовую память наиболее выгодным выбором для хранения программ операционной системы в случае традиционных компьютеров общего назначения.

В заключение следует подчеркнуть, что понимание процесса первоначальной загрузки, равно как и четкое представление о различиях между операционной системой, служебными программами-утилитами и прикладным программным обеспечением, позволяет глубже разобраться в общей методологии, согласно которой работает большинство компьютерных систем общего назначения. Когда подобные машины включают в работу, программа начальной загрузки загружает в память машины и активирует операционную систему. Далее пользователь может вводить запросы к операционной системе на выполнение любых служебных или прикладных программ. По окончании работы утилиты или прикладной программы диалог пользователя с операционной системой возобновляется и он получает возможность вводить следующие запросы. Таким образом, можно сделать вывод, что обучение работе с такой системой представляет собой двухуровневый процесс. Помимо изучения необходимых

деталей по работе с определенной утилитой или приложением, каждый пользователь должен получить необходимые знания о взаимодействии с самой операционной системой, чтобы научиться находить нужные приложения и утилиты из числа доступных и выполнять их в нужной последовательности.

3.2. Вопросы и упражнения

1. Перечислите компоненты типичной операционной системы и одной фразой охарактеризуйте роль каждого из них.
2. В чем заключаются различия между прикладным программным обеспечением и обслуживающими программами-утилитами?
3. Что такое виртуальная память?
4. Опишите процедуру начальной загрузки.

3.3. Координация действий машины

В этом разделе мы рассмотрим, как операционная система координирует выполнение прикладных программ, утилит и собственных программных элементов. Начнем обсуждение с введения понятия процесса.

Понятие процесса

Одной из наиболее фундаментальных концепций в современных операционных системах является разграничение между самой программой и деятельностью, связанной с ее выполнением. Первая представляет собой статический набор инструкций, в то время как второе — это динамическая деятельность, свойства которой меняются во времени. (Это различие аналогично различиям между нотами некоторого музыкального произведения, лежащими на полке, и музыкантом, исполняющим это произведение, следуя его нотной записи.) Деятельность, связанная с выполнением программы под управлением операционной системы, получила название **процесс**. Связанное с процессом текущее состояние работы, называют **состоянием процесса**. Это состояние включает текущую позицию выполняемой программы (значение счетчика адреса), а также значения прочих регистров центрального процессора и тех ячеек памяти, к которым производится обращение. Говоря упрощенно, состояние процесса — это моментальный снимок состояния машины в определенный момент времени. В различные моменты выполнения программы (процесса) будут получаться различные моментальные снимки (состояния процесса).

В отличии от музыканта, который в каждый момент обычно исполняет одно произведение, в типичном компьютере, работающем в режимах разделения времени и мультизадачности, одновременно выполняется сразу много процессов, которые конкурируют за право доступа к различным ресурсам компьютера. На операционную систему возложена обязанность управлять этими процессами таким образом, чтобы каждый процесс получал необходимые ему ресурсы (доступ к периферийным устройствам, необходимый объем основной памяти, доступ к файлам и доступ к центральному процессору), и при этом независимые процессы не оказывали влияния друг на друга, а те процессы, которым необходимо обмениваться информацией, имели возможность выполнить этот обмен.

Управление процессами

Задачи, связанные с координацией выполнения процессов, решаются планировщиком и диспетчером, входящими в состав ядра операционной системы. Планировщик ведет пул записей о процессах, присутствующих в вычислительной системе, вводит в него сведения о новых процессах и удаляет информацию о завершившихся. Следовательно, когда пользователь выдает запрос на выполнение некоторого приложения, именно планировщик добавляет в пул текущих процессов сведения о выполнении этого приложения.

Для отслеживания состояния всех процессов планировщик организует в основной памяти блок информации, называемый **таблицей процессов**. Каждый раз, когда машине дается новое задание, планировщик создает процесс для этого задания посредством занесения новой записи в таблицу процессов. Эта запись содержит сведения об объеме выделенной процессу памяти (эта информация поступает от модуля управления памятью), о присвоенном ему приоритете, а также о том, в каком состоянии находится процесс: готовности или ожидания. Процесс находится в **состоянии готовности**, если его развитие может продолжаться, и переводится **в состояние ожидания**, если его развитие приостанавливается до тех пор, пока не произойдут некоторые внешние события, например завершится процедура доступа к массовой памяти, на клавиатуре будет нажата некоторая клавиша или поступит сообщение от другого процесса.

Диспетчер — это компонент ядра, отвечающий за выполнение запланированных процессов. В системе с многими пользователями и мультизадачностью эта проблема решается посредством механизма **разделения времени**, т.е. разбиения времени процессора на короткие интервалы, называемые **квантами** (обычно измеряемые милли- или микросекундами). По истечении этого времени происходит принудительное переключение центрального процессора от одного процесса к другому; так что каждому процессу предоставляется возможность непрерывного выполнения лишь в течение одного кванта времени

(рис. 3.6). Процедура смены одного процесса другим называется **переключением процессов** (или иначе **переключением контекста**).

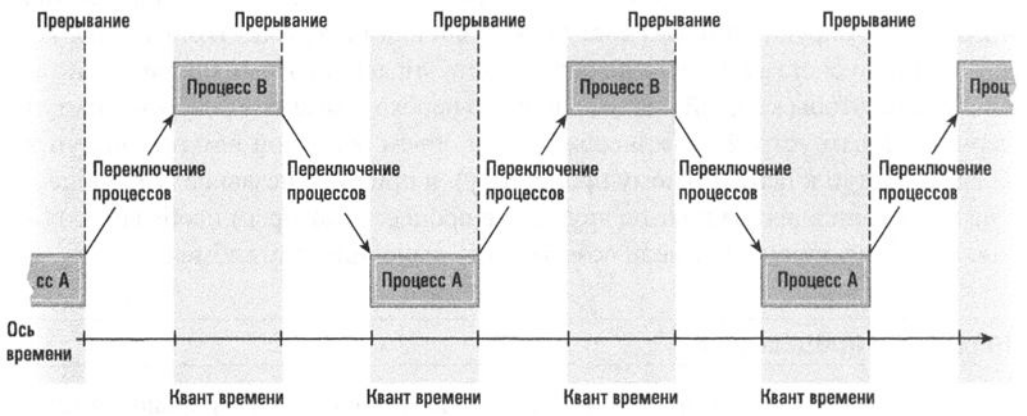


Рис. 3.6. Разделение времени между процессами А и В

Каждый раз, когда процессу предоставляется очередной квант времени, диспетчер инициирует схему таймера, подготавливая его к измерению продолжительности следующего кванта. По окончании установленного кванта схема таймера генерирует сигнал, называемый **прерыванием**. Центральный процессор реагирует на этот сигнал так же, как и человек, которого останавливают во время выполнения определенного задания. Человек прекращает свою работу, записывает текущее состояние задачи (чтобы иметь возможность продолжить ее выполнение позднее) и обращает внимание на то, что его отвлекло. При получении сигнала прерывания центральный процессор завершает текущий машинный цикл, сохраняет информацию о состоянии текущего процесса (подробнее мы обсудим это чуть ниже) и начинает выполнять программу, называемую **обработчиком прерываний**, помещенную в заранее определенное место в основной памяти. Обработчик прерываний является той частью диспетчера, которая определяет, как диспетчер должен реагировать на поступивший сигнал прерывания.

Таким образом, результатом поступления сигнала прерывания является приостановка текущего процесса и передача управления диспетчеру. В этот момент диспетчер выбирает из таблицы процессов процесс с наивысшим приоритетом из числа тех, которые находятся в состоянии готовности, и заново инициализирует схему таймера, отмеряющего квант времени, после чего разрешает выбранному процессу использовать этот новый квант.

Главным условием успешной работы системы с разделением времени является ее способность остановить, а затем повторно запустить процесс. Если вас прерывают во время чтения книги, то возможность продолжить чтение

позднее зависит от вашей способности вспомнить, на чем вы остановились и что прочли до этого момента. Короче говоря, вы должны иметь возможность воссоздать ситуацию такой, какой она была непосредственно в момент прерывания.

В отношении процесса то, что должно быть восстановлено, — это его состояние. Напомним, что состояние процесса включает в себя значение счетчика адреса, а также содержимое регистров ЦП и ячеек памяти, к которым выполняется обращение. Машины, разработанные для систем с разделением времени, включают средства, позволяющие автоматически выполнить сохранение этой информации как реакцию центрального процессора на сигнал прерывания. Кроме того, машинный язык таких процессоров обычно включает специальные команды для перезагрузки ранее сохраненной информации о состоянии. Подобные функциональные возможности вычислительных машин упрощают задачу диспетчера по переключению процессов и служат примером того, как потребности современных операционных систем оказывают влияние на разработку компьютеров.

В завершение отметим, что механизм разделения времени был разработан с целью повышения общей эффективности работы вычислительных машин. Подобный подход выглядит несколько противоречивым, ведь сама процедура переключения процессов, обязательная для системы с разделением времени, создает в ней дополнительную вычислительную нагрузку. Однако без механизма разделения времени каждый процесс будет беспрепятственно выполняться до своего завершения и только после этого следующий процесс сможет начать свое выполнение, а это означает, что пока любой процесс будет ожидать завершения операции обращения к внешнему устройству либо ожидать ответа пользователя, все остальные процессы будут простаивать, бесполезно теряя время. Механизм разделения времени позволяет передать это теряемое время другому процессу. Например, если процесс выдает запрос на выполнение операции ввода-вывода, скажем запрос на получение данных с диска, планировщик обновляет таблицу процессов, отразив в ней переход данного процесса в состояние ожидания. В свою очередь, диспетчер прекратит выделение новых квантов времени этому процессу. Позднее (возможно, через несколько сотен миллисекунд), когда контроллер сообщит о том, что поступивший запрос на операцию ввода-вывода уже выполнен, планировщик вновь отметит данный процесс как готовый к выполнению, и этот процесс сможет конкурировать за получение очередного кванта времени ЦП. Короче говоря, за время выполнения операций ввода-вывода одного задания будет достигнут прогресс в выполнении других заданий, а значит, вся совокупность заданий будет выполнена за меньшее время, чем при строго последовательном их выполнении.

Прерывания

Использование механизма прерываний для завершения квантов времени, как было описано выше, — это только один из вариантов применения системы прерываний в компьютерах. Существует немало других ситуаций, в которых может генерироваться сигнал прерывания, для каждого из которых имеется собственная подпрограмма обработки. В действительности механизм прерываний является важнейшим инструментом координации действий компьютера в соответствии с событиями в его внутренней среде. Например, как щелчок мышью, так и нажатие клавиши клавиатуры генерирует сигнал прерывания, вынуждающий центральный процессор свернуть выполняемую им в данный момент задачу и переключиться на обработку этого прерывания.

С целью управления задачей распознавания и реагирования на поступающие прерывания разным сигналам прерывания назначается различный приоритет, и это позволяет более важные задачи обрабатывать в первую очередь. Самый высокий приоритет прерывания обычно связывается с отключением питания машины. Подобный сигнал прерывания генерируется, если в системе питания компьютера происходят неожиданные нарушения. Подпрограмма обработки этого прерывания направляет в центральный процессор целую серию указаний по аварийному наведению порядка в его "домашнем хозяйстве" за те считанные миллисекунды, пока уровень напряжения питания не упадет ниже допустимого для нормальной работы процессора.

3.3. Вопросы и упражнения

1. Кратко опишите различия между программой и процессом.
2. Коротко опишите действия, предпринимаемые центральным процессором при возникновении прерывания.
3. Каким образом в системе с разделением времени процесс с высоким приоритетом может выполняться быстрее других?
4. Пусть в системе с разделением времени каждый квант времени равен 50 мс, а на каждое переключение между процессами затрачивается по крайней мере микросекунда. Сколько процессов может обслужить машина за одну секунду?
5. Если каждый процесс в машине с характеристиками, указанными в предыдущем вопросе, использует свой квант времени полностью, какая часть машинного времени используется для реального выполнения процессов? Какой будет эта часть в случае, если каждый процесс выполняет запрос на ввод-вывод по истечении только одной микросекунды от начала каждого выделенного ему кванта времени?

3.4. Организация конкуренции между процессами

Важной задачей операционной системы является распределение машинных ресурсов между процессами в системе. Здесь понятие ресурсов используется в широком смысле и включает как периферийные устройства, так и ресурсы самой машины. Программа управления файлами отвечает как за организацию доступа к уже существующим файлам, так и за распределение места в массовой памяти при создании новых файлов. Программа управления памятью распределяет свободные области памяти. Планировщик распределяет место в таблице процессов, а диспетчер — кванты времени. Как и в случае многих других проблем в компьютерных системах, задача распределения, на первый взгляд, может показаться несложной. Однако, если присмотреться, можно обнаружить несколько проблем, способных привести к сбоям в недостаточно тщательно разработанной системе. Не забывайте, что сама по себе машина не думает, а лишь следует имеющимся инструкциям. Поэтому для создания надежной операционной системы следует разработать алгоритмы, учитывающие все существующие аспекты выполняемой работы, какими бы незначительными они ни казались.

Семафоры

Представим себе операционную систему с разделением времени, управляющую работой машины с одним принтером. Если процессу требуется напечатать полученные результаты, он должен запросить у операционной системы доступ к драйверу принтера. Получив запрос, операционная система должна решить, следует ли его удовлетворять, исходя из того, используется ли принтер в данный момент другим процессом. Если принтер свободен, операционная система может удовлетворить запрос и разрешить процессу продолжить свою работу; в противном случае она должна ответить на запрос отказом и, возможно, перевести процесс в состояние ожидания, которое будет продолжаться до тех пор, пока принтер вновь не станет доступен. Так или иначе, если доступ к принтеру будет предоставлен двум процессам одновременно, полученные результаты будут неудовлетворительны для каждого из них.

Чтобы управлять доступом к принтеру, операционная система должна следить, занят ли принтер в данный момент. Одним из возможных решений может быть использование флажка, который в данном контексте будет представлять собой бит памяти, состояние которого будет означать *занят* или *свободен*, а не просто `1` или `0`. Если значение флажка — свободен, то это означает, что принтер доступен, а если занят, это указывает на то, что принтер уже предоставлен некоторому процессу. На первый взгляд, использование такого подхода не должно вызывать каких-либо непредвиденных проблем. Операционная система

Программа Диспетчер задач в Microsoft Windows

Некоторое представление о внутренней деятельности операционной системы Microsoft Windows можно получить, если запустить на выполнение ее утилиту под названием Диспетчер задач (Task Manager); для этого одновременно нажмите клавиши <Ctrl+Alt+Delete> и выберите в меню команду Запустить диспетчер задач. В открывшемся окне этой утилиты перейдите на вкладку Процессы (Processes) — и вы увидите реальную таблицу процессов операционной системы. Вы даже сможете выполнить простой эксперимент. Запомните текущее состояние этой таблицы, а затем запустите какую-нибудь программу. (У вас может вызвать удивление тот факт, что в этой таблице уже находится так много процессов. Это нормально, большинство из них отвечает за выполнение базовых функций операционной системы.) А теперь запустите новую программу и убедитесь, что в эту таблицу был добавлен соответствующий процесс. Здесь же вы можете узнать, сколько основной памяти было выделено этому процессу.

просто проверяет состояние флажка при каждом поступлении запроса на предоставление доступа к принтеру. Если флажок находится в состоянии свободен, то запрос удовлетворяется и операционная система изменяет значение флажка на занят. Если текущее значение флажка — занят, то операционная система переводит процесс, направивший запрос, в состояние ожидания. Каждый раз, когда очередной процесс заканчивает работу с принтером, операционная система или предоставляет принтер ожидающему процессу, или, если ожидающих процессов нет, просто изменяет значение флажка на свободен.

Хотя это решение выглядит вполне удовлетворительным, оно заключает в себе проблему. Задача проверки и, возможно, установки состояния флажка решается операционной системой за *несколько* машинных циклов. (Значение флажка должно быть извлечено из основной памяти, обработано внутри центрального процессора и вновь записано в память.) Следовательно, вполне возможно, что выполнение этой задачи будет прервано после того, как будет установлено, что флажок находится в состоянии свободен, но еще до того, как его состояние будет изменено на занят. В результате возможно возникновение следующей последовательности событий.

Предположим, что в данный момент времени принтер доступен и некоторый процесс запрашивает его использование. Система проверяет соответствующий флажок и обнаруживает, что он находится в состоянии свободен, т.е. принтер доступен. Однако в этот момент выполнение процесса прерывается и другой процесс начинает использовать выделенный ему квант времени. Он также запрашивает обращение к принтеру. Состояние флажка еще раз проверяется и обнаруживается, что он по-прежнему имеет значение свободен, так как

выполнение предыдущего процесса было прервано прежде, чем операционная система успела изменить значение флажка. В результате операционная система разрешает использовать принтер и второму процессу. Через некоторое время первый процесс возобновляет свою работу с того места, где он был приостановлен, т.е. непосредственно после того, как операционная система обнаружила, что флажок имеет значение свободен. В результате операционная система разрешит первому процессу продолжить работу и предоставит ему доступ к принтеру. В итоге два процесса одновременно будут использовать один и тот же принтер.

Решение проблемы состоит в том, что задача проверки и, возможно, установки значения флажка должна выполняться *без прерываний*. Одним из решений может быть использование команд запрета прерываний и команд разрешения прерываний, имеющихся в большинстве машинных языков. Если операционная система начнет процедуру проверки состояния флажка с команды, запрещающей выдачу прерываний в системе, и закончит ее командой, вновь разрешающей прерывания, то как только эта процедура будет начата, ее выполнение уже никогда не сможет быть прервано каким-либо другим процессом.

Другой подход к решению этой проблемы состоит в использовании команды **test-and-set** (проверить и установить), имеющейся во многих машинных языках. По этой команде центральному процессору предписывается считать значение флажка, проанализировать полученное значение, а затем при необходимости установить новое значение — и все это в пределах одной машинной команды. Преимущество этого варианта заключается в том, что центральный процессор, прежде чем проанализировать наличие прерывания, всегда завершает выполнение текущей команды, и выполнение задачи проверки и установки флажка не может быть прервано, если она реализована в виде одной команды.

Реализованный таким образом флажок называется **семафором**. Это напоминает железнодорожное сигнальное устройство, используемое для управления доступом к определенному участку пути. Фактически в системах программного обеспечения семафоры используются почти так же, как и на железнодорожных линиях. Участку пути, на котором может находиться только один состав, соответствует последовательность команд, которая может выполняться одновременно только одним процессом. Такая последовательность команд называется **критической областью**. Требование о том, что в любой заданный момент только один процесс может выполнять команды критической области, называют **взаимным исключением**. Типичным способом реализации взаимного исключения для некоторой критической области является защита этой области с помощью семафора. Чтобы войти в критическую область, процесс должен удостовериться, что семафор открыт, и затем изменить его значение на закрыт еще до того, как войдет в критическую область. При выходе из критической области процесс должен вновь открыть семафор. Если же при проверке окажется,

что семафор закрыт, процесс, предпринявший попытку зайти в критическую область, должен быть переведен в процесс ожидания, пока семафор не будет открыт.

Взаимная блокировка

Другой проблемой, которая может иметь место при распределении ресурсов, является **взаимная блокировка**. Это состояние, при котором дальнейшая работа двух или нескольких процессов взаимно заблокирована, поскольку каждый из них ожидает доступа к ресурсам, которые уже предоставлены другому. Например, один процесс может иметь доступ к принтеру, но ожидать доступа к накопителю на магнитных лентах, в то время как другой уже получил доступ к накопителю на магнитных лентах, но ожидает освобождения принтера. Другим примером является ситуация, когда процессы создают новые процессы (это действие называется **ветвление** (*forking*) на разговорном диалекте UNIX), предназначенные для выполнения подзадач. Если у планировщика уже нет свободного места в таблице процессов, а каждому из процессов в системе для завершения своей задачи необходимо создать дополнительный процесс, то ни один из процессов не сможет продолжить свою работу. Эта и подобные ситуации (рис. 3.9) могут серьезно нарушить работу системы.

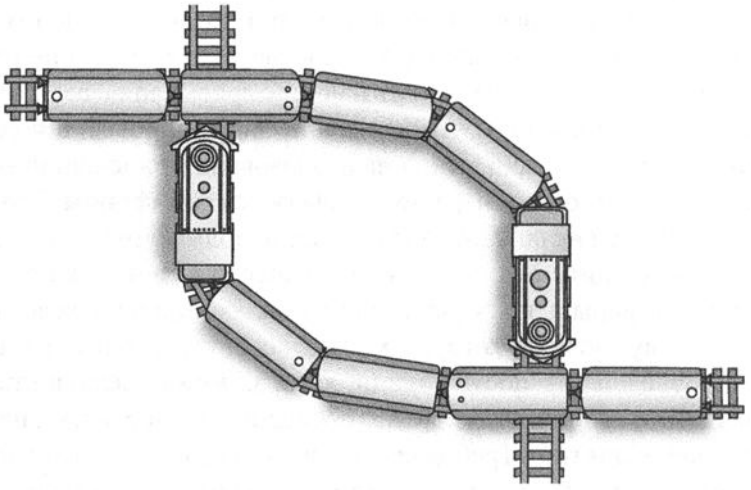


Рис. 3.7. Взаимная блокировка, возникшая в результате конкуренции за использование неразделяемых пересечений железнодорожных путей

Анализ причин появления взаимных блокировок показывает, что такие ситуации возможны только тогда, когда удовлетворены следующие три условия.

1. В системе имеет место конкуренция за использование неразделяемых ресурсов.
2. Ресурсы запрашиваются частями — процесс, уже получив некоторые ресурсы, продолжает запрашивать другие.
3. Предоставленный ресурс не может быть отобран принудительно.

Смысл выделения этих трех условий состоит в том, что проблема возникновения взаимной блокировки может быть решена посредством устранения любого из них. В общем случае методы, предназначенные для подавления третьего условия, относятся к категории схем обнаружения взаимных блокировок с последующей коррекцией. При данном подходе считается, что тупиковые ситуации возникают настолько редко, что не стоит тратить усилия на предотвращение этой проблемы. Наоборот, смысл этого метода состоит в обнаружении ситуации взаимной блокировки, когда она уже возникла, с последующим устранением ее посредством принудительного отбора некоторых из уже предоставленных ресурсов. Приведенный выше пример с заполненной до отказа таблицей процессов можно отнести к этому классу. Если взаимная блокировка по причине переполнения таблицы процессов все же возникнет, подпрограмма операционной системы (или, возможно, ее администратор, использующий свое право "суперпользователя") может удалить (технический термин — **убить**) некоторые из процессов. В результате освобождается место в таблице процессов и ситуация взаимной блокировки устраняется, после чего оставшиеся процессы смогут продолжить выполнение своих заданий.

Язык Python и операционные системы

Когда пользователь начинает выполнение сценария на языке Python, операционная система запускает новый процесс, предназначенный для выполнения этого сценария. Подобные сценарии обычно являются приложениями, хотя могут рассматриваться и как вспомогательное программное обеспечение, если они расширяют возможности системы или выполняют их настройку. Для выполнения предусмотренных в них действий сценарии на языке Python взаимодействуют с компонентами операционной системы, такими как программа управления файлами при чтении и записи файлов или графический интерфейс пользователя либо оболочка при необходимости организовать взаимодействие с пользователем. В модуле "`os`" языка Python можно найти необходимый набор предопределенных системно-независимых функций для получения доступа к общим функциям операционной системы, таким как создание новой подзадачи в среде Python или выполнение другой утилиты или прикладной программы.

Методики, нацеленные на устранение двух первых условий, относятся к схемам исключения тупиковых ситуаций. Одна из них, например, позволяет исключить второе условие за счет того, что каждый процесс должен запрашивать все необходимые ему ресурсы сразу. Другая, возможно, более утонченная методика, направлена на устранение первого условия, но не за счет простого запрета конкуренции, а за счет превращения неделимых ресурсов в разделяемые. Например, предположим, что рассматриваемым ресурсом является принтер и множество процессов запрашивают доступ к нему. Каждый раз, когда процесс запрашивает доступ к принтеру, система предоставляет ему этот доступ. Но вместо того чтобы подключить процесс непосредственно к драйверу принтера, операционная система подключает его к драйверу логического устройства, записывающего предназначенную для печати информацию в массовую память, вместо того чтобы отправить ее непосредственно на принтер. В результате каждый процесс выполняется нормальным образом, полагая, что он имеет доступ к принтеру. Позднее, когда принтер освободится, операционная система может переслать данные с диска на принтер. В этом случае операционная система придала неразделяемому ресурсу вид разделяемого, создав иллюзию наличия в системе более чем одного принтера. Подобная методика предварительного сохранения выводимых данных в целях ожидания подходящего момента для их действительного вывода получила название **спулинг** (*spooling*).

Мы представили здесь спулинг как метод, обеспечивающий нескольким процессам доступ к общему ресурсу, но такой подход может иметь множество вариаций. Например, программа управления файлами должна предоставлять доступ к одному и тому же файлу сразу нескольким процессам, если они только считывают данные этого файла. Однако, если нескольким процессам в одно и то же время потребуется изменить содержимое файла, это может привести к конфликтной ситуации. Поэтому программа управления файлами должна предоставлять доступ к файлам с учетом конкретных потребностей процессов, в каждый момент позволяя сразу нескольким процессам иметь доступ для чтения файла, но разрешая только одному процессу иметь доступ для записи в файл. Некоторые системы допускают разделение файла на части, в результате чего различные процессы получают возможность одновременно изменять различные части одного файла. Однако каждый из этих методов имеет свои нюансы, которые необходимо учитывать для получения надежной системы. Например, как проинформировать процессы, получившие доступ для чтения, о том, что процесс с доступом для записи изменяет файл?

Многоядерные операционные системы

Традиционные многопользовательские и многозадачные системы создают иллюзию одновременного выполнения многих процессов посредством быстрого переключения между квантами времени, — быстрее, чем человек способен это воспринять. Современные системы поддерживают многозадачность тем же самым способом, но в дополнение к нему не так давно появились многоядерные процессоры, действительно способные выполнять два, четыре и более процессов одновременно. В отличие от группы одноядерных компьютеров, работающих совместно, многоядерные машины содержат по нескольку независимых процессоров (их в этом случае называют *ядрами*), которые совместно используют периферийные устройства компьютера, его основную память и другие ресурсы. Для мультиядерной операционной системы это означает, что диспетчер и планировщик должны дополнительно принимать решение, какие процессы будут выполняться на каждом ядре. При наличии различных процессов, выполняющихся на разных ядрах, управление конкурирующими процессами становится еще более сложной задачей, поскольку блокировка прерываний на всех ядрах каждый раз, когда какому-то из процессов необходимо войти в критическую область, является очень неэффективным подходом. Сейчас ученые активно ведут научные исследования в областях компьютерных наук, связанных с построением механизмов операционных систем, более подходящих к условиям нового многоядерного мира.

3.4. Вопросы и упражнения

1. Предположим, что процессы A и B функционируют в системе с разделением времени и что каждый из них нуждается в одном и том же неделимом ресурсе на короткий период времени. (Например, каждый процесс может печатать ряд независимых коротких сообщений.) В этой ситуации каждый процесс может многократно запрашивать доступ к ресурсу, освобождать его, а затем вновь запрашивать доступ. Какой недостаток существует в схеме контроля доступа к этому ресурсу, построенной по приведенному ниже принципу?

   Исходно флажку присваивается значение 0. Если процесс A запрашивает ресурс и значение флажка равно 0, следует удовлетворить этот запрос. В противном случае необходимо перевести процесс A в состояние ожидания. Если процесс B запрашивает ресурс и флажок имеет значение 1, следует удовлетворить этот запрос. В противном случае необходимо перевести процесс B в состояние ожидания. Каждый раз, когда процесс A заканчивает работу с ресурсом, значение

флажка следует изменить на 1. Каждый раз, когда процесс В заканчивает работу с ресурсом, следует изменить значение флажка на 0.

2. Предположим, что дорога имеет двустороннее движение, которое переходит в одностороннее при прохождении через туннель. Для координации использования туннеля была применена следующая система сигналов.

   Когда машина въезжает в туннель с любого его конца, над обоими входами загорается красный свет. Когда автомобиль выезжает, свет гаснет. Если, когда машина подъезджает, красный свет включен, она ожидает, пока он не будет выключен, и только после этого въезжает в туннель.

   Какой недостаток есть в этой системе?

3. Предположим, что предложено несколько решений для устранения возможности взаимной блокировки при встрече двух автомобилей на мосту с односторонним движением. Определите, какое из трех условий взаимной блокировки, упомянутых выше в данном разделе, снимается каждым из следующих решений.

   а. Автомобиль не заезжает на мост до тех пор, пока он не освободится.

   б. Если на мосту встречаются две машины, одна из них едет назад.

   в. Добавляется вторая полоса движения.

4. Предположим, что каждый процесс в системе с разделением времени будет представлен точкой; дополнительно рисуется стрелка от одной точки к другой, если процесс, представленный первой точкой, ожидает освобождения (неделимого) ресурса, который используется вторым процессом. Математики называют этот чертеж **ориентированным графом**. Какая особенность в структуре ориентированного графа будет указывать на наличие в системе ситуации взаимной блокировки?

3.5. Безопасность

Поскольку операционная система следит за любыми действиями, происходящими в компьютере, вполне естественно будет возложить на нее ключевую роль и в обеспечении его безопасности. В широком смысле эта ответственность может проявлять себя во множестве различных форм, одной из которых

является надежность. Если изъян в работе программы управления файлами приводит к потере части файла, то нельзя считать, что этот файл был в достаточной степени защищен. Если ошибка диспетчера приводит к отказу в работе системы (иначе это называют системным сбоем), приведшему к потере результатов нескольких часов ввода данных с клавиатуры, мы можем обоснованно утверждать, что наша работа оказалась незащищенной. Следовательно, для безопасности компьютерной системы требуется хорошо спроектированная, надежная и заслуживающая доверия операционная система.

Высокая важность разработки надежного программного обеспечения характерна не только для операционных систем. Эта проблема распространяется на весь спектр создаваемого программного обеспечения и образует отдельную область компьютерных наук, получившую название *технология разработки программного обеспечения*, — подробно мы обсудим эту тему в главе 7. В этом же разделе мы сконцентрируемся только на тех аспектах безопасности, которые в большей степени связаны со спецификой операционных систем.

Атаки снаружи

Важной задачей, решаемой операционной системой, является защита ресурсов компьютера от доступа неавторизованных пользователей. В тех случаях, когда компьютер используется многими лицами, эта проблема обычно решается посредством создания **учетных записей** (*account*) для различных авторизованных пользователей. По своей сути учетная запись представляет собой регистрацию в операционной системе таких данных о пользователе, как его имя, пароль и перечень предоставленных ему привилегии. В дальнейшем операционная система может использовать эту информацию при выполнения процедуры **входа в систему** (login procedure), т.е некоторой последовательности транзакций, посредством которых пользователь устанавливает исходный контакт с операционной системой компьютера, что позволяет ей контролировать доступ к системе.

Учетные записи создаются ответственным лицом, известным как **администратор** (или **суперпользователь**) системы. Этот человек получает высокие привилегии в отношении доступа к операционной системе посредством отождествления себя как администратора (обычно с использованием имени и пароля) непосредственно в процессе входа в систему. Как только такой контакт будет установлен, администратор получает право изменять параметры настройки системы, вносить поправки в критические важные пакеты программ, корректировать наборы привилегий, предоставленные другим пользователям и выполнять те или иные прочие действия, необходимые ему в процессе сопровождения системы, которые запрещено выполнять обычным пользователям.

Пользуясь своим "высоким положением", администратор также может наблюдать за всем, что происходит в компьютерной системе, с целью обнаружения любого деструктивного поведения, как намеренного, так и случайного. Чтобы помочь ему в этом важном деле, было создано множество утилит и различных **программ аудита** системы, позволяющих фиксировать, а затем анализировать любую деятельность, происходящую в компьютерной системе. В частности, программы аудита могут выявлять поток попыток входа в систему с использованием неправильных паролей, сигнализирующий о том, что некий неавторизованный пользователь предпринимает попытки получить доступ к компьютеру. Программы аудита могут также выявлять такую *активность* в пределах учетной записи пользователя, которая не согласуется с предыдущим поведением этого пользователя, и это может указывать на то, что неавторизованный пользователь получил доступ к этой учетной записи. (Маловероятно, что пользователь, который обычно работал только с текстовым процессором и электронными таблицами, внезапно обратится к высокотехнологичному программному обеспечению или попытается выполнить утилиту из пакета служебных программ, использование которого выходит за пределы привилегий этого пользователя.)

Другой вид нарушений, которые программы аудита способны обнаружить, — это присутствие в компьютере программы-сниффера (или анализатора трафика), которая анализирует входящий и исходящий потоки данных компьютера, а затем пересылает потенциальному злоумышленнику интересующие его сведения. Самым известным примером является программа, имитирующая процедуру регистрации в операционной системе. Такая программа может использоваться для обмана авторизованных пользователей, полагающих, что они ведут диалог с операционной системой, и соответственно, предоставляющих программе-имитатору свои действительные имена пользователей и пароли, которые в результате становятся известны злоумышленнику.

При всех технических сложностях, связанных с безопасностью компьютеров, вызывает удивление тот факт, что одной из основных причин нарушений безопасности компьютерных систем является небрежность самих ее пользователей. Они часто выбирают пароли, которые очень легко подобрать (такие, как имена или даты), делятся своими паролями с друзьями, пренебрегают необходимостью смены паролей через определенные промежутки времени, подвергают носители массовой памяти опасности разрушения, передавая их туда и обратно между различными машинами, загружают в систему непроверенное программное обеспечение, которое потенциально может разрушить защиту всей компьютерной системы. Для решения подобных проблем большинство учреждений с большими компьютерными системами принимают специальный набор правил, в котором четко зафиксированы требования и обязанности пользователей, и строго следят за его соблюдением.

Безопасность паролей

Поскольку компьютеры превратились в место хранения большого количества личных и других ценных данных, чрезвычайно важной стала проблема защиты учетных записей надежными паролями. Хотя многие подключенные к сети системы поддерживают различные типы процедур удаленного входа в систему, злонамеренные личности все же имеют возможность создать и распространить программу, которая очень быстро попытается ввести множество всевозможных паролей во много раз быстрее, чем это способен сделать человек. Следовательно, защищенный пароль — это такой пароль, подобрать который будет непросто, а значит, и период времени, за который можно будет обнаружить, что атакующий начал процедуру подбора пароля посредством перебора всех возможных комбинаций, будет больше. По этой причине для многих учетных записей сейчас установлена минимально допустимая длина пароля, а также выдвигается требование, чтобы пароль обязательно включал одновременно строчные и прописные буквы, числа и различные знаки пунктуации.

К сожалению, люди в большинстве своем плохо запоминают длинные защищенные пароли, особенно если они обязаны регулярно их изменять.

Существует много программных инструментов, предназначенных помочь пользователям управляться с большим количеством учетных записей и связанных с ними паролей, и они, кажется, уже стали обязательным инструментом в повседневной жизни в режиме онлайн. Однако все эти программы защищены ровно настолько, насколько защищенным является мастер-пароль, которым пользуются для доступа к ним.

Обычным пользователям помочь сгенерировать и запомнить достаточно хорошо защищенные пароли могут несколько мнемонических правил. Полных слов из словаря и общеизвестных личных дат (например, дней рождения членов семьи) следует избегать, поскольку это именно те сочетания, которые первыми будут опробованы любым достаточно опытным атакующим. Однако фраза из нескольких умеренно длинных слов, которые обычно не употребляются вместе, смешанных в различных позициях с цифрами и знаками пунктуации, может образовать легко запоминаемый мнемонический узор, который будет уже не так просто подобрать или угадать.

В качестве примера приведу такую историю. Одного из друзей автора, большого любителя животных, однажды попросили пожить некоторое время в доме коллеги. При этом дополнительными заданиями было кормить сухим собачьим кормом стайку диких енотов на заднем дворе, а также смазывать листья домашних растений майонезом для большего блеска. В этой ситуации ему несложно было запомнить пароль `Racc0OnTreat&May0Sh1n`, чего не скажешь о попытках его разгадать, по крайней мере до тех пор, пока он не был напечатан в этой книге.

Атаки изнутри

Как только незваный гость (или, возможно, авторизированный пользователь с преступными намерениями) получит доступ к компьютерной системе, следующим его шагом, скорее всего, будет ее изучение в поисках интересующей его информации или мест, где можно разместить вредоносное программное обеспечение. Это достаточно простая процедура, если мошенник получил доступ к учетной записи администратора, и по этой причине пароль администратора всегда тщательно охраняется. Если же доступ к системе был получен через учетную запись обычного пользователя, возникает необходимость обмануть операционную систему, чтобы она позволила злоумышленнику выйти за пределы того набора привилегий, который предоставлен этому пользователю. Например, злоумышленник может попытаться обмануть программу управления памятью, чтобы получить для своего процесса разрешение на доступ к ячейкам памяти вне выделенной ему области, либо мошенник может попытаться обмануть программу управления файлами для считывания файла, доступ к которому ему запрещен.

Современные центральные процессоры обладают специальными средствами, предназначенными для пресечения подобных попыток. В качестве примера рассмотрим необходимость ограничить для процесса доступ к ячейкам памяти исключительно той областью, которая была выделена ему программой управления памятью. При отсутствии подобных ограничений процесс мог бы удалить программы операционной системы из памяти и взять управление компьютером на себя. Чтобы пресечь такую возможность, центральные процессоры, предназначенные для систем с разделением времени, обычно включают регистр специального назначения, в котором операционная система сможет хранить верхний и нижний пределы выделенной процессу области памяти. В этом случае, пока процесс выполняется, ЦП сможет сравнить адрес каждой ячейки памяти, к которой он обращается, с установленными границами и удостовериться, что этот адрес находится в заданных пределах. Если произойдет обращение к ячейке памяти за пределами области, выделенной данному процессу, ЦП автоматически остановит процесс (выполнив процедуру прерывания) и передаст управление операционной системе, чтобы она могла предпринять соответствующие действия.

В этом примере присутствует тонкая, но очень важная проблема. Без применения дополнительных функций обеспечения безопасности процесс все еще имеет возможность получить доступ к ячейкам памяти вне выделенной ему области, — просто изменив значения в том регистре специального назначения, в котором хранится информация о ее границах. Иначе говоря, если процесс стремится получить доступ к дополнительным участкам основной памяти, он

может просто увеличить значение того регистра, в котором хранится информация о верхней границе выделенного ему участка, а затем воспользоваться этим дополнительным участком в своих целях без получения разрешения со стороны операционной системы.

Чтобы предотвратить подобные действия, центральные процессоры, предназначенные для систем с разделением времени, могут функционировать при одном из двух **уровней привилегий**. Один из них называют *привилегированным* режимом, а другой — обычным или *непривилегированным* режимом. Работая в привилегированном режиме, ЦП может выполнять все команды своего машинного языка, но когда он переходит в обычный режим, выполнение некоторых команд ему запрещается. Те команды, которые доступны только в привилегированном режиме, называют **привилегированными командами**. (Типичным примером привилегированных команд являются команды изменения содержимого регистров границ выделенного участка памяти и команды, изменяющие текущий уровень привилегий ЦП.) Всякая попытка выполнить привилегированную команду, когда ЦП находится в обычном режиме, приводит к прерыванию. В результате ЦП переходит в привилегированный режим работы, а управление передается обработчику прерываний операционной системы.

В момент включения центральный процессор всегда оказывается в привилегированном режиме работы. Поэтому, когда операционная система запускается и начинается процедура начальной загрузки, процессор может выполнять любые команды. Однако после ее завершения каждый раз, когда операционная система разрешает некоторому процессу начать свое выполнение в пределах отведенного ему кванта времени, она переключает ЦП в обычный, непривилегированный режим работы посредством выполнения команды "Изменить уровень привилегий ЦП". В результате операционная система гарантированно получает соответствующее извещение, как только работающий процесс предпримет попытку выполнить любую привилегированную команду, а значит, у нее появляется возможность обеспечить сохранение целостности компьютерной системы.

Привилегированные команды и управление уровнем привилегий являются мощными инструментами, позволяющими операционной системе обеспечивать безопасную работу системы. Однако использование этих инструментов представляет собой сложную задачу при разработке операционных систем, поэтому ошибки все еще периодически обнаруживаются и в уже существующих системах. Единственный изъян в управлении уровнем привилегий может создать лазейку для катастрофического вмешательства со стороны злонамеренных программистов или просто в результате непреднамеренных программных ошибок. Если процесс получит возможность изменить значение таймера, который

Многофакторная аутентификация

Одно из решений глубочайшей проблемы всего человечества, связанной с использованием слабых паролей для защиты своих учетных записей на компьютерах, состоит в требовании предоставления более чем одного свидетельства аутентичности для получения доступа. Пароли — это средство проверки того, что пользователи *знают*, давая ответы на вопросы, обычно используемые в процедурах обеспечения безопасности, такие как "Девичья фамилия вашей бабушки по материнской линии". Вторым фактором аутентификации может стать то, что пользователь *имеет*, например банковская карточка, специальный токен системы защиты или смартфон с конкретным телефонным номером. Все возрастающее количество сетевых учетных записей требует перехода к *двухфакторной аутентификации*, при которой пользователь должен предоставить сначала пароль, а затем специальный код доступа из текстового сообщения, поступающего на смартфон пользователя в процессе входа в систему. Третий набор факторов аутентификации может зависеть от чего-то, связанного с тем, *кто* является пользователем. Биометрические сенсоры, встроенные в мобильные устройства, сейчас способны легко распознать уникальные отпечатки пальцев, голос или узор на сетчатке глаза пользователя. Некоторые исследователи работают над дополнительными биометрическими механизмами, к которым можно отнести, например, использование данных акселерометров смартфонов для распознавания уникального образца походки пользователя.

Однако ни один из этих механизмов пока не доказал своей безусловной надежности, так как искусные злоумышленники применяют различные схемы для отслеживания паролей пользователей, обмана встроенных датчиков или даже принуждения обманным путем администратора системы изменить номер смартфона, связанного с учетной записью другого пользователя, на желаемый.

контролирует работу системы разделения времени, он сможет увеличить свои кванты времени и захватить доминирующие позиции в работе всей системы. Если процесс получит возможность прямого доступа к периферийным устройствам, он сможет считывать файлы без какого-либо контроля со стороны программы управления файлами. Если процесс получит доступ к ячейкам памяти за пределами выделенного ему сегмента, он сможет читать и даже изменять любые данные, используемые другими процессами. Таким образом, обеспечение безопасности по-прежнему остается важнейшей задачей администратора системы, а также операционной системы и разработчиков процессоров.

3.5. Вопросы и упражнения

1. Приведите примеры ненадежных паролей и объясните, почему именно эти пароли являются неудачными.
2. Процессоры серии Pentium от корпорации Intel предоставляют четыре уровня привилегий. Почему, по вашему мнению, разработчики этих процессоров решили использовать именно четыре уровня привилегий, а не три или пять?
3. Если процесс в системе с разделением времени получит доступ к ячейкам памяти вне выделенного ему сегмента, то как он сможет получить контроль над всей машиной?

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. Перечислите четыре вида действий, выполняемых типичной операционной системой.
2. Кратко охарактеризуйте различия между пакетной и интерактивной обработкой.
3. Предположим, что три элемента, *R*, *S* и *T*, помещены в очередь в указанном порядке. Затем один из элементов удаляется из очереди еще до того, как в нее будет помещен четвертый элемент — *X*. Далее один элемент удаляется из очереди, а элементы *Y* и *Z* помещаются в нее в указанном порядке, после чего из очереди последовательно удаляются один элемент за другим, пока она не окажется пустой. Перечислите все элементы в том порядке, в котором они удалялись из очереди.
4. В чем заключается основное различие между встроенными системами и ПК?
5. Что такое многозадачная операционная система?
6. Если у вас есть персональный компьютер или смартфон, укажите несколько ситуаций, в которых вы сможете извлечь преимущества из его мультизадачных возможностей.
7. Обратившись к той компьютерной системе, с которой вы достаточно хорошо знакомы, приведите два примера прикладного программного

обеспечения и два примера служебных программ. Поясните, чем вы руководствовались, классифицируя эти программы тем или иным образом.

8. **а.** В чем состоит назначение интерфейса пользователя в составе операционной системы?

   б. Какую роль в операционной системе играет ее ядро?

9. Какая структура из папок определяется следующим путем доступа: `X/Y/Z`?

10. Дайте определение термину "процесс" при его использовании в контексте операционной системы.

11. Какая информация содержится в таблице процессов, поддерживаемой операционной системой?

12. В чем различие между процессом, готовым к выполнению, и ожидающим процессом?

13. В чем заключается различие между виртуальной и основной памятью?

14. Предположим, что компьютер имеет 512 Мбайт основной памяти и операционной системе необходимо создать виртуальную память вдвое большего размера, используя страницы размером 2 Кбайт. Сколько страниц для этого потребуется?

15. Какие сложности могут возникнуть в системе с разделением времени, если два процесса одновременно запрашивают доступ к одному и тому же файлу? Существуют ли ситуации, в которых программа управления файлами предоставляет такой доступ? В каких случаях программа управления файлами отвечает отказом?

16. В чем состоит различие между прикладным программным обеспечением и системным программным обеспечением? Приведите примеры программного обеспечения обоих типов.

17. Дайте определение понятий балансировки загрузки и масштабирования в контексте многопроцессорной архитектуры.

18. Кратко опишите процесс начальной загрузки.

19. Чем объясняется необходимость выполнения процесса начальной загрузки?

20. Если у вас есть ПК, запишите последовательность действий, которую вы сможете наблюдать при его включении. Затем попробуйте определить, какое сообщение появляется на экране компьютера до того, как процесс начальной загрузки действительно начнется. Какое программное обеспечение выводит эти сообщения?

21. Предположим, что в операционной системе с разделением времени процессам выделяются кванты времени по 10 мс, а машина в среднем выполняет пять команд за наносекунду. Сколько команд может быть выполнено за один полный квант времени?
22. Если оператор вводит шестьдесят слов в минуту (при этом слова состоят из пяти букв), сколько времени пройдет между моментом окончания ввода каждого символа? Если операционная система с разделением времени выделяет процессам кванты времени по 10 мс и мы игнорируем время, необходимое на переключение процессов, сколько квантов времени может быть распределено системой между двумя последовательно введенными оператором символами?
23. Предположим, что операционная система с разделением времени использует кванты времени длительностью 50 мс. Если обычно процедура позиционирования головки чтения/записи диска над нужной дорожкой занимает 8 мс и еще 17 мс будет затрачено, пока требуемые данные пройдут под головкой чтения/записи, то какую часть кванта времени программа проведет в ожидании выполнения операции чтения с диска? Если машина способна выполнять по десять команд за каждую наносекунду, сколько команд она смогла бы выполнить за время этого ожидания? (Именно по этой причине системы с разделением времени обычно позволяют выполняться другим процессам, в то время как первый процесс будет ожидать окончания обслуживания периферийным устройством.)
24. Перечислите пять ресурсов, доступ к которым должна координировать многозадачная операционная система.
25. Говорят, что процесс зависит от ввода-вывода, если ему требуется выполнить много операций ввода-вывода. Процесс, преимущественно выполняющий вычисления в пределах системы "ЦП–память", называют вычислительно зависимым. Если вычислительно зависимый процесс и процесс, зависимый от ввода-вывода, ожидают предоставления кванта времени, кому должен быть предоставлен приоритет и почему?
26. Какая система достигнет большей производительности — система, выполняющая два процесса, зависимых от ввода-вывода (см. задание 25), или система с одним процессом, зависимым от ввода-вывода, и вычислительно зависимым процессом? Почему?
27. Разработайте набор инструкций, который будет определять действия программы — диспетчера операционной системы по истечении кванта времени, выделенного выполняющемуся процессу.
28. Назовите компоненты информации о состоянии процесса.

29. Приведите пример ситуации в системе с разделением времени, при которой процесс не использует весь предоставленный ему квант времени.

30. Перечислите в хронологическом порядке основные события, которые происходят при прерывании процесса.

31. Дайте ответ на каждый из приведенных ниже вопросов в терминах той операционной системы, которой вы пользуетесь.

а. Как можно попросить операционную систему скопировать файл из одного местоположения в другое?

б. Как можно попросить операционную систему показать вам содержимое каталога на диске??

в. Как дать операционной системе указание выполнить некоторую программу?

32. Дайте ответ на каждый из приведенных ниже вопросов в терминах той операционной системы, которой вы пользуетесь.

а. Как операционная система ограничивает доступ к системе только теми пользователями, которые имеют на это право?

б. Как дать операционной системе указание отобразить текущее состояние ее таблицы процессов?

в. Как можно сообщить операционной системе о том, что вы не хотите, чтобы другие пользователи машины могли получать доступ к вашим файлам?

***33.** Объясните назначение команды "test-and-set" (проверить и установить), присутствующей во многих машинных языках. Почему важно, чтобы весь процесс проверки и установки был реализован в виде одной команды?

***34.** Банкир, который всего имеет 100 000 долларов, дает взаймы по 50 000 долларов двум клиентам. Позже оба клиента снова обращаются к нему с уверениями о том, что, прежде чем они смогут вернуть долг, им нужно еще по 10 000 долларов для завершения тех сделок, в которых задействованы кредиты. Банкир решает проблему путем заимствования дополнительных сумм у внешнего источника и дает своим клиентам дополнительный кредит (под более высокий процент). Какое из трех условий тупиковой ситуации (ситуации взаимной блокировки) устранил банкир в данном случае?

***35.** Студенты, желающие записаться на курс "Моделирование железных дорог" в местном университете, должны получить разрешение от преподавателя и внести плату за работу в лаборатории. Эти два требования могут выполняться независимо и в любом порядке, причем в различных точках

студенческого городка. Количество слушателей ограничено 20 студентами. Это ограничение отслеживают как преподаватель, который выдаст разрешение только 20 студентам, так и финансовый отдел, который разрешит внести плату также только 20 студентам. Предположим, что 19 студентов уже зачислены на этот курс, а на последнее место претендуют еще два студента: один уже получил разрешение от преподавателя, а второй уже внес плату. Какое условие возникновения тупиковой ситуации устраняется каждым из следующих возможных решений проблемы.

а. Обоим студентам разрешено посещать курс.

б. Группа уменьшена до 19 человек, так что ни одному из двух последних студентов не разрешено записаться на курс.

в. Ни одному из двух претендентов не разрешено участвовать в занятиях, а разрешение выдано третьему студенту.

г. Принято решение о том, что единственным требованием для зачисления на курс является внесение платы. Таким образом, студент, уже внесший плату, записывается на курс, а второй получает отказ.

***36.** Поскольку каждая область на экране дисплея может использоваться только одним процессом в одно и то же время (иначе изображение на экране будет неразборчивым), эти области можно рассматривать как неразделяемые ресурсы, которые распределяются программой управления окнами. Устранение какого из трех условий, необходимых для возникновения взаимной блокировки, должна обеспечивать программа управления окнами, чтобы исключить возможность возникновения подобной ситуации?

***37.** Предположим, что неразделяемые ресурсы в компьютерной системе классифицированы как ресурсы первого, второго и третьего уровней. Также предположим, что каждому процессу в системе предписано запрашивать нужные ему ресурсы согласно этой классификации, т.е. сначала он должен запросить все необходимые ему ресурсы первого уровня, прежде чем сможет запрашивать какой-либо ресурс второго уровня. После того как все требуемые ресурсы первого уровня будут получены, процесс может запросить все необходимые ему ресурсы второго уровня и т.д. Может ли в этой схеме возникнуть ситуация взаимной блокировки? Почему "да" или почему "нет"?

***38.** Каждая из двух рук робота запрограммирована поднимать детали с конвейерной ленты, проверять их на допуски и класть в один из двух контейнеров в зависимости от результатов проверки. Детали поступают по одной с достаточным интервалом между поступлениями. Чтобы избежать ситуации, когда обе руки робота попытаются взять одну и ту же деталь, компьютеры, управляющие руками, используют общую ячейку памяти.

Если в момент приближения детали рука свободна, управляющий ею компьютер считывает значение общей ячейки. Если оно отлично от нуля, рука пропускает деталь. В противном случае компьютер помещает ненулевое значение в ячейку памяти и направляет руку, чтобы поднять деталь. После того как это действие завершено, компьютер вновь помещает в ячейку памяти значение 0. Какая последовательность действий может привести к конфликту между двумя руками?

***39.** Опишите, как можно использовать механизм очереди в процессе спулинга выводимой на принтер информации.

***40.** Процесс, который ожидает получения кванта времени, но так его никогда и не получает, называют **зависшим**.

а. Участок дороги в центре перекрестка можно рассматривать как неделимый ресурс, на который претендуют все приближающиеся к перекрестку автомобили. Для распределения права доступа к этому ресурсу используется не операционная система, а светофор. Если светофор способен учитывать интенсивность дорожного движения в каждом из направлений и запрограммирован так, чтобы давать зеленый свет наибольшему из потоков, то автомобили из меньшего потока могут подолгу простаивать под этим светофором (поток зависает). Как можно избежать этого зависания?

б. По какой причине может зависнуть процесс, если диспетчер всегда выделяет кванты времени согласно системе приоритетов, в которой приоритет каждого процесса остается постоянным? (*Подсказка.* Какой приоритет у процесса, который только что завершил использование своего кванта времени по сравнению с ожидающими процессами, и как следствие какому процессу будет предоставлен следующий квант времени?) Как, по вашему мнению, во многих операционных системах удается избежать подобной проблемы?

***41.** В чем сходство между ситуацией взаимной блокировки и зависанием (см. задание 40)? В чем различие между ними?

***42.** Ниже представлена задача об обедающих философах, которая в свое время была предложена Э.В. Дейкстрой, а сейчас стала частью фольклора в области компьютерных наук. Пять философов сидят вокруг круглого стола. Перед каждым из них стоит тарелка со спагетти. На столе лежит пять вилок, по одной между каждой парой тарелок. Любой философ может либо есть, либо думать. Чтобы есть, философу необходимо взять две вилки, лежащие по обе стороны его тарелки. Укажите на возможность взаимной блокировки и зависания (см. задание 40), которая присутствует в задаче об обедающих философах.

*43. Какая проблема возникнет, если в системе с разделением времени размер кванта времени делать все меньше и меньше? Что будет происходить, если делать эти кванты все больше и больше?

*44. По мере развития компьютерных наук машинные языки постоянно расширялись с целью включения в них новых специализированных команд. Три такие машинные команды, которые сейчас широко используются в современных операционных системах, были представлены в разделе 3.4. Что это за команды?

45. Укажите два вида действий, которые могут быть выполнены администратором операционной системы, но не могут быть выполнены ее рядовым пользователем.

46. Каким образом операционная система не позволяет процессу получить доступ к областям памяти, выделенным другим процессам?

47. Предположим, что пароль представляет собой строку из девяти символов латинского алфавита (он включает 26 букв). Если каждый возможный в этом случае пароль можно протестировать за одну миллисекунду, сколько времени займет проверка всех возможных в этой ситуации паролей?

48. Почему процессоры, разработанные для операционных систем с разделением времени, могут работать при различных уровнях привилегий?

49. Назовите два действия, выполнение которых требует использования привилегированных команд.

50. Укажите три способа, которыми процесс может бросить вызов безопасности компьютерной системы, если они не будут своевременно предотвращены операционной системой.

51. Что такое многоядерная операционная система?

52. В чем состоит различие между обновлением прошивки и обновлением операционной системы?

53. Каким образом программа управления окнами связана с операционной системой?

54. Является ли программа Internet Explorer частью операционной системы Microsoft Windows?

55. Какие особенности свойственны адресу встроенной операционной системы?

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответ на предложенный вопрос. Вы должны также разобраться, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. Предположим, что вы работаете в многопользовательской операционной системе, которая позволяет просматривать имена файлов, принадлежащих другим пользователям, а также содержимое этих файлов, если оно не защищено паролем. Будет ли просмотр этой информации выглядеть так, как будто вы рыщете по чужому незапертому дому, или же это похоже на чтение журналов, предлагаемых пациентам, ожидающим приема у стоматолога?
2. Если вы имеете доступ к многопользовательской вычислительной системе, то в чем заключаются ваши обязанности при выборе пароля?
3. Если дефект в системе защиты операционной системы позволяет злонамеренному программисту получить неавторизованный доступ к важным данным, в какой степени ответственность за это должен нести разработчик операционной системы?
4. Является ли вашей обязанностью запереть свой дом таким образом, чтобы незваные гости не смогли в него проникнуть, либо все окружающие обязаны стоять у вашего дома и ожидать, пока их не пригласят? Является ли обязанностью операционной системы охранять доступ к компьютеру и его содержимому, или же это хакеры обязаны оставить его в покое?
5. В своей книге "Уолден, или Жизнь в лесу" Генри Дэвид Торо утверждал, что мы превращаемся в инструменты наших инструментов, т.е. вместо того, чтобы получать выгоду от использования наших инструментов, мы тратим свое время на получение и поддержание работы своих инструментов. До какой степени, по вашему мнению, это утверждение справедливо в отношении компьютеров? Например, если вы являетесь владельцем персонального компьютера, сколько времени вы потратили на то, чтобы заработать сумму денег, необходимую для его приобретения; на то, чтобы разобраться в его операционной системе; на то, чтобы научиться пользоваться его служебными программами и прикладным программным обеспечением; сколько времени ушло на его обслуживание и на загрузку

обновлений установленного на нем программного обеспечения — в сравнении со временем, которое вы потратили, получая ту или иную выгоду от его использования? Когда вы им пользуетесь, то считаете, что ваше время было хорошо потрачено? Вы будете более социально активны при наличии у вас компьютера или без него?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Bishop M. *Introduction to Computer Security.* — Boston, MA: Addison-Wesley, 2005.
2. Craig B. *CyberLaw, The Law of the Internet and Information Technology.* — Upper Saddle River, NJ: Prentice-Hall, 2012.
3. Davis W.S., Rajkumar T.M. *Operating Systems: A Systematic View*, 6th ed. — Boston, MA: Addison-Wesley, 2005.
4. Deitel H. M., Deitel P. J., Choffnes D.R. *Operating Systems*, 3rd ed. — Upper Saddle River, NJ: Prentice-Hall, 2005.
5. Silberschatz A., Galvin P.B., Gagne G. *Operating System Concepts*, 9th ed. — New York: Wiley, 2012.
6. Stallings W. *Operating Systems*, 8th ed. — Upper Saddle River, NJ: Prentice-Hall, 2014.
7. Tanenbaum A.S. *Modern Operating Systems*, 3rd ed. — Upper Saddle River, NJ: Prentice-Hall, 2008.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Столлингс В. *Операционные системы: внутренняя структура и принципы проектирования*. 9-е изд. — СПб.: ООО "Диалектика", 2020.
2. Столлингс В. *Операционные системы. Основные принципы построения и проектирования*, 4-е изд. — М.: Издательский дом "Вильямс", 2002.
3. Бармен С. *Разработка правил информационной безопасности*. — М.: Издательский дом "Вильямс", 2002.

Глава 4

В этой главе мы рассмотрим ту область компьютерных наук, задачей которой является изучение различных вопросов, связанных с соединением компьютеров между собой с целью обмена информацией и совместного использования их ресурсов, т.е. создания компьютерных сетей. Вы познакомитесь с принципами построения и работы сетей, отличительными особенностями сетевых приложений и основными проблемами обеспечения безопасности. Отдельной важной темой будет обсуждение различных аспектов построения и функционирования Всемирной сети, образованной соединением между собой множества локальных сетей, известной как Интернет.

Компьютерные сети и Интернет

4.1. ОСНОВЫ КОМПЬЮТЕРНЫХ СЕТЕЙ
- Классификация сетей
- Сетевые протоколы
- Объединение сетей
- Методы взаимодействия процессов
- Распределенные системы

4.2. ИНТЕРНЕТ
- Архитектура Интернета
- Интернет-адресация
- Интернет-приложения

4.3. ВСЕМИРНАЯ ПАУТИНА: WORLD WIDE WEB
- Реализация веба
- Язык гипертекстовой разметки HTML
- Язык XML
- Действия на стороне клиента и на стороне сервера

*4.4. ПРОТОКОЛЫ ИНТЕРНЕТА
- Многоуровневый подход к организации сетевого ПО
- Семейство протоколов TCP/IP

*4.5. ПРОСТОЙ ПРИМЕР МОДЕЛИ "КЛИЕНТ/СЕРВЕР"
- Сценарий "В бесконечность и далее"

4.6. КИБЕРБЕЗОПАСНОСТЬ
- Типы атак
- Защита систем и их лечение
- Криптография
- Правовое регулирование безопасности сетей

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

Необходимость обеспечить совместный доступ к информации и другим ресурсам различных компьютеров естественным образом привела к разработке методов связывания компьютерных систем между собой в единую структуру, получившую название **компьютерная сеть**, в которой компьютеры соединяются между собой особым образом, позволяющим организовать передачу данных от одной машины к другой. В таких системах пользователи компьютеров имеют возможность обмениваться информацией и совместно использовать ресурсы — такие, как потоковый контент мультимедиа, программное обеспечение или устройства хранения данных, — разбросанные по всем уголкам глобальной сетевой структуры. Программное обеспечение, обеспечивающее столь широкие возможности, за десятилетия прошло путь от простейших утилит до постоянно развивающейся и расширяющейся системы сетевого программного обеспечения, поддерживающей функционирование сложной сетевой инфраструктуры, охватывающей миллиарды пользователей многих миллиардов различных компьютерных устройств. В сущности, сетевое программное обеспечение эволюционировало из простого набора утилит в особую операционную систему, обеспечивающую совместную работу всей компьютерной сети. В этой главе вы познакомитесь с этой бурно развивающейся областью компьютерных наук.

4.1. Основы компьютерных сетей

Мы начнем эту главу с обсуждения всего разнообразия базовых концепций компьютерных сетей.

Классификация сетей

Каждая компьютерная сеть принадлежит к одной из следующих категорий: **персональная** (или **домашняя**) **вычислительная сеть**, или **ПВС** (Personal Area Network — PAN), **локальная вычислительная сеть**, или **ЛВС** (Local Area Network — LAN), **муниципальная вычислительная сеть** или **МВС** (Metropolitan Area Network — MAN), и **глобальная вычислительная сеть**, или **ГВС** (Wide Area Networks — WAN). Персональная вычислительная сеть обычно используется для коммуникаций на очень небольшие расстояния — обычно от одного-двух до полутора десятков метров — как между беспроводными наушниками и смартфоном или между беспроводной мышью и персональным компьютером. В противоположность этому локальная сеть, как правило, состоит из нескольких компьютеров, находящихся в одном здании или комплексе зданий. Например, компьютеры, используемые в университетском городке или на одном заводе, могут быть соединены единой локальной сетью. Муниципальная сеть имеет средние размеры и может охватывать все

местное сообщество, тогда как глобальная сеть соединяет машины, которые могут находиться на очень больших расстояниях, скажем, в противоположных концах крупного города и даже на противоположных сторонах Земного шара.

Другой принцип классификации сетей базируется на том, является ли право собственности на проект внутреннего устройства сети общественным достоянием или же принадлежит отдельному субъекту, как, например, отдельному лицу или корпорации. Сеть первого типа называется **открытой** сетью, а второго типа — **закрытой** или **частной** сетью. Открытые сетевые проекты распространяются свободно, и часто их популярность возрастает до такой степени, что они в конечном счете преобладают над закрытыми разработками, использование которых ограничивается стоимостью лицензии и условиями контракта.



Основные положения для запоминания

- **Именно открытые стандарты являются той питательной средой, которая обеспечивает рост Интернета.**

Глобальная сеть Интернет (*Internet* — популярная Всемирная сеть сетей, которую мы будем подробно рассматривать в этой главе) является открытой системой. В частности, связь через Интернет регулируется открытой системой стандартов, известной как семейство протоколов TCP/IP, которое мы будем обсуждать в разделе 4.4. Каждый волен использовать эти стандарты без необходимости внесения какой-либо платы или заключения лицензионных соглашений. В противоположность этому некоторая компания, например корпорация Microsoft, может разработать запатентованные системы, на которые решит сохранить права собственности, что позволит этой компании получать доход от продажи или передачи этих продуктов в аренду.

Совершенно иной способ классификации компьютерных сетей основывается на их топологии, т.е. на выбранной физической схеме соединения входящих в них машин. К двум наиболее распространенным топологиям относятся **шина** (*bus*), при которой все машины соединены с общей линией связи, называемой шиной (рис. 4.1, *а*), и **звезда**, при которой одна машина служит координационным центром (*switch*, **сетевой коммутатор** или свитч), к которому подключаются все остальные машины (рис. 4.1, *б*). Технология шины получила распространение в 1990-х годах, когда специально для нее было разработано семейство стандартов, получившее название "Ethernet", но после ревизии этих стандартов с целью повышения скорости передачи данных логически сети стандарта Ethernet по-прежнему ведут себя, как имеющие топологию шины, но физически они чаще всего реализуются с использованием топологии звезды.

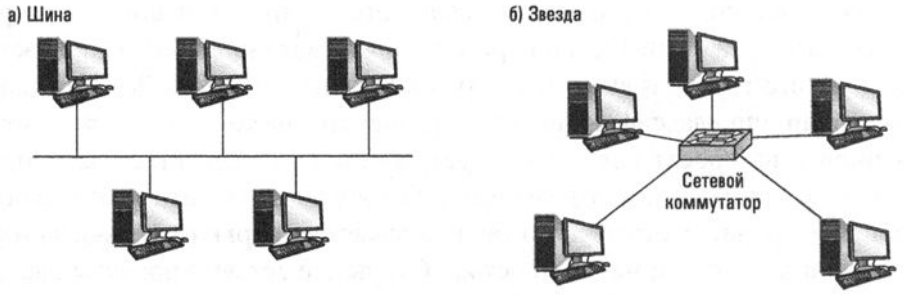


Рис. 4.1. Два основных типа конфигурации сетей

Топология звезды ведет свою историю еще с 1970-х годов. Фактически она явилась логическим следствием общепринятой в то время парадигмы большого центрального компьютера, обслуживающего многих пользователей. Поскольку простой терминал, за которым работали эти пользователи, со временем эволюционировал в самостоятельные небольшие компьютеры, это естественным образом привело к появлению сетей с топологией звезды. Сегодня конфигурация звезды также очень популярна в беспроводных сетях, в которых взаимодействие осуществляется посредством радиоволн и центральное устройство, получившее название **точка доступа**, служит координационным центром, через которое и осуществляется взаимодействие всех остальных компонентов сети.

Различие между сетями с топологией шины и сетями с топологией звезды не всегда можно провести исходя лишь из физической компоновки этих сетей. Основное различие состоит в том, как машины в сети взаимодействуют между собой: непосредственно друг с другом через общую шину или через некоторую промежуточную центральную машину. Например, сеть с топологией шины может выглядеть вовсе не как длинная шина, с которой компьютеры соединяются короткими линиями связи, как показано на рис. 4.1. В действительности это может быть очень короткая шина с достаточно длинными линиями связи, идущими к отдельным машинам, в результате чего физическое представление сети больше напоминает звезду. На практике сеть с топологией шины чаще всего строится с использованием длинных линий связи, идущих от каждого компьютера к центральной точке, в которой они подключаются к специальному устройству, получившему название **сетевой концентратор** или хаб (*hub*). Этот концентратор представляет собой нечто большее, чем просто очень короткий отрезок шины. Однако все, что он делает, исчерпывается пересылкой любого полученного им сигнала (возможно, с предварительным усилением) обратно ко всем подключенным к нему машинам. В результате получается сеть, которая физически выглядит, как звезда, но работает по принципам сети с топологией шины.

Сетевые протоколы

Чтобы сеть функционировала надежно, очень важно установить определенные правила, регулирующие любое взаимодействие ее компонентов между собой. Подобные правила называются **протоколами**. Разработка и принятие сетевых протоколов в качестве стандартов позволяет различным изготовителям разрабатывать такие собственные сетевые продукты, которые будут совместимы с продукцией от других поставщиков. Следовательно, разработка стандартных сетевых протоколов является совершенно необходимым элементом развития сетевых технологий.

В качестве введения в концепцию сетевого протокола давайте рассмотрим проблему координации передачи сообщений между компьютерами в сети. При отсутствии правил, регулирующих подобное взаимодействие, все эти компьютеры могут одновременно настаивать на передаче сообщений в одно и то же время или отказать в помощи другим машинам, когда им эта помощь потребуется.

В компьютерной сети, построенной в соответствии со стандартами Ethernet, право передачи сообщений регулируется протоколом, получившим название **CSMA/CD** (*Carrier Sense, Multiple Access with Collision Detection* — множественный доступ с опросом несущей и разрешением конфликтов). Этот протокол предусматривает, что каждое посланное сообщение будет передано всем машинам, соединенным шиной (рис. 4.2). Каждая машина просматривает все поступающие сообщения, но отбирает только те, которые адресованы именно ей. Чтобы отправить собственное сообщение, машина ожидает, пока в шине наступит тишина, затем начинает отправку, одновременно продолжая наблюдать за шиной. Если в этот момент и другая машина пытается отправить свое сообщение, обе обнаруживают конфликт и делают паузу на произвольно короткий промежуток времени, а затем предпринимают попытку вновь начать отправку. В результате складывается ситуация, аналогичная той, которая возникает при разговоре в небольшой группе людей. Если два человека начинают говорить одновременно, оба замолкают. Различие состоит в том, что люди могут выйти из ситуации таким образом: "Простите, что вы хотели сказать?", "Нет, нет, говорите вы первым"; в то время как по протоколу CSMA/CD каждая машина просто делает следующую попытку с небольшой задержкой.

Следует отметить, что протокол CSMA/CD несовместим с беспроводными сетями, имеющими топологию звезды, когда все устройства взаимодействуют между собой через центральное звено, т.е. *точку доступа*. Проблема состоит в том, что устройство может оказаться неспособным определить, что передаваемое им сообщение накладывается на сообщения от других устройств. Например, устройство может просто не воспринимать сигналы от других устройств, поскольку его собственный сигнал заглушает сигналы от остальных устройств

в сети. В другой ситуации сигналы от других устройств могут быть блокированы некоторым объектом или устройства могут располагаться между собой на слишком большом расстоянии, хотя каждое из них может успешно взаимодействовать с точкой доступа (эта ситуация получила название **проблема скрытого терминала**; рис. 4.3). В результате для беспроводных сетей за основу была принята такая политика: пытаться избегать коллизий вместо попыток обнаружить их. Такой подход получил название **CSMA/CA** (*Carrier Sense, Multiple Access with Collision Avoidance* — множественный доступ с прослушиванием несущей и избеганием коллизий). Он был стандартизован институтом IEEE (см. врезку "IEEEngineers" в главе 7) в протоколе, получившем обозначение IEEE 802.11, однако чаще всего на этот протокол ссылаются как на **Wi-Fi**. Следует подчеркнуть, что протокол CSMA/CA разработан так, что позволяет лишь избегать коллизий, но не полностью исключить их появление. Если коллизия все же возникает, соответствующее сообщение необходимо будет передать заново.

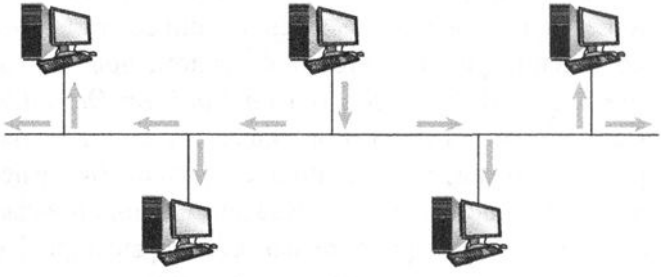


Рис. 4.2. Обмен информацией в сети с топологией шины

Наиболее общий подход к реализации требования избегать коллизий строится на предоставлении преимуществ тому устройству, которое уже ожидает возможности передать сообщение. Используемый протокол подобен протоколу CSMA/CD сетей Ethernet. Основное различие состоит в том, что когда у устройства возникает необходимость передать сообщение и оно обнаруживает, что в канале связи тишина, оно не начинает передачу сообщения немедленно. Вместо этого оно ожидает короткий промежуток времени, а затем начинает передачу только в том случае, если канал все это время оставался свободным. Если за этот период обнаруживается занятость канала, устройство ожидает некоторый случайно выбранный период времени, прежде чем предпринять следующую попытку. По истечении этого периода устройство имеет право немедленно нарушить молчание в канале. Это означает, что коллизии между "новичками" и теми устройствами, которые уже ожидали возможности отправки сообщения можно будет избежать, поскольку "новички" не получат разрешения захватить свободный канал до тех пор, пока любому уже ожидавшему доступа к каналу устройству не будет предоставлена возможность начать передачу.

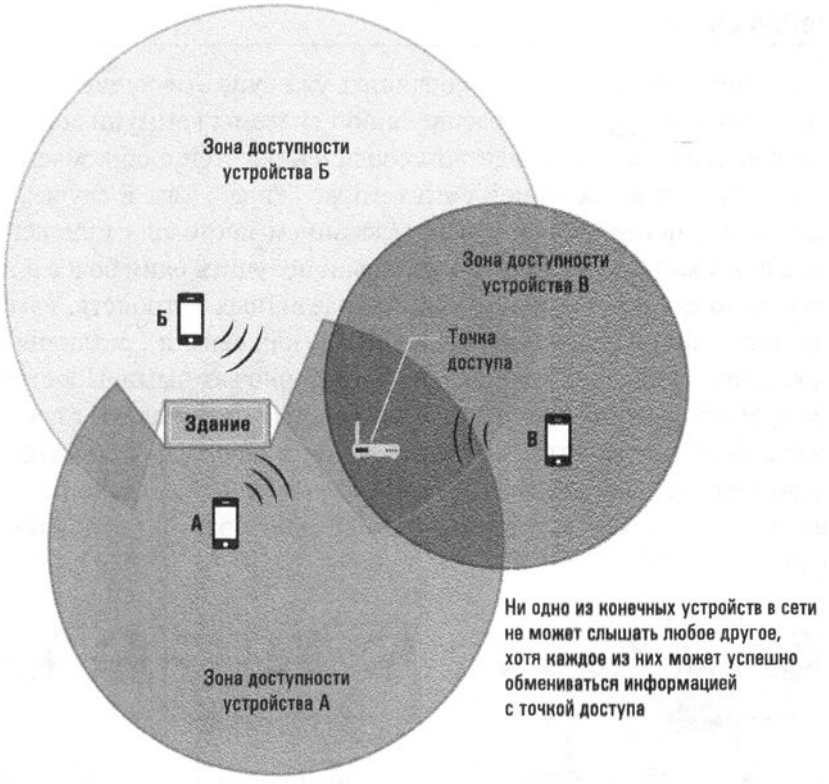
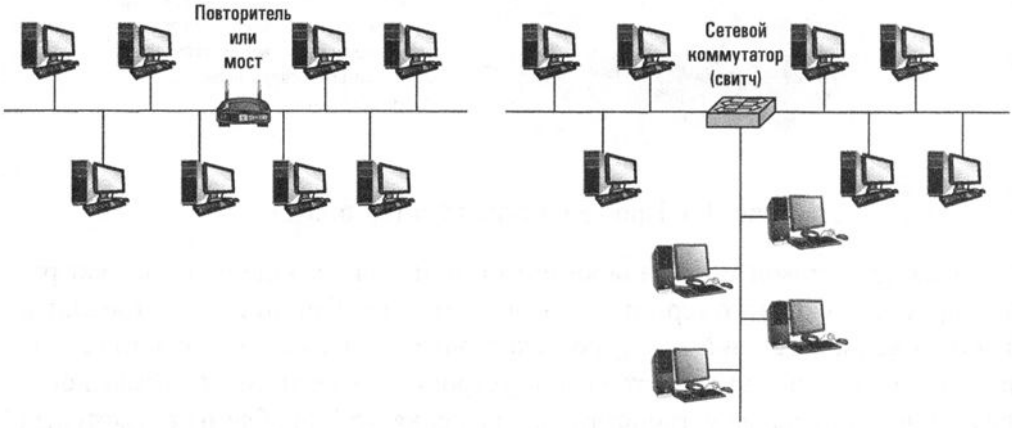


Рис. 4.3. Проблема скрытого терминала

Однако даже такой модифицированный протокол все еще не позволяет решить проблему скрытого терминала. Более того, любой протокол, построенный на обнаружении различий между состояниями занятости и незанятости канала связи, требует, чтобы каждое отдельное устройство в сети имело возможность слышать все остальные устройства. Для решения этой проблемы в некоторых сетях Wi-Fi требуется, чтобы каждое устройство, прежде чем начать передачу своего сообщения, отправляло точке доступа короткие сообщения-"запросы" и ожидало, пока точка доступа подтвердит получение этого запроса. Если точка доступа в этот момент занята обслуживанием некоторого "скрытого" устройства, она проигнорирует поступивший запрос, и отправившее его устройство поймет, что ему следует ожидать. В противном случае точка доступа подтвердит получение запроса, и устройство получит гарантии, что оно может безопасно начать передачу сообщения. Обратите внимание, что *все* устройства в сети будут слышать *все* подтверждения, отправленные точкой доступа, а значит, иметь четкое представление о том, занята ли точка доступа в данный конкретный момент, даже если данное устройство и не будет слышать передачу данных с другого устройства.

Объединение сетей

Иногда возникает необходимость соединить уже существующие компьютерные сети с целью формирования расширенной системы коммуникаций. Этого можно достичь, например, посредством соединения сетей с образованием более крупной версии компьютерной сети того же "типа". Так, в случае сетей с топологией шины, построенных с использованием протокола Ethernet, часто оказывается возможным объединить их шины и получить одну более длинную шину. Это можно сделать с использованием различных устройств, известных как повторители, мосты и сетевые контроллеры (или свитчи), различия между которыми являются довольно тонкими, но достаточно важными. Простейшими из таких устройств являются **повторители**, которые представляют собой чуть больше чем простые устройства для передачи сигналов туда и обратно между двумя соединяемыми ими шинами (обычно с усилением сигналов в той или иной форме). При этом смысл проходящих сигналов этими устройствами не анализируется (рис. 4.4, *а*).



а) Соединение двух шин повторителем или мостом

б) Объединение трех сетей посредством сетевого коммутатора

Рис. 4.4. Объединение сетей с топологией шины в одну большую сеть

Мост очень похож на повторитель, но является более сложным устройством. Как и повторитель, он предназначен для соединения двух шин, однако он не обязательно пропускает через соединение *все* поступающие сообщения. Вместо этого он анализирует адреса назначения, присвоенные каждому поступившему сообщению, и пересылает через соединение только те, которые предназначены для компьютеров в сети с другой стороны. В результате две машины, расположенные с одной стороны моста, могут обмениваться сообщениями, не оказывая никакого влияния на то, что происходит с другой его стороны. Как видите, мост позволяет создать более эффективную систему в сравнении с простым повторителем.

Сетевой коммутатор (или **свитч**), в сущности, является мостом с несколькими соединениями, что позволяет ему вместо двух объединить в единое целое сразу несколько шин. Следовательно, использование этого устройства позволяет сформировать сеть, состоящую из нескольких шин, связанных между собой подобно спицам в колесе (рис 4.4, *б*). Как и в случае моста, свитч анализирует адрес назначения всех поступающих сообщений и пересылает только те из них, которые адресованы в другие "спицы". Более того, каждое сообщение, которое пересылается, поступает только в соответствующую его адресу сеть, что позволяет минимизировать сетевой трафик во всех "спицах" объединенной сети.

Очень важно отметить, что, когда сети соединяются между собой с использованием повторителей, мостов или сетевых коммутаторов, в результате образуется единая сеть большего размера. Вся система функционирует таким же образом (с использованием тех же самых протоколов), что и каждая из исходных сетей меньшего размера.

Однако иногда предназначенные для объединения сети могут иметь несовместимые характеристики. Например, характеристики сетей Wi-Fi не всегда совместимы с сетями Ethernet. В подобных случаях исходные сети следует объединять в одно целое таким образом, чтобы построить *сеть из сетей*, или **internet**. В этой ситуации и после объединения исходные сети сохраняют свои индивидуальные характеристики и продолжают функционировать как независимые сети. (Обратите внимание на то, что общий термин *internet* отличается от понятия Интернета. Термин "Интернет" (*Internet*), который всегда пишется с прописной буквы И (*I*), ссылается на конкретную глобальную сеть сетей (т.е. в этом смысле тоже internet), — уникальный феномен, речь о котором пойдет в следующих разделах этой главы. Существует множество примеров сетей из сетей, т.е. интернетов меньшего, не глобального размера. В действительности традиционная служба телефонной связи обеспечивалась системой в виде сети из сетей еще задолго до того, как Интернет обрел популярность.

Соединение между различными сетями с образованием сети из сетей обеспечивается устройством с названием **маршрутизатор** (*router*), которое представляет собой особый компьютер, специально сконструированный для пересылки сообщений. Обратите внимание, что задача маршрутизатора отличается от той, которую решают повторители, мосты и сетевые коммутаторы. Суть в том, что маршрутизатор должен обеспечивать связь между различными сетями, в то же время позволяя им полностью сохранять свои уникальные характеристики. В качестве примера на рис. 4.5 представлены две сети Wi-Fi с топологией звезды и сеть Ethernet с топологией шины, соединенные между собой с помощью маршрутизаторов. Когда устройству в одной из сетей Wi-Fi требуется отправить сообщение машине в сети Ethernet, оно сначала отправляет это сообщение точке доступа своей сети. Далее эта точка доступа отправляет

поступившее сообщение связанному с ней маршрутизатору, который пересылает его маршрутизатору сети Ethernet. От маршрутизатора сообщение поступает на машину, соединенную с шиной сети Ethernet, и она пересылает сообщение по указанному адресу в этой сети.

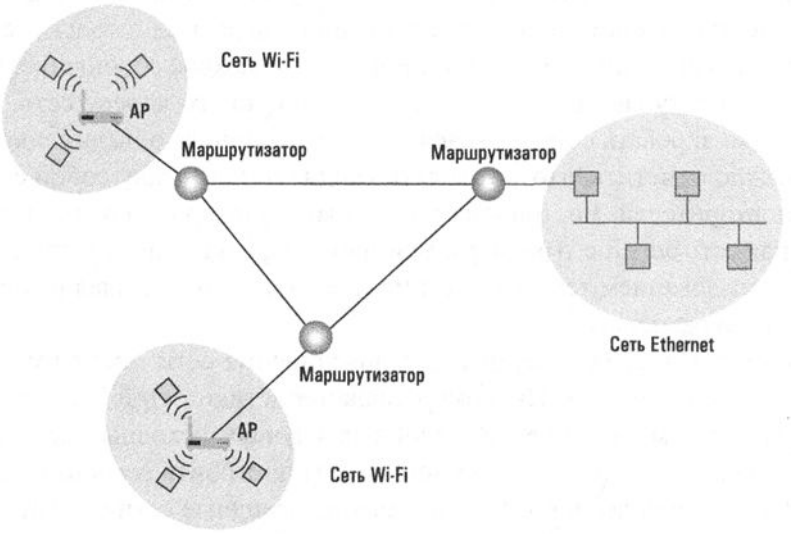


Рис. 4.5. Образование сети из двух сетей Wi-Fi и сети Ethernet с помощью маршрутизаторов

Причина, по которой маршрутизаторы получили свое название, состоит в том, что их назначение — пересылать сообщения в том направлении, которое для них требуется. Этот процесс пересылки строится на системе адресации, охватывающей всю сеть из сетей, в которой всем входящим в объединенную сеть устройствам (включая машины в объединенных сетях и сами маршрутизаторы) присваивается уникальный адрес. (Следовательно, каждое устройство в любой из исходных сетей должно иметь два адреса: его исходный "локальный" адрес в собственной сети и его адрес в сети сетей.) Устройство, отправляющее сообщение на устройство в удаленной сети, должно присвоить ему соответствующий адрес устройства назначения в сети сетей и отослать это сообщение на маршрутизатор своей локальной сети. Оттуда оно будет передано в требуемом направлении. Чтобы успешно осуществлять процесс подобной пересылки, каждый маршрутизатор поддерживает **таблицу маршрутизации**, которая содержит необходимую информацию о направлениях, в которых сообщения должны быть отправлены, в зависимости от указанного в них адреса устройства назначения.

"Точку", в которой некоторая сеть связана с внешней сетью из сетей, часто называют **шлюзом** (gateway), поскольку она служит пропускным каналом между этой сетью и внешним миром. Шлюзы могут быть представлены в

различном виде, поэтому данный термин имеет довольно широкую трактовку. Во многих случаях сетевой шлюз — это просто маршрутизатор, через который она осуществляет взаимодействие с остальной частью объединенной сети. В других случаях термин *шлюз* могут использовать для ссылки на устройство, представляющее собой нечто большее, чем просто маршрутизатор. Например, в большинстве домашних сетей Wi-Fi, подключенных к Интернету, термин *шлюз* применяют одновременно к точке доступа сети и к подключенному к ней маршрутизатору, поскольку эти два устройства обычно размещают в общем корпусе.



Основные положения для запоминания

- **Устройства и сети, из которых состоит Интернет, соединяются между собой и взаимодействуют с использованием адресов и протоколов.**

Методы взаимодействия процессов

Различные действия (или процессы), выполняемые на различных компьютерах в сети (или даже выполняемые на одной и той же машине в режиме разделения времени или мультизадачности), часто должны взаимодействовать между собой с целью координации их действий при решении поставленных перед ними задач. Такое взаимодействие между процессами называют **межпроцессным взаимодействием**.

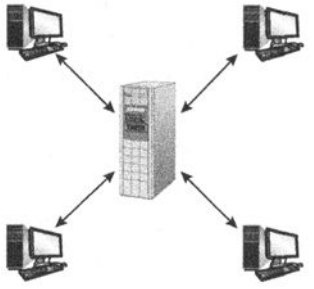
Популярным решением, используемым для реализации межпроцессных взаимодействий, является модель **клиент/сервер**. В этой модели определяются две основные роли, в которых могут выступать взаимодействующие процессы: это либо **клиент**, посылающий запросы другим процессам, либо **сервер**, удовлетворяющий запросы, поступающие от клиентов.

Ранние реализации модели "клиент/сервер" осуществлялись в сетях, объединяющих все компьютеры в пределах офисов. В такой ситуации единственный качественный и дорогостоящий принтер подключался к сети, в результате чего он становился доступным всем входящим в нее компьютерам. В данном случае принтер играл роль сервера (и потому часто назывался *сервером печати*), а всем остальным машинам отводилась роль клиентов, которые отсылали свои запросы на печать на сервер печати.

Другой вариант ранней реализации модели "клиент/сервер" применялся для сокращения расходов на магнитные дисковые накопители, для чего в сети создавалось общедоступное хранилище информации, за счет чего исключалась необходимость хранения многих копий определенной информации, а значит,

сокращалась потребность в объемах используемой в организации дисковой памяти. В этом случае на одной из машин в сети размещалась система массовой памяти большой емкости (как правило, это был жесткий магнитный диск), на которой хранилась вся информация, необходимая для работы организации. Все остальные машины в сети по мере необходимости просто отправляли на нее запросы на предоставление им тех или иных данных. Таким образом, та машина, на которой действительно хранились вся информация, играла роль сервера (так называемый **файл-сервер**), а все остальные компьютеры являлись клиентами, запрашивающими возможность доступа к файлам, постоянно хранившимся на файл-сервере.

В наши дни модель "клиент/сервер" очень широко используется в сетевых приложениях, как будет показано ниже в этой главе. Однако модель "клиент/сервер" — это не единственный способ организации межпроцессного взаимодействия. Другой моделью является **одноранговая**, или **децентрализованная** (*peer-to-peer* — **P2P**), модель. В то время как модель "клиент/сервер" предполагает один процесс (сервер), предоставляющий обслуживание нескольким другим (клиенты), в одноранговой модели подразумевается наличие процессов, которые предоставляют обслуживание один другому и получают обслуживание один от другого (рис. 4.6). Более того, тогда как сервер должен работать постоянно, чтобы быть готовым предоставить клиентам обслуживание в любой момент времени, одноранговая модель обычно подразумевает процессы, которые выполняются на временной основе. Например, к приложениям, использующим одноранговую модель, относятся приложения обмена мгновенными сообщениями, в которых люди ведут письменный разговор через Интернет, а также те, которые позволяют людям играть в интерактивные игры.



а) Сервер в любое время должен быть готов обслуживать многих клиентов



б) Одноранговые пользователи обмениваются друг с другом данными на равных условиях

Рис. 4.6. Модель "клиент/сервер" в сравнении с одноранговой моделью

Одноранговая модель — это также популярное средство распространения через Интернет таких файлов, как музыкальные записи или видеоролики. В этом случае один из участников обмена может получить файл от другого, а затем предоставить другим пользователям возможность получить этот файл от него. Такой подход к распространению файлов полностью противоположен более раннему методу, разработанному с использованием модели "клиент/сервер" и предполагающему создание центра распространения (сервера), с которого клиенты загружают нужные им файлы (или по крайней мере могут отыскать источники для их получения).

Одной из причин, по которым одноранговая модель пришла на смену модели "клиент/сервер" в отношении совместного использования файлов, состоит в том, что она распределяет функцию обслуживания среди множества пар компьютеров вместо того, чтобы возложить ее на один сервер. Такой подход с устранением централизованной схемы обслуживания позволяет строить более эффективные системы. К сожалению, другой причиной популярности системы распространения файлов, построенной с использованием одноранговой модели, является то, что в случае сомнительной законности подобных действий отсутствие центрального сервера существенно усложняет любые юридические усилия по соблюдению законов об авторском праве. Тем не менее известно множество случаев, когда отдельные люди неожиданно обнаруживали, что "сложно" вовсе не означает "невозможно", столкнувшись с серьезными проблемами, связанными с ответственностью за нарушение ими авторских прав.

Возможно, вам уже приходилось слышать или читать об одноранговых *сетях*, что является ярким примером того, к чему может привести неправильное использование терминологии, когда технические термины подхватываются нетехническим сообществом. Термином "одноранговая" определяется система, посредством которой два процесса взаимодействуют через компьютерную сеть. Это понятие вовсе не является характеристикой сети. Любой процесс может использовать одноранговую модель для взаимодействия с одним процессом, а затем воспользоваться моделью "клиент/сервер" для взаимодействия с другим процессом в той же самой сети. Следовательно, более точно будет говорить о взаимодействии по принципам одноранговой модели, а не о взаимодействии в одноранговой сети.

Распределенные системы

По мере достижения все новых и новых успехов в развитии сетевой технологии взаимодействие между компьютерами через сети становилось явлением все более и более распространенным и многогранным. Многие современные программные системы, такие как глобальные информационно-поисковые системы, корпоративные системы финансового планирования, бухгалтерского учета

и управления запасами, компьютерные игры и даже программное обеспечение, управляющее самой сетевой инфраструктурой, разрабатываются как **распределенные системы** в том смысле, что они состоят из программных компонентов, которые выполняются как отдельные процессы на различных компьютерах.

Ранние распределенные системы разрабатывались независимо одна от другой практически "с нуля". Однако на сегодняшний день проведенные исследования выявили наличие общей инфраструктуры, свойственной всем подобным системам, включая такие их аспекты, как взаимодействие процессов и средства защиты. Как следствие были приложены определенные усилия для создания готовых коммерческих систем, предоставляющих такую базовую инфраструктуру, а следовательно, позволяющих создавать собственные распределенные приложения, ограничившись разработкой лишь той их части, которая является уникальной.

Сегодня широкое распространение получили несколько типов распределенных вычислительных систем. **Кластерные вычисления** подразумевают распределенную систему, в которой многие независимые компьютеры тесно сотрудничают для выполнения вычислений или предоставления сервисов, сравнимых с возможностями намного больших компьютеров. Стоимость этих отдельных машин плюс расходы на объединяющую их высокоскоростную сеть могут быть существенно меньше стоимости сверхмощного суперкомпьютера, и при этом обеспечивается большая надежность при меньших издержках на обслуживание. Подобные распределенные системы используются для обеспечения **высокой доступности** (поскольку более вероятно, что хотя бы один компьютер в кластере сможет ответить на запрос, даже если остальные члены этого кластера выйдут из строя или будут недоступны) и **балансировки загрузки** (поскольку рабочая нагрузка может быть автоматически перенесена с тех членов кластера, которые перегружены, на те, которые в данный момент недогружены). Термин **грид-вычисления** (от англ. *grid* — "решетка", "сетка") относится к таким распределенным системам, члены которых менее тесно связаны между собой, чем это имеет место в кластерах, но, тем не менее, также совместно работают над решением больших задач. Проведение грид-вычислений может предполагать использование специализированного программного обеспечения, упрощающего распределение данных и алгоритмов между машинами, входящими в состав этой распределенной сети. Примерами подобных программных систем являются HTCondor Университета штата Висконсин и BOINC — система Open Infrastructure for Network Computing Университета в Беркли. Обе эти системы часто устанавливают на компьютерах, которые изначально были предназначены для других целей, например на рабочих ПК в офисах или на домашних компьютерах, которые затем могут добровольно предоставлять свои вычислительные мощности соответствующей распределенной сети в те периоды, когда они простаивают. Благодаря широкому

распространению Интернета этот тип добровольных распределенных систем позволяет миллионам домашних ПК принимать участие в решении чрезвычайно сложных математических и научных задач. **Облачные вычисления** предполагают выделение вычислительных мощностей громадных пулов совместно используемых компьютеров для нужд клиентов в соответствии с поступающими от них запросами. Этот подход является новейшим направлением в области использования распределенных систем. Во многом подобно тому, как в начале XX века бурное распространение силовых электрических сетей исключило необходимость иметь собственные генераторы на отдельных заводах и других предприятиях, распространение и общедоступность Интернета сделали возможным всем желающим доверить свои данных и вычисления "облаку", под которым в данном случае понимаются громадные вычислительные ресурсы, уже развернутые и доступные в Сети. Такие службы, как Elastic Compute Cloud компании Amazon, позволяют их клиентам брать в аренду на почасовой основе виртуальные компьютеры, не имея при этом ни малейшего представления о том, где в действительности находится это вычислительное оборудование. Сервисы Google Drive и Google Apps позволяют их клиентам совместно иметь доступ к информации или создавать веб-службы без необходимости знать, сколько компьютеров заняты решением поставленной задачи и где именно хранятся те или иные данные. Сервисы облачных вычислений предоставляют обоснованные гарантии в отношении надежности и масштабируемости, но одновременно вызывают озабоченность в отношении сохранения конфиденциальности и обеспечения безопасности в мире, где мы уже не можем знать, кто именно владеет и управляет компьютерами, которые мы используем.

4.1. Вопросы и упражнения

1. Что такое открытая сеть?
2. Поясните, в чем состоит различие между мостом и сетевым коммутатором.
3. Что представляет собой маршрутизатор?
4. Укажите какие-либо отношения в обществе, которые соответствуют концепциям модели "клиент/сервер".
5. Приведите пример протоколов, которым принято следовать в современном обществе.
6. Укажите различия, существующие между кластерными вычислениями и грид-вычислениями.

4.2. Интернет

Самым известным примером сети из сетей является глобальная сеть **Интернет** (*Internet* — обратите внимание на прописную букву I), которая появилась на свет как результат исследовательского проекта, проводившегося еще в начале 1960-х годов. Целью этого проекта было найти возможность связать различные компьютерные сети между собой таким образом, чтобы они могли функционировать как единая система даже в случае значительных локальных повреждений. Значительная часть этих работ финансировалась правительством Соединенных Штатов через агентство DARPA в интересах Министерства обороны США. С течением времени разработка Интернета постепенно сменила свой статус проекта, финансируемого правительством, на статус проекта академических исследований, а сегодня он в большей степени является коммерческим предприятием, которое во всемирном масштабе связывает бесчисленное количество ПВС, ЛВС, МВС и ГВС, охватывающих многие миллионы компьютеров.



Основные положения для запоминания

- **Интернет объединяет устройства и сети по всему миру.**
- **Сети и сетевая инфраструктура поддерживаются как коммерческими, так и правительственными инициативами.**

Архитектура Интернета

Выше уже отмечалось, что Интернет является совокупностью соединенных между собой компьютерных сетей. В общем случае создание и обеспечение работы этих сетей осуществляется организациями, получившими общее название **провайдеры Интернета** (Internet Service Provider — ISP). Очень часто этот термин используют и для ссылок на сами эти сети, поэтому мы говорим о подключении к провайдеру Интернета, в действительности имея в виду подключение к сети, предоставляемое этим провайдером.

Система сетей, контролируемая провайдерами Интернета, может быть классифицирована на иерархической основе в соответствии с той ролью, которую каждая из них играет в общей структуре Интернета (рис. 4.7). На верхнем уровне этой иерархии будут находиться относительно немногочисленные **провайдеры уровня 1**, которые представляют собой чрезвычайно скоростные международные ГВС очень высокой пропускной способности. Совокупность этих сетей можно представить себе как основной костяк Интернета. Обычно они

контролируются крупными компаниями, которые специализируются в коммуникационном бизнесе. Как пример можно привести компанию, которая прежде владела традиционной телефонной сетью или сетью кабельного телевидения, а затем приняла решение расширить свое поле деятельности в направлении предоставления прочих телекоммуникационных услуг.

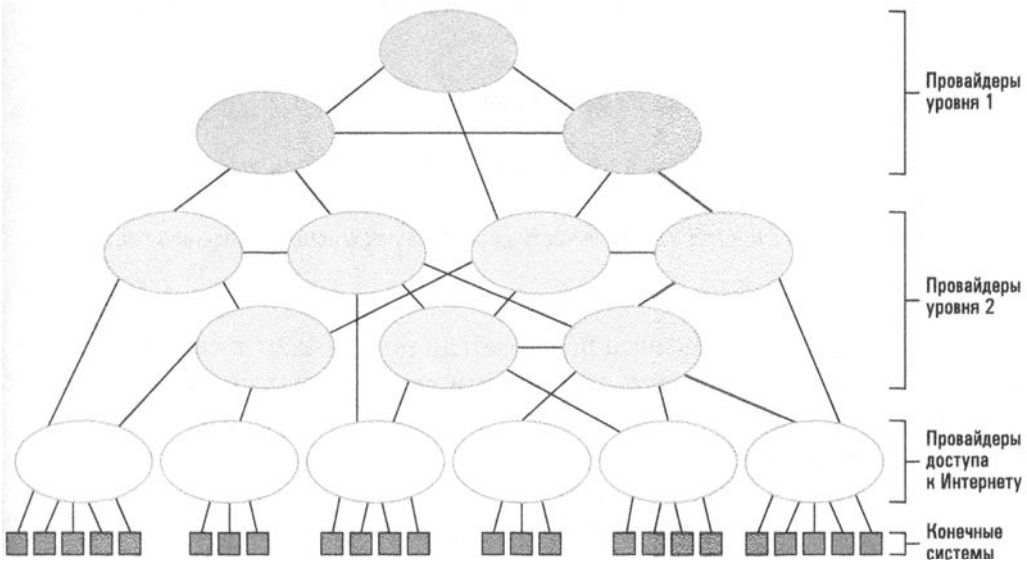


Рис. 4.7. Структура Интернета

К провайдерам уровня 1 подключают свои сети **провайдеры уровня 2**, которые в большинстве случаев представляют региональные структуры и обладают меньшим потенциалом в отношении своих возможностей. (На практике при разделении провайдеров на первый и второй уровни руководствуются по большей части просто личным мнением.) И вновь, эти сети как правило контролируются компаниями, специализирующимися в коммуникационном бизнесе.

Провайдеры первого и второго уровней по большей части представляют собой сети, состоящие из маршрутизаторов, которые в своей совокупности образуют коммуникационную инфраструктуру Интернета. Множество возможных путей прохождения информации между маршрутизаторами придает этому ядру Интернета высокую **надежность** в том смысле, что большинство или даже все сети сохранят соединения между собой даже в том случае, если откажет несколько чрезвычайно важных маршрутизаторов на верхних уровнях. Доступ к этому ядру обычно предоставляется еще одним промежуточным уровнем провайдеров, которых называют **провайдерами 3-го уровня** или **провайдерами доступа к Интернету**. Эти провайдеры по большей части являются независимыми сетями сетей, которые иногда называют **интранетами** (т.е. внутренними

сетями), контролируемыми единственным владельцем, бизнес которого состоит в предоставлении доступа к Интернету конечным домашним и бизнес-пользователям. Примерами являются кабельные и телефонные компании, которые взимают плату за предоставляемые ими услуги, а также организации, подобные университетам или иным учреждениям, которые берут на себя предоставление доступа к Интернету конечным пользователям в пределах их организации.



Основные положения для запоминания

- **Сеть Интернет и построенные на ее основе системы являются иерархическими и весьма надежными.**
- **Иерархическая структура и надежность способствуют масштабированию систем.**

Устройства, которые конечные пользователи подключают к сети провайдера Интернета, принято называть **конечными системами**, **узлами** или **хостами** (*host*). Эти конечные системы могут представлять собой ПК или ноутбуки, но все чаще это понятие распространяется и на множество других устройств, включая телефоны и планшеты, видеокамеры, автомобили и даже различные бытовые устройства. В конечном счете Интернет представляет собой систему *коммуникации*, а следовательно, любое устройство, которое может получить какую-либо пользу от взаимодействия с другими устройствами, потенциально является конечной системой.

Технологии, с помощью которых конечные системы подключаются к более крупным сетям, также могут быть различными. Сейчас наиболее популярным вариантом является, вероятно, беспроводное соединение с использованием технологии Wi-Fi. Используемый в этом случае подход состоит в подключении к сети провайдера Интернета точки доступа, которая, в свою очередь, обеспечивает доступ к этой Сети любым устройствам, находящимся в пределах области слышимости ее радиосигнала. Область слышимости точки доступа или группы точек доступа часто называют **горячей точкой** или **хот-спот** (*hot spot*), особенно в тех случаях, когда доступ к сети открыт для всех или является бесплатным. Горячие точки можно найти в частных домах, отелях, офисных зданиях, заведениях малого бизнеса или парках. В некоторых случаях они могут полностью накрывать целый город. Схожая технология используется в индустрии сотовых телефонов, в которой горячие точки носят название "сота", а "маршрутизаторы", генерирующие сигнал в этих сотах, координируют свои действия с целью обеспечения непрерывности обслуживания конечных систем при их перемещении из одной соты в другую.

Другой популярной технологией для подключения к сетям провайдеров доступа к Интернету является использование телефонных линий либо кабельных или спутниковых систем. Эти технологии могут использоваться для предоставления прямого подключения непосредственно конечной системе либо для подключения маршрутизатора, к которому затем подключается необходимое количество конечных систем. Этот последний прием очень популярен в частных домах и квартирах, где локальная горячая точка создается с помощью комбинированного устройства из маршрутизатора и точки доступа, подключаемого к сети провайдера доступа к Интернету посредством уже существующего кабеля или телефонной линии.

Те аспекты конструктивной структуры Интернета, которые позволяют ему обеспечить возможность взаимодействия столь широкого разнообразия устройств, использующих различные технологии, придает ему **сквозную (end-to-end) архитектуру**. Благодаря очень малому числу встроенных в систему допущений в отношении конечных устройств, Интернет оказался способен быстро развиваться в отношении поддержки новых видов сетей и устройств по мере их появления и распространения.

Инженерный совет Интернета

Инженерный совет Интернета (The Internet Engineering Task Force — IETF) — это открытое международное сообщество проектировщиков, ученых и сетевых провайдеров, созданное во второй половине 1980-х годов с целью разработки и распространения стандартов на протоколы и архитектуру Интернета. Изначально это был проект федерального правительства США, но уже к началу 1990-х IETF превратился в международную организацию, состоящую из добровольцев. Открытые стандарты, благодаря которым Интернет оказался столь успешным начинанием, тем не менее, требуют периодических улучшений, поскольку сеть растет и развивается. Совет IETF следит за строгим соблюдением подхода, строящегося на достижении консенсуса в отношении стандартизации и пересмотра протоколов, регламентирующих принципы именования, адресации, маршрутизации и особенностей работы многих популярных интернет-приложений. Более подробную информацию о IETF и его деятельности можно найти на сайте `https://www.ietf.org/`.



Основные положения для запоминания

- **Сквозная (end-to-end) архитектура Интернета упрощает подключение к нему новых типов устройств и сетей.**

В XX столетии все коммуникационные сети — телефонные, кабельного телевидения, а затем и спутниковые — проектировались для передачи аналоговых сигналов, таких как голос человека или сигналы аналогового телевидения. Современные сети проектируются уже для передачи цифровых данных непосредственно между компьютерами, но более ранняя аналоговая сетевая инфраструктура все еще составляет существенную часть Интернета. Сложности, возникающие в связи с использованием в Сети этих устаревших сетевых соединений, часто в совокупности называют **проблемой последней мили**. Основные артерии ГВС, МВС и многих ЛВС относительно просто модернизировать с использованием высокоскоростных цифровых технологий, таких как оптоволоконные каналы связи, но попытка заменить ими существующие телефонные линии из медных проводов и коаксиальные телевизионные кабели, соединяющие эти артерии непосредственно с конечными пользователями в офисах или местах их проживания, может оказаться намного дороже. В результате информация, поступающая по Интернету на устройство конечного пользователя с других континентов, может почти весь путь молниеносно проделать по цифровому высокоскоростному соединению и только на "последней миле" проталкиваться к конечному устройству по медленной аналоговой телефонной линии столетней давности. Как упоминалось в главе 2, в свое время было разработано несколько хитроумных схем, позволяющих расширить пропускную способность таких устаревших аналоговых соединений для эффективной передачи цифровых данных. Для того чтобы предоставить конечным пользователям возможность широкополосного доступа к Интернету, сейчас используют DSL-модемы, кабельные модемы, спутниковые восходящие линии связи и даже прямое подведение оптоволоконных соединений в их дома и офисы.

Интернет-адресация

Как уже говорилось в разделе 4.1, сеть из сетей нуждается в системе адресации, способной охватить ее во всей полноте и позволяющей присвоить уникальные идентифицирующие их адреса каждому входящему в нее компьютеру. В Интернете эти адреса называют **IP-адресами**. (Термин "IP" является сокращением от *Internet Protocol* — протокол Интернета; с этим понятием вы познакомитесь в разделе 4.4.) Первоначально каждый IP-адрес представлял собой комбинацию из 32 бит, но для того, чтобы расширить диапазон доступных адресов, в настоящее время идет процесс постепенного преобразования этих адресов в 128-разрядные битовые комбинации (см. обсуждение протокола IPv6 в разделе 4.4). Блоки последовательно пронумерованных IP-адресов выдаются для использования провайдерам Интернета **Корпорацией Интернета по управлению доменными именами и IP-адресами** (Internet Corporation for Assigned

Names and Numbers — *ICANN*), которая представляет собой международную некоммерческую организацию, созданную для регулирования вопросов, связанных с функционированием Интернета. Адреса, входящие в предоставленный им блок, провайдеры Интернета могут распределять между устройствами, входящими в область их полномочий, при этом клиентским организациям обычно предоставляется блок последовательных адресов меньшего размера. В результате всем устройствам в Интернете присваиваются уникальные IP-адреса, образующие своеобразную иерархическую структуру.



Основные положения для запоминания

- **Система IP-адресов имеет иерархическую структуру.**
- **Подключение устройств к Интернету осуществляется посредством присвоения им уникальных IP-адресов, соответствующих требованиям протокола Интернета.**

IP-адреса традиционно записывают в **десятичной нотации с точками**, когда каждый байт адреса отделяется точкой, а его значение представляется целым десятичным числом. Например, значение 5.2, записанное в десятичной нотации с точками, представляет двухбайтовую комбинацию битов `0000010100000010`, состоящую из байта `00000101` (представленного числом 5), за которым следует байт `00000010` (представлен числом 2), а значение 17.12.25 будет представлять трехбайтовую комбинацию битов, состоящую из байта `00010001` (число 17 в двоичной нотации), за которым следуют байт `00001100` (12 в двоичной нотации) и байт `00011001` (25 в двоичной нотации). В общем случае 32-разрядный IP-адрес, представленный в десятичной нотации с точками, может выглядеть как 192.207.177.133.

Представление адресов в битовой форме (даже когда они сжаты с использованием десятичной нотации с точками) неудобно для нашего восприятия. Поэтому в Интернете существует еще одна, альтернативная система адресации, в которой машинам присваиваются идентификаторы в виде мнемонических имен. Эта система адресации построена на концепции **домена**, который можно представить себе как "область" в Интернете, контроль над которой осуществляет определенный владелец, такой, как университет, клуб, компания или организация, государственная или общественная. (Слово *область* здесь взято в кавычки по той причине, что, как вы скоро узнаете, эта область может не соответствовать какому-либо физическому региону Интернета.) Каждый домен должен быть зарегистрирован в организации ICANN, — процесс, осуществляемый **компаниями-регистраторами**, которые получили этот статус от ICANN. Как часть процесса регистрации каждому домену присваивается

мнемоническое **доменное имя**, которое должно быть уникальным среди всех доменных имен, существующих в Интернете. Доменные имена часто являются описательными по отношению к той организации, которая зарегистрировала этот домен, что повышает их полезность для людей.

В качестве примера можно привести доменное имя издательской компании Addison Wesley Longman — `awl.com`. Обратите внимание на суффикс `com`, стоящий после точки. Он используется для отражения категории данного домена, который в этом случае является "коммерческим", на что и указывает суффикс `.com`. Подобные суффиксы называют **доменами высшего уровня** (Top-Level Domain — *TLD*). К прочим оригинальным доменам высшего уровня относятся `.edu` для образовательных учреждений, `.gov` — для правительственных органов, `.org` — для неприбыльных организаций, `.mil` —для военных структур и домен `.net`, который исходно предназначался для провайдеров Интернета, но сейчас используется гораздо шире. В первые дни существования Интернета домены высшего уровня по умолчанию определялись для различных типов организаций, существовавших в те дни в США. Однако Интернет очень скоро вышел за пределы этой страны и сейчас охватывает практически все страны мира, что потребовало создания особого набора двухбуквенных доменов верхнего уровня, каждый из которых предполагает его использование в одной конкретной стране (их называют **национальными доменами** верхнего уровня). Примерами таких доменов являются `.au` для Австралии, `.ca` — для Канады, `.ru` — для России и `.ua` — для Украины. Уже в XXI веке в иерархию доменов высшего уровня были добавлены сотни новых значений, таких как `.biz` для бизнес-структур и `.museum` — для музеев.

Как только мнемоническое имя домена будет зарегистрировано, получившая его организация получает право расширять это имя, чтобы создать мнемонические идентификаторы для отдельных элементов внутри своего домена. Например, некоторый узел в домене издательской компании Addison Wesley Longman может получить мнемоническое имя `ssenterprise.awl.com`. Обратите внимание, что имена доменов иерархически расширяются новыми элементами *слева* и при этом разделяются точками. В некоторых случаях многократные расширения, называемые **поддоменами**, используются как средство упорядочения имен в пределах домена. Такие поддомены часто представляют различные сети в пределах юрисдикции домена. Например, если корпорации Yoyodyne было присвоено доменное имя `yoyodyne.com`, то отдельный компьютер в этой компании может получить такое доменное имя, как `overthruster.propulsion.yoyodyne.com`, означающее, что это компьютер `overthruster`, принадлежащий поддомену `propulsion` в домене `yoyodyne`, относящемуся к домену верхнего уровня `.com`. (Следует подчеркнуть, что нотация с точками, используемая в мнемоническом адресе, никак не связана с десятичной нотацией с точками, используемой для представления адресов в виде комбинации битов.)



Основные положения для запоминания

- **Доменные имена имеют иерархическую структуру.**

Хотя мнемонические адреса более удобны для человека, сообщения передаются через Интернет исключительно с использованием IP-адресов. Следовательно, если человек хочет отправить сообщение на удаленное устройство, указанное посредством его мнемонического адреса, программное обеспечение должно иметь возможность преобразовать его в соответствующий IP-адрес, прежде чем сообщение можно будет отправить. Такое преобразование осуществляется с помощью многочисленных серверов, которые принято называть **серверами имен**. По сути, они представляют собой каталоги, предоставляющие услуги перевода адресов клиентам. В своей совокупности эти серверы имен используются как система каталогов, охватывающая весь Интернет и получившая название **система доменных имен** (Domain Name System — *DNS*). Процедура использования системы доменных имен для выполнения трансляции адреса получила название **просмотр DNS-записей**. Иерархическая структура системы доменных имен является важным фактором в ее способности расширяться с ростом количества доменов, DNS-серверов и количества узлов в Интернете. Отдельный узел Интернета не обязан поддерживать или хранить полный глобальный каталог сопоставления всех доменных имен с целью просмотра DNS-записей. Вместо этого узел достаточно лишь изначально настроить так, чтобы он мог найти собственный локальный сервер имен, обычно предоставляемый провайдером доступа к Интернету. DNS-протокол позволяет осуществить просмотр DNS-записей для неизвестного домена начиная от корневого сервера для указанного домена высшего уровня. Так, если узлу необходимо определить IP-адрес еще неизвестного ему узла `lectroid.propulsion.yoyodyne.com`, локальный сервер может начать просмотр с известного ему сервера имен `.com`, который затем перешлет этот запрос на сервер имен `yoyodyne.com`, который, в свою очередь, переправит запрос на сервер имен `propulsion.yoyodyne.com`, который, наконец, авторитетно преобразует указанное мнемоническое имя в соответствующий IP-адрес. На практике кеширование запросов в локальных DNS-серверах позволяет быстро предоставлять ответы на большинство обычных DNS-запросов без необходимости каждый раз обращаться к авторитетным DNS-серверам, что экономит время и позволяет сократить избыточный сетевой трафик.

Следовательно, чтобы некоторое устройство было доступно по его мнемоническому доменному имени, это имя должно быть представлено на сервере имен в системе DNS. В тех случаях, когда создатель домена имеет необходимые ресурсы, он может развернуть и поддерживать собственный сервер имен,

содержащий необходимую информацию обо всех именах, определенных в пределах его домена. В действительности это та модель, на которой изначально строилась вся доменная система. Каждый зарегистрированный домен тогда представлял физическую область Интернета, которая контролировалась локальной властной структурой, такой как компания, университет или правительственная организация. Эта властная структура, в сущности, и являлась провайдером доступа к Интернету, предоставлявшим этот доступ своим сотрудникам через собственную объединенную внутреннюю сеть, имевшую выход в Интернет. Как часть этой глобальной системы данная организация поддерживала собственный сервер имен, обеспечивающий услуги перевода для всех имен, существовавших в пределах ее домена.



Основные положения для запоминания

- **Система доменных имен (DNS) преобразует мнемонические имена в соответствующие IP-адреса.**
- **Иерархическая структура системы доменных имен обеспечивает возможность ее масштабирования.**

Такая модель остается достаточно обычной и настоящее время. Однако многие отдельные лица или небольшие организации хотели бы иметь в Интернете собственный домен, но без привлечения ресурсов, необходимых для его поддержки. Например, для местного шахматного клуба может быть полезно опубликовать в Интернете свой домен под именем `KingsandQueens.org`, но при этом клуб вовсе не заинтересован в развертывании собственной сети, организации ее подключения к Интернету и поддержке собственного сервера имен. В этой ситуации клуб может заключить контракт с местным провайдером доступа к Интернету на создание видимости зарегистрированного домена с использованием ресурсов, уже развернутых этим провайдером. Скорее всего, клуб (возможно, при участии провайдера Интернета) зарегистрирует выбранное им имя домена и заключит с провайдером контракт, по которому последний разместит информацию об этом домене на своем сервере имен. А это будет означать, что все DNS-запросы в отношении нового доменного имени будут перенаправляться на сервер имен провайдера Интернета, где и будут успешно разрешаться. При таком подходе многие зарегистрированные домены окажутся развернутыми в рамках сети единственного провайдера, и при этом каждый из них будет использовать лишь малую часть одного и того же компьютера. Вся эта процедура в наше время может быть осуществлена в считанные минуты, часто всего за несколько долларов, на множестве различных сайтов по всему Интернету.

Интернет-приложения

В первые дни существования Интернета большинство сетевых приложений представляло собой независимые простые программы, каждая из которых следовала требованиям определенного сетевого протокола. Так, приложения для чтения групп новостей связывались с соответствующими серверами с использованием протокола **Network News Transfer Protocol** (NNTP), приложения для просмотра и копирования файлов в сети строились на использовании протокола **File Transfer Protocol** (FTP), тогда как приложения, позволявшие получить доступ к другим компьютерам, находящимся на любых расстояниях, использовали протокол **Telnet**, а позднее — протокол **Secure Shell (SSH)**. По мере того как веб-серверы и браузеры становились все более и более сложными, все больше этих традиционных сетевых приложений уступало свое место веб-страницам, использовавшим для тех же целей мощный и универсальный протокол **Hyper Text Transfer Protocol** (HTTP). Тем не менее, знакомясь с сетевыми протоколами Интернета, полезно будет начать с нескольких более простых примеров и лишь затем перейти к протоколу HTTP, речь о котором пойдет в следующем разделе.

Поколения беспроводной телефонии

Технология мобильной телефонной связи быстро развивалась с момента первого появления в 1980-х годах простых ручных электронных устройств. С того времени очередная волна новых телефонных технологий появлялась на свет примерно каждые десять лет, конечным результатом чего стало появление современных сложных и многофункциональных устройств, получивших название "смартфон". Первое поколение сетей беспроводной телефонии позволяло передавать лишь аналоговые голосовые сигналы, как и обычные телефоны, но им уже не требовались медные провода, связывавшие их с телефонной станцией. В ретроспективе мы можем классифицировать эти ранние телефонные сети как "1G" (от англ. *1 generation*, т.е. *первое поколение*). Сети второго поколения уже использовали для кодирования голосовых сигналов цифровые технологии, что обеспечивало более эффективное использование радиочастот, а также предоставляло возможность передачи других типов цифровых данных, например текстовых сообщений. В телефонных сетях третьего поколения ("3G") была реализована более высокая скорость передачи, позволявшая осуществлять даже телефонные видеозвонки и пользоваться другими услугами, требовавшими более широкополосной связи. Телефонные сети четвертого поколения ("4G") обеспечивают еще более высокую скорость передачи данных, достаточную для полноценной поддержки IP-сетей с коммутацией пакетов, что позволяет владельцам смартфонов пользоваться такой гибкостью в отношении доступа к любой информации в Сети, которая ранее была доступна только пользователям компьютеров с высокоскоростным соединением с Интернетом.

Электронная почта

Сейчас конечным пользователям в Сети доступно множество систем обмена сообщениями: текстовые мессенджеры, интерактивный чат с помощью браузеров, системы обмена краткими "твитами", подобные Твиттеру, или "стена" в Фейсбуке; это лишь несколько примеров. Одной из самых старых и наиболее используемых в Интернете систем является **электронная почта**. Хотя в наше время многие пользователи для работы с электронной почтой предпочитают пользоваться браузерами или достаточно сложными приложениями, подобными *Outlook* компании Microsoft, *Mail* компании Apple или *Thunderbird* корпорации Mozilla, фактическая передача сообщений электронной почты через Интернет от одного компьютера к другому по-прежнему остается той областью, в которой преобладают базовые сетевые протоколы, подобные SMTP.

Протокол **SMTP** (*Simple Mail Transfer Protocol* — простой протокол передачи почтовых сообщений) определяет способ, посредством которого два компьютера в сети могут взаимодействовать между собой при необходимости передать сообщение электронной почты от одного узла к другому. В качестве примера рассмотрим случай, когда **почтовый сервер** `mail.skaro.gov` отправляет сообщение от конечного пользователя "`dalek`" конечному пользователю "`doctor`", входящему в домен `tardis.edu`. Прежде всего, процесс обработки почты `onmail.skaro.gov` устанавливает соединение с процессом почтового сервера на сервере `mail.tardis.edu`. Чтобы решить эту задачу, он использует DNS — другой сетевой протокол, позволяющий сопоставить имя домена назначения, указанное в понятном человеку виде, действительному имени почтового сервера, что позволит установить его IP-адрес. Эта процедура не отличается от обычного поиска номера телефона знакомого человека, перед тем как набрать его номер. Аналогичным образом, когда серверный процесс на другом конце отвечает, протокол требует, чтобы он идентифицировал себя для того, кто инициировал установку соединения с ним. Расшифровка сообщений, которыми эти процессы обменивались согласно протоколу SMTP, может выглядеть приблизительно следующим образом.

```
 1 220 mail.tardis.edu SMTP Sendmail Gallifrey-1.0; Fri, 23
     Aug 2413 14:34:10
 2 HELO mail.skaro.gov
 3 250 mail.tardis.edu Hello mail.skaro.gov, pleased to meet you
 4 MAIL From: dalek@skaro.gov
 5 250 2.1.0 dalek@skaro.gov... Sender ok
 6 RCPT To: doctor@tardis.edu
 7 250 2.1.5 doctor@tardis.edu... Recipient ok
 8 DATA
 9 354 Enter mail, end with "." on a line by itself
10 Subject: Extermination.
11
12 EXTERMINATE!
```

```
13 Regards, Dalek
14 .
15 250 2.0.0 r7NJYAEl028071 Message accepted for delivery
16 QUIT
17 221 2.0.0 mail.tardis.edu closing connection
```

В строке 1 приведено ответное сообщение, которое удаленный почтовый сервер отправил вызывающему процессу. В нем указаны имя сервера, протокол, который он использует, и другая необязательная информация, а именно — версия протокола, дата и время. В строке 2 отправляющий сообщение почтовый сервер просто указал свое имя.

В большинстве интернет-протоколов необязательно предусматривается передача символов в коде ASCII, которые легко читаются человеком. Разумеется, необычайная вежливость фразы `pleased to meet you` (*рад встрече с вами*) в строке 3 ни в коем случае не будет по достоинству оценена программным обеспечением на любом конце этого соединения и не имеет какого-либо значения для правильной работы протокола SMTP. Числового значения "`250`" в начале строки вполне достаточно для того, чтобы подтвердить, что сообщение в предыдущей строке было получено и обмен сообщениями может быть продолжен. Тем не менее отправка всего, что приведено в строке 3, отражает действительное поведение одного из популярных почтовых серверов SMTP, пришедшее к нам из первых дней существования Интернета, когда оператору-человеку частенько приходилось просматривать транскрипции SMTP-сообщений на понятный язык с целью устранения возможной несогласованности в поведении между почтовыми серверами. Бессчетные миллионы подобных обменов происходят в Сети каждый день, о чем известно только программам-посредникам, осуществляющим пересылку электронной почты через Интернет.

В простейшем случае удаленный почтовый сервер выделит только `mail.skaro.gov` из всего сообщения в строке 2, приняв к сведению, что вызывающий сервер является машиной с именем `mail`, расположенной в домене `skaro.gov`. Протокол SMTP является ярким примером протокола, изначально построенного на доверии, чем впоследствии охотно злоупотребили спамеры и другие интернет-злоумышленники. Современным почтовым серверам приходится использовать расширенные версии протокола SMTP или его эквиваленты, чтобы гарантировать, что сообщения электронной почты пересылаются безопасным способом. Более подробно ключевые вопросы сетевой безопасности мы будем обсуждать в последнем разделе этой главы.



Основные положения для запоминания

- **Модель доверия Интернета подразумевает компромиссы.**

Вернемся к нашей расшифровке. В строке 4 отправляющий сообщение сервер извещает о том, что у него есть почтовое сообщение для отправки, и указывает, кто является его отправителем. В строке 5 удаленный сервер подтверждает, что он примет сообщение от этого пользователя в этом домене. В строке 6 отправляющий сервер извещает, кто должен быть получателем сообщения на удаленном сервере. В строке 7 удаленный сервер подтверждает, что примет сообщение электронной почты, предназначенное для указанного пользователя. В строке 8 отправляющий сервер завершает официальную вводную часть и объявляет, что он готов отправить `DATA` (*данные*), т.е. собственно почтовое сообщение. В строке 9 удаленный сервер извещает (с помощью кода 354 в соответствии с протоколом SMTP), что он готов получить указанное почтовое сообщение, и включает в свой ответ текстовую инструкцию, предназначенную для прочтения человеком, — о том, как следует завершить процедуру передачи этого сообщения.

В строках с 10 по 14 отправляющий сервер передает текст сообщения электронной почты для указанного получателя. В строке 15 удаленный сервер извещает о том, что принял его, а в строке 17 — что получил от отправляющего сервера уведомление `QUIT` (*прекратить*), отправленное им в строке 16, и прекращает данный сеанс связи.

В технической документации, описывающей протокол SMTP, определяются все возможные и допустимые этапы сеансов связи, подобных обсуждавшимся выше. Каждое из ключевых слов `HELO`, `MAIL`, `RCPT`, `DATA` и `QUIT` точно определяется с точки зрения того, как их следует отправлять, какие им могут сопутствовать необязательные параметры и как их следует интерпретировать. Аналогичным образом для принимающего сервера четко определяются и подробно описываются все допустимые числовые коды его ответов. Создатели программного обеспечения используют это описание протокола для разработки алгоритмов, позволяющих корректно осуществлять процедуры отправки и получения почтовых сообщений в компьютерных сетях.

Существуют и другие протоколы, определяющие прочие аспекты процессов передачи сообщений электронной почты. Поскольку протокол SMTP изначально был разработан для передачи текстовых сообщений, закодированных в коде ASCII, позднее были разработаны дополнительные протоколы, подобные протоколу **MIME** (*Multipurpose Internet Mail Extensions* — многоцелевые расширения интернет-почты), предназначенному для преобразования данных с кодировкой, отличной от ASCII, в форму, совместимую с протоколом SMTP.

Существует два популярных протокола, предназначенных для получения доступа к почтовым сообщениям, поступившим в адрес пользователя и сохраненным на почтовом сервере. Это протоколы **POP3** (*Post Office Protocol version 3* — протокол почтового отделения, версия 3) и **IMAP** (*Internet Mail Access Protocol* — протокол доступа к интернет-сообщениям). Из этой пары

протокол POP3 является более простым. Пользуясь им, пользователь может переместить (загрузить) адресованные ему почтовые сообщения на свой локальный компьютер, где они затем могут быть прочитаны, сохранены в различных папках, отредактированы и подвергнуты любым прочим манипуляциям, которые потребуются пользователю. Все это выполняется непосредственно на локальной машине пользователя, с использованием массовой памяти этой машины. Протокол IMAP, напротив, обеспечивает пользователю возможность хранить почтовые сообщения и связанные с ними материалы на той же самой машине, где функционирует почтовый сервер, и манипулировать ими. В этом случае пользователи, которым необходимо иметь доступ к своей электронной почте с различных компьютеров, получают возможность хранить все свои сообщения на почтовом сервере, доступ к которому они могут получить с любого удаленного компьютера, за которым этот пользователь будет работать.

Протокол VoIP

Как пример более новых интернет-приложений рассмотрим приложения голосовой связи, использующие протокол **VoIP** (*Voice over Internet Protocol* — протокол передачи голоса по Интернету). Он позволяет использовать инфраструктуру Интернета для организации голосовой связи, подобной обычной телефонии. В своей простейшей форме протокол VoIP предусматривает взаимодействие двух процессов, выполняющихся на различных машинах, с целью передачи аудиоданных по принципам одноранговой модели, — процедура, реализация которой сама по себе не представляет существенных проблем. Однако такие задачи, как инициация и прием входящих вызовов, связывание VoIP-систем с традиционными телефонными системами и предоставление услуг, подобных обращению к экстренным службам по номеру 112, являются проблемами, выходящими за рамки традиционных интернет-приложений. Более того, правительства, которые владеют традиционными телефонными компаниями своей страны, часто рассматривают системы VoIP как угрозу и либо облагают их тяжелыми налогами, либо вообще объявляют незаконными.

Существующие системы VoIP можно разделить на четыре категории, которые конкурируют между собой в отношении популярности среди пользователей. VoIP-системы **программной телефонии** представляют собой программное обеспечение, работающее по принципу одноранговой модели и позволяющее двум или более ПК устанавливать телефонное соединение без привлечения какого-либо дополнительного оборудования, кроме микрофона и громкоговорителя. Примером приложения подобного класса является Skype, которое, помимо прочего, предоставляет своим пользователям возможность устанавливать связь с системами традиционной телефонной связи. Одним из недостатков Skype является то, что это запатентованная система, и следовательно, большая

часть ее внутренней структуры остается закрытой для широкой публики. А это означает, что пользователи приложения Skype должны верить в честность и надежность его программного обеспечения без возможности провести соответствующую проверку со стороны третьих лиц. Например, чтобы принять вызов, пользователь Skype должен оставить свой компьютер подключенным к Интернету и доступным для системы Skype, а это означает, что некоторые из ресурсов этого ПК могут использоваться для поддержки других соединений Skype без извещения об этом владельца данного компьютера, — функция, способная вызвать определенную обеспокоенность.

Ко второй категории VoIP-систем относятся **адаптеры аналоговой телефонии**, представляющие собой устройства, обеспечивающие пользователям возможность подключить свой обычный домашний телефон к сетевой телефонной службе, предоставляемой провайдером Интернета. Эта услуга часто предоставляется в одном пакете с традиционным доступом к Интернету и/или с услугами кабельного цифрового телевидения.

Третий тип VoIP-систем существует в форме встроенных VoIP-телефонов, представляющих собой устройства, заменяющие обычные телефоны и имеющие вид телефона-трубки, подключаемой непосредственно к сети TCP/IP. Такие типы VoIP-телефонов становятся все более и более популярными в больших организациях, где они заменяют традиционные внутренние телефоны, требующие соединений посредством медных телефонных проводов. Их замена VoIP-телефонами, работающими по внутренней сети Ethernet организации, сокращает расходы и предоставляет дополнительные возможности.

И наконец, последние поколения смартфонов позволяют использовать технологию беспроводной VoIP-телефонии. Другими словами, прежние поколения беспроводных телефонов были способны устанавливать связь только с сетью соответствующей телефонной компании с использованием ее протоколов. Доступ к Интернету предоставлялся им через шлюзы, соединяющие сеть компании с Интернетом. Именно в этой точке сигналы с телефонов преобразовывались с учетом требований внешней системы TCP/IP. Однако, поскольку телефонная сеть 4G является сетью, построенной на протоколе IP по всей области ее покрытия, 4G телефон, в сущности, представляет собой просто другой тип широкополосного хост-компьютера в глобальной сети Интернет.

Потоковое мультимедиа в Интернете

В настоящее время громадная доля интернет-трафика используется для передачи аудио- и видеоданных через Интернет в режиме реального времени. Этот метод доставки данных принято называть **потоковым мультимедиа**. Американская развлекательная компания Netflix отправила конечным пользователям

потоковых данных более чем на 4 миллиарда часов вещания только за первых три месяца 2013 года и согласно отчетам отвечает более чем за треть всего Северо-Американского интернет-трафика в 2016 году. В 2017 году службы потокового аудио и видео использовали более 70% пропускной способности Интернета.

На первый взгляд, реализация потокового мультимедиа в Интернете не требует каких-то специальных соглашений. Например, можно полагать, что для развертывания в Интернете радиостанции достаточно просто установить соответствующий сервер, который будет рассылать сообщения с вещаемыми аудиоданными каждому клиенту, который их запросит. Этот метод известен как **N-unicast**. (Говоря точнее, под *unicast*, или однонаправленной (односторонней) передачей данных, понимается, что один отправитель отправляет сообщения одному получателю, тогда как в случае *N-unicast* предполагается, что один отправитель осуществляет однонаправленную передачу в адрес многих получателей.) Подход по принципу N-unicast был опробован на практике, но при этом был выявлен существенный недостаток, выражающийся в значительной нагрузке как на сервер самой станции, так и на серверы его ближайших соседей по Интернету. И действительно, принцип N-unicast вынуждает сервер отправлять индивидуальные сообщения каждому из его клиентов, причем в режиме реального времени, и все эти сообщения должны пересылаться в Интернете через серверы его ближайших соседей.

Большинство альтернатив принципу N-unicast представляют собой попытки облегчить эту проблему. В одном случае обращаются к одноранговой модели, применяемой по принципу, напоминающему системы совместного использования файлов. Идея состоит в том, что как только некий адресат получает данные, он начинает рассылать их тем адресатам, которые все еще ожидают их поступления, в результате чего большая часть нагрузки по рассылке данных адресатам перекладывается с их источника на промежуточные получатели в рамках одноранговых связей.

Другая альтернатива, получившая название **multicast** (групповая передача данных), смещает решение проблемы рассылки данных с серверов на маршрутизаторы в Интернете. При использовании групповой передачи данных сервер передает одно сообщение сразу нескольким клиентам с указанием одного адреса, представляющего эту группу, и полагается на то, что маршрутизаторы в Интернете правильно распознают этот адрес и создадут, а затем разошлют копии данного сообщения всем клиентам в группе. Обратите внимание, что приложения, использующие принцип групповой передачи, требуют, чтобы функциональность маршрутизаторов в Интернете была расширена по отношению к тем обязанностям, которые были возложены на них исходно. Поддержка групповой передачи данных была реализована в небольшом количестве сетей, но

еще не получила распространения на всех конечных пользователей Интернета в глобальном масштабе.

Еще более важным является тот факт, что большинство приложений в этой категории являются сейчас **потоковыми приложениями по запросу**, т.е. их конечный пользователь ожидает, что просмотр или прослушивание медиаданных будет осуществляться в произвольное время по его выбору. И это несколько иная проблема в сравнении с обсуждавшимся выше примером интернет-радиостанции, поскольку каждый конечный пользователь ожидает, что будет иметь возможность начать, приостановить, возобновить или даже повторить воспроизведение фрагмента по своему усмотрению. В этом случае технологии, работающие по принципам *N-unicast* и *multicast*, едва ли окажутся полезными. Каждый поток данных по запросу по необходимости является однонаправленным между сервером, на котором хранятся воспроизводимые данные, и конечным пользователем, который хочет их получить.

Чтобы этот тип потоковой передачи можно было масштабировать до тысяч и даже миллионов одновременно обслуживаемых пользователей, у каждого из которых будет свой личный поток, важное значение имеет репликация медиасодержимого на многие отдельные серверы. Огромный масштаб потоковых медиауслуг вынуждает соответствующие компании использовать **сети доставки контента** (*content delivery networks* — CDN), представляющие собой группы серверов, распределенных в нескольких стратегически важных частях Интернета и предназначенных для потоковой передачи данных ближайшим конечным пользователям, размещенным в сетях по соседству. Во многих случаях машины в сетях доставки контента могут размещаться непосредственно в сети локального провайдера услуг Интернета, что гарантирует клиентам этого провайдера возможность с чрезвычайно высокой скоростью получать потоковое мультимедиа с сервера, который физически находится в ближайшей сети, исключая необходимость потоковой доставки данных с центральных серверов компании. И наконец, сетевая технология, получившая название **anycast** (*отправка данных кому угодно*), позволяет конечному пользователю автоматически подключаться к ближайшему серверу из заранее определенной группы серверов, что делает сети доставки контента еще более эффективными.

Сегодня в Интернете потоковое видео высокого разрешения по запросу охватывает уже значительно больше разнообразных устройств, чем обычные ПК. Обширная категория "Smart-устройств", таких как телевизоры, DVD/Blu-ray-плееры, смартфоны или игровые консоли, имеющих непосредственное подключение к сети TCP/IP, также позволяет своим пользователям выбирать и просматривать контент, доступный во множестве как бесплатных, так и требующих подписки источников видеоконтента.

4.2. Вопросы и упражнения

1. В чем состоит назначение провайдеров первого и второго уровней? Каково назначение провайдеров доступа к Интернету?
2. Что такое DNS?
3. Какую битовую комбинацию представляет значение 3.6.9, представленное в десятичной нотации с точками? Представьте комбинацию битов `0001010100011100` в десятичной нотации с точками.
4. Чем структура мнемонического адреса компьютера в Интернете (например, `overthruster.propulsion.yoyodyne.com`) напоминает традиционную систему записи почтовых адресов? Можно ли обнаружить аналогичную структуру в IP-адресе?
5. Назовите три типа серверов, присутствующих в Интернете, и опишите назначение каждого из них.
6. Какие аспекты сетевых взаимодействий определяются протоколами?
7. Чем использование одноранговой модели и модели групповой передачи данных отличается от использования модели N-unicast в отношении организации радиовещания в Интернете?
8. Какими критериями следует руководствоваться при выборе одного из четырех типов голосовой связи по протоколу VoIP?

4.3. Всемирная паутина: World Wide Web

Начало World Wide Web, или Всемирной паутине, было положено в работах Тима Бернерса-Ли, который первым оценил потенциал комбинации интернет-технологий с концепцией связанных перекрестными ссылками документов, получившей название **гипертекст**. Первая реализация программного обеспечения для веба была представлена им в декабре 1990 года. Хотя этот первый прототип вообще не поддерживал работу с мультимедиа, он включал все ключевые компоненты того, что мы теперь понимаем под названием "World Wide Web": формат гипертекстовых документов, обеспечивающий внедрение в них **гиперссылок** на содержимое других документов; протокол передачи гипертекста по сети и серверный процесс, высылавший станицы гипертекста конечным пользователям по их запросам. С этого скромного начала подсистема WWW быстро развивалась, обеспечив поддержку изображений, аудио- и видеоконтента, и к середине 1990-х годов превратилась в доминирующее приложение Интернета, многократно ускорившее его рост и развитие.

Реализация веба

Пакеты программ, помогающие пользователям работать с гипертекстовыми документами, относятся к одной из двух категорий: **браузеры** и **веб-серверы**. Браузер выполняется на машине пользователя и имеет своей целью получение запрошенных пользователем материалов и представление их пользователю в подобающем виде. К самым распространенным в Интернете браузерам относятся Chrome, Firefox, Safari и Internet Explorer. Веб-сервер функционирует на машине, содержащей те гипертекстовые документы, к которым запрошен доступ. Задача сервера — предоставить доступ к размещенным на его машине документам в соответствии с запросами, поступающими от клиентов (браузеров). К наиболее популярным пакетам программного обеспечения веб-серверов относятся Apache, Microsoft IIS и Nginx. Как правило, гипертекстовые документы пересылаются между браузерами и веб-серверами с использованием протокола, получившего название "HTTP" (*HyperText Transfer Protocol* — протокол передачи гипертекста).

Чтобы в вебе можно было находить и извлекать документы, каждый из них получает уникальный адрес, называемый **URL** (*Uniform Resource Locator* — единообразный определитель местонахождения ресурса). Каждый URL-адрес содержит информацию, необходимую браузеру для установления контакта с нужным сервером и выдачи запроса на получение требуемого документа. Следовательно, чтобы увидеть веб-страницу, пользователю необходимо указать в браузере URL-адрес требуемого документа, а затем выдать браузеру команду загрузить и отобразить этот документ.

Структура типичного URL-адреса представлена на рис. 4.8. Он состоит из четырех сегментов: названия протокола, используемого для взаимодействия с сервером, управляющим доступом к документу; мнемонического адреса узла, на котором находится этот сервер; пути доступа к каталогу, необходимого для того, чтобы сервер мог найти каталог, содержащий документ; и собственно имени требуемого документа. Короче говоря, URL-адрес, представленный на рис. 4.8, указывает браузеру на необходимость установить соединение с веб-сервером, размещенным на веб-узле с адресом `eagle.mu.edu`, воспользовавшись для этого протоколом `HTTP`, а затем загрузить документ с именем `Julius_Caesar.html`, который хранится на этом узле в подкаталоге `Shakespeare`, содержащемся в каталоге `authors`.

Иногда URL-адрес не содержит явным образом все сегменты, представленные на рис. 4.8. Например, если сервер не нуждается в пути доступа к каталогу, чтобы получить доступ к некоторому документу, то часть адреса, представляющая этот путь, будет отсутствовать в URL этого документа. Более того, иногда URL-адрес может включать только протокол и мнемонический адрес

компьютера. В подобных случаях веб-сервер этого компьютера возвращает заранее определенный документ, обычно называемый **домашней страницей** (*home page*), в котором кратко описывается информация, доступная на этом веб-сайте. Такие сокращенные URL-адреса представляют собой простое средство установления контакта с организациями. Например, URL-адрес `http://www.google.com` указывает на домашнюю страницу корпорации Google, на которой содержатся ссылки на многие сервисы, продукты и документы, связанные с этой компанией и ее деятельностью.

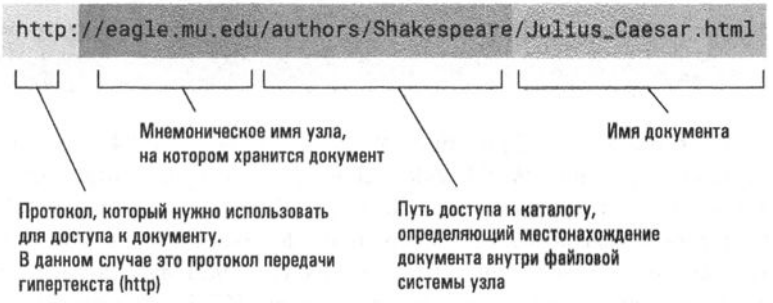


Рис. 4.8. Общий вид и структура типичного URL-адреса

Чтобы еще более упростить доступ к веб-сайтам, многие браузеры, когда протокол не указан явно, по умолчанию предполагают использование в URL-адресах протокола HTTP. Такие браузеры корректно обратятся к домашней странице Google, даже если пользователь укажет URL-адрес просто как `www.google.com`.

Язык гипертекстовой разметки HTML

Традиционный гипертекстовый документ похож на обычный текстовый файл, поскольку его содержание точно так, символ за символом, закодировано с использованием таблицы символов стандарта ASCII или Unicode. Различие состоит лишь в том, что гипертекстовый документ дополнительно содержит специальные маркеры, называемые **тегами**, которые детально описывают, как этот документ должен выглядеть на экране дисплея, какие ресурсы мультимедиа (например, изображения) должны присутствовать в отображаемом документе и какие элементы этого документа должны быть связаны с другими документами. Данная система маркеров-тегов получила название "язык **HTML**" (*Hyper Text Markup Language* — язык разметки гипертекста).

Таким образом, при создании веб-страницы автор, пользуясь соответствующими средствами языка HTML, дополнительно помещает в нее информацию, необходимую браузеру клиента для корректного представления документа на

экране пользователя, а также для отыскания любых связанных документов, ссылки на которые присутствуют на этой веб-странице. Эта процедура аналогична добавлению необходимых указаний на рукопись набираемого текста (возможно, с использованием красной ручки), чтобы наборщик знал, как этот материал должен выглядеть в его конечной форме. В случае гипертекста красные пометки на листе бумаги заменяются тегами HTML, а браузер в этом случае играет роль наборщика, анализируя теги языка HTML, чтобы узнать, как текст должен быть представлен на экране пользователя.

Консорциум World Wide Web

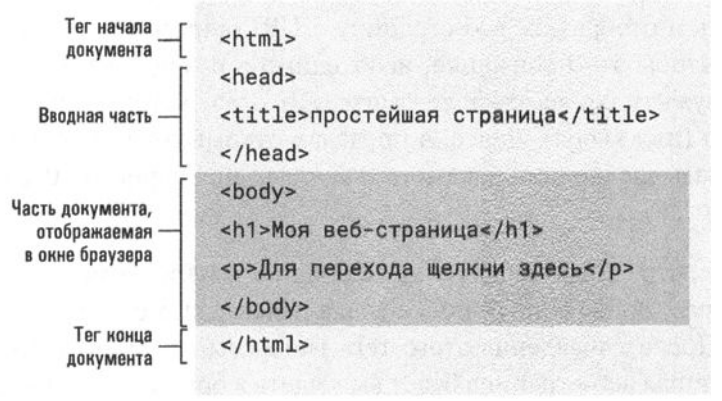
Консорциум World Wide Web Consortium (W3C) был создан в 1994 году с целью содействия распространению World Wide Web посредством разработки стандартных протоколов (известных как стандарты W3C). Штаб консорциума W3C размещается в научном центре CERN, лаборатории физики частиц высоких энергий, расположенном в городе Женева, Швейцария. Именно в лаборатории CERN первоначально разработан язык разметки документов HTML, а также протокол HTTP — протокол передачи HTML-документов по Интернету. Сегодня консорциум W3C является инициатором разработки многих стандартов (включая стандарты на язык XML и многочисленные приложения мультимедиа), предназначенных для обеспечения совместимости в широком диапазоне интернет-продуктов. Чтобы больше узнать о деятельности консорциума W3C, посетите его веб-сайт по адресу `http://www.w3c.org`.

Текст (называемый **исходным** текстом) исключительно простой веб-страницы, представленный на языке HTML, показан на рис. 4.9, *а*. Обратите внимание, что теги этого языка всегда заключаются в пару символов "<" и ">". Исходный текст данного HTML-документа состоит из двух разделов — *заголовка* (находится между тегами `<head>` и `</head>`) и *тела* (находится между тегами `<body>` и `</body>`). Различие между заголовком и телом веб-страницы подобно различию между шапкой и остальным содержанием любого офисного документа. В обоих случаях первая часть содержит общую информацию о документе (дата, кому, от кого и т.д.), а вторая, основная, часть включает собственно содержание документа, которое в случае веб-страницы является именно тем материалом, который будет выведен в окне браузера пользователя при отображении этой страницы.

Заголовок веб-страницы, представленной на рис. 4.9, *а*, содержит только название этого документа (между открывающим и закрывающим тегами "`title`"). Это название предназначено исключительно для целей документирования, оно

не является частью страницы, отображаемой для пользователя в его браузере, хотя некоторые современные браузеры могут поместить это название в верхней части своего окна или вывести его на корешок вкладки. То содержимое, которое отображается в окне браузера, всегда находится в теле документа.

а) Исходный текст веб-страницы на языке HTML



б) Та же веб-страница, отображаемая браузером пользователя



Рис. 4.9. Простейшая веб-страница

На рис. 4.9, *а* первым элементом тела документа является заголовок первого уровня (заключенный между тегами `<h1>` и `</h1>`), содержащий текст Моя веб-страница. То, что это заголовок уровня 1, означает, что на экране браузер должен отобразить этот текст как можно заметнее. Следующим элементом тела документа является абзац обычного текста (заключенный между тегами `<p>` и `</p>`), содержащий слова Для перехода щелкни здесь. На рис. 4.9, *б* показано, как эта страница будет отображена браузером на экране монитора.

В представленном виде веб-страница на рис 4.9 не является полностью функциональной в том смысле, что, если пользователь щелкнет мышью на слове здесь, ровным счетом ничего не произойдет, хотя в отображаемом тексте указано, что в результате должен быть выполнен переход на другую веб-страницу. Чтобы указанное действие имело место в действительности, следует слово здесь связать с ссылкой на другой документ.

Предположим, что после щелчка мышью на слове здесь браузер должен будет загрузить и отобразить веб-страницу с URL-адресом `http://crafty.com/demo.html`. Чтобы это произошло, необходимо в исходной версии нашего документа окружить слово здесь тегами `<a>` и `</a>`, которые являются тегами гиперссылки (или якоря). Далее, в пределах открывающего тега гиперссылки следует указать следующий параметр (как показано на рис. 4.10, *а*).

```
href="http://crafty.com/demo.html"
```

Он указывает, что гиперссылка (`href`), связанная с этим тегом, представляет собой URL-адрес, приведенный после знака равенства, т.е. `http://crafty.com/demo.html`. После добавления этого тега гиперссылки на исходную страницу наша простейшая веб-страница будет выглядеть в браузере так, как показано на рис. 4.10, *б*. Обратите внимание, что ее вид практически идентичен прежнему варианту, представленному на рис. 4.9, *б*, за исключением того, что слово здесь в данном случае выделено другим цветом, указывая на то, что оно представляет собой ссылку на другую веб-страницу. Щелчок мышью на таком выделенном слове потребует от браузера загрузить и отобразить ту веб-страницу, которая указана в данной ссылке. Таким образом, именно посредством тегов гиперссылок веб-страницы связываются между собой.

И наконец, будет полезно показать, как на нашу простейшую веб-страницу можно поместить какое-либо изображение. С этой целью предположим, что изображение, которое требуется поместить на страницу, сохранено в формате JPEG в файле с именем `OurPic.jpg` в каталоге `Images` на сайте `Images.com`, и доступ к нему можно получить, обратившись к веб-серверу этого сайта. Принимая все это во внимание, мы можем указать браузеру, что он должен вывести данное изображение в верхней части нашей веб-страницы, просто поместив в ее исходный HTML-код тег вставки изображения `image`, имеющий вид `<img src="http://Images.com/Images/OurPic.jpg">`, сразу же после тега `<body>`. Этот тег указывает браузеру, что в начале отображаемого документа следует поместить изображение, сохраняемое в файле с именем `OurPic.jpg`. (Здесь терм `src` является сокращением от слова “source”, т.е. *источник*, и означает, что текст, следующий после знака равенства, определяет источник, где можно будет найти требуемое изображение.) Когда браузер обнаружит этот тег, он отправит сообщение на HTTP-сервер сайта `Images.com` с требованием переслать

ему файл изображения с именем `OurPic.jpg`, а затем отобразит его соответствующим образом.

а) Исходный текст веб-страницы на языке HTML

```
<html>
<head>
<title>простейшая страница</title>
</head>
<body>
<h1>Моя веб-страница</h1>
<p>Для перехода щелкни
   <a href="http://crafty.com/demo.html">
   здесь
   </a>
   </p>
</body>
</html>
```

Тег гиперссылки, содержащий параметр

Закрывающий тег гиперссылки

б) Та же веб-страница, отображаемая браузером пользователя



Рис. 4.10. Улучшенный вариант простейшей веб-страницы

Если тег вывода изображения `image` переместить в конец документа, непосредственно перед тегом `</body>`, браузер выведет изображение в нижней части веб-страницы. Безусловно, существуют и более сложные методы размещения изображений на веб-страницах, но сейчас это не является для нас предметом рассмотрения.

Язык XML

Язык HTML, в сущности, представляет собой систему нотации, посредством которой содержимое текстового документа и информация о том, как он должен выглядеть, может быть закодирована в виде простого текстового файла. Сходным образом мы можем закодировать в виде текстовых файлов и некоторый нетекстовый материал, например музыку, записанную в виде нот. На первый взгляд, комбинации из линий нотного стана, тактовых линий и нотных знаков, с помощью которых традиционно представляют музыкальные записи, никак не соответствуют формату "символ за символом", характерному для текстовых файлов. Однако мы можем обойти это затруднение, разработав альтернативную систему нотации. Говоря точнее, мы можем договориться представлять начало нотного стана со скрипичным ключом посредством записи `<стан ключ = "скрипичный">`, а конец нотного стана — записью `</стан>` для указания размера использовать формат `<размер> 2/4 </размер>`, а начало и конец каждого такта задавать в виде `<такт>` и `</такт>` соответственно. Запись нот, например одной восьмой ноты "до", мы будем выполнять в виде `<ноты> восьмая до </ноты>`, и т.д. В этом случае текстовый фрагмент

```
<стан ключ = "скрипичный"> <тональность>до минор</тональность>
<размер> 2/4 </размер>
<такт> <пауза> восьмая </пауза> <ноты> восьмая соль,
восьмая соль, восьмая соль </ноты></такт>
<такт> <ноты> половина ми </ноты></такт>
</стан>
```

можно будет использовать для кодирования музыкального фрагмента, представленного на рис. 4.11. Пользуясь этой нотацией, страницы с музыкальными записями можно будет кодировать, модифицировать, сохранять и передавать через Интернет в виде текстовых файлов! Более того, можно написать программы, которые будут способны представлять содержимое подобных файлов в виде обычных нотных записей или даже воспроизводить записанную в них музыку на синтезаторе.
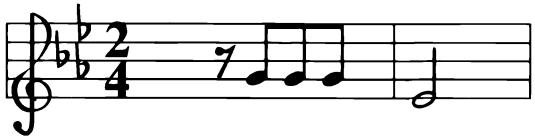


Рис. 4.11. Первые два такта пятой симфонии Бетховена

Обратите внимание, что в нашей системе кодирования нотной записи применяется тот же стиль, который используется в языке HTML. Для выделения тегов, идентифицирующих отдельные компоненты музыкальной записи, мы

выбрали символы "<" и ">". Мы решили определять начало и конец различных структурных элементов нотной записи (таких, как нотный стан, последовательность нот или такт) с помощью тегов с одним и тем же названием, но конечный тег при этом всегда содержит дополнительную косую черту (конец тега `<такт>` определяется тегом `</такт>`). И мы решили задавать значения специальных атрибутов в пределах тегов выражениями вида `ключ = "скрипичный"`. Тот же самый стиль можно будет использовать и при разработке систем для представления любых других форматов, например математических выражений или графических изображений.

Язык **XML** (*eXtensible Markup Language* — расширяемый язык разметки) представляет собой стандартизованный стиль (подобный тому, который мы выбрали выше в музыкальном примере) для разработки нотационных систем представления данных в виде текстовых файлов. (На самом деле язык XML представляет собой упрощенный вариант семейства стандартов более высокого уровня, известных как SGML (*Standard Generalized Markup Language* — стандартный обобщенный язык разметки). В соответствии со стандартом XML были разработаны системы нотации, получившие название **языки разметки**, для представления математических выражений, презентаций мультимедиа и музыки. В действительности язык HTML представляет собой аналогичный язык разметки, основанный на стандарте XML, который был разработан для текстового представления веб-страниц. (Исторически оригинальная версия языка HTML была создана еще до утверждения стандарта XML, в результате чего некоторые средства языка HTML не вполне точно соответствуют требованиям XML. Именно по этой причине вам могут встретиться ссылки на язык XHTML, который является диалектом языка HTML, но при этом строго соответствует всем положениям XML.)

Язык XML являет собой хороший пример того, как могут разрабатываться стандарты широкого применения. Вместо создания отдельных, не связанных между собой языков разметки для кодирования различных типов документов, представленный языком XML подход заключается в разработке стандарта для языков разметки в целом. При наличии такого стандарта можно разработать языки разметки для самых разных применений. При этом все разработанные в такой манере языки разметки будут обладать унифицированным строением, что позволит комбинировать их для получения языков разметки, предназначенных для создания комплексных приложений, таких как текстовые документы, содержащие фрагменты нотной записи или математические выражения.

И наконец, следует отметить, что язык XML позволяет разрабатывать новые языки разметки, которые будут отличаться от HTML прежде всего в отношении семантики, а не своим внешним видом. Например, средствами языка HTML ингредиенты в некотором рецепте могут быть размечены маркерами так, что будут

отображаться в виде списка, в котором каждый ингредиент будет представлен отдельной строкой. Но если воспользоваться семантически ориентированными тегами, то ингредиенты в рецепте можно будет маркировать именно как ингредиенты (скажем, с использованием тегов `<ingredient>` и `</ingredient>`), вместо того чтобы просто представить их элементами списка. Различие здесь тонкое, но очень важное. Семантический подход позволит **поисковым машинам** (так называют веб-сайты, которые предоставляют пользователям возможность находить в вебе различные материалы в соответствии с их интересами) идентифицировать рецепты, которые содержат или не содержат определенные *ингредиенты*, что можно рассматривать как существенное улучшение в сравнении с нынешним состоянием дел, когда можно осуществлять поиск рецептов, которые содержат или не содержат лишь определенные *слова*. Иными словами, это положение можно пояснить так: если бы имелась возможность использовать семантические теги, то поисковые машины могли бы осуществлять поиск рецептов лазаньи, в которых нет шпината, тогда как аналогичный поиск, основанный лишь на анализе слов, вероятнее всего, отбросит рецепт, который будет начинаться с фразы "Этот рецепт *лазаньи* не содержит *шпината*", поскольку здесь есть оба слова, совместное присутствие которых в рецепте не допускается по условию поиска. В свою очередь, использование действующих в масштабах всего Интернета стандартов для разметки документов в соответствии с их семантикой, а не внешним представлением, позволит создать *семантический* веб (World Wide Semantic Web) вместо *синтаксического* веба (World Wide Syntactic Web), который мы имеем на сегодняшний день.

Действия на стороне клиента и на стороне сервера

А теперь давайте рассмотрим те действия, которые браузеру потребуется выполнить, чтобы загрузить и отобразить в своем окне простейшую веб-страницу, представленную на рис. 4.10. Прежде всего, в соответствии с ролью клиента браузер должен воспользоваться информацией, содержащейся в URL-адресе (возможно, полученном от другого пользователя с помощью браузера), чтобы установить контакт с веб-сервером, управляющим доступом к этой веб-странице, и запросить у него копию этой страницы. Сервер выполнит запрос, отправив браузеру текстовый документ, представленный на рис. 4.10, *а*. Далее браузер интерпретирует содержащиеся в документе HTML-теги, чтобы установить, как следует отображать эту страницу, после чего соответствующим образом выводит ее в своем окне на экране монитора. В результате пользователь браузера видит ту же картину, которая показана на рис. 4.10, *б*. Если пользователь в окне браузера щелкнет мышью на слове здесь, браузер воспользуется URL-адресом, указанным в теге гиперссылки, связанном с этим словом, и установит

соединение с соответствующим сервером для получения и отображения следующей веб-страницы. В целом, можно считать, что весь процесс — это просто извлечение и отображение браузером веб-страниц по указанию пользователя.

Но что произойдет, если будет загружена веб-страница, содержащая анимацию, или страница, на которой пользователю будет предложено заполнить некоторую форму и отправить ее на сайт отправителя? Эти ситуации потребуют дополнительных действий либо со стороны браузера, либо со стороны веб-сервера. Эти дополнительные действия называют действиями **на стороне клиента**, если они выполняются клиентом (таким, как браузер), или действиями **на стороне сервера**, если они выполняются сервером (таким, как веб-сервер).

В качестве примера предположим, что туристическому агентству необходимо, чтобы клиент имел возможность указать желаемое направление и дату поездки, после чего ему будет отправлена соответствующим образом настроенная веб-страница, содержащая только ту информацию, которая отвечает намерениям этого клиента. В данном случае веб-сайт туристического агентства должен прежде всего направить посетителю страницу, на которой будут указаны все возможные направления туристических поездок. Исходя из этой информации, клиент сможет указать интересующие его направления, а также желаемые дату и продолжительность поездки (действия на стороне клиента). Затем эту информацию нужно будет отправить в обратном направлении, на сервер туристического агентства, где она будет использована для формирования соответствующим образом настроенной веб-страницы (действия на стороне сервера), которая в конечном счете и будет отправлена на браузер клиента.

Другой интересный пример — это использование сервисов, предоставляемых поисковыми машинами. В этом случае пользователь, выступающий в роли клиента, определяет интересующую его тему (деятельность на стороне клиента), и эта информация пересылается на сайт поисковой машины, где формируется, а затем отсылается пользователю соответствующим образом настроенная веб-страница, содержащая ссылки на документы, которые, возможно, могут представлять для него интерес (действия на стороне сервера). Еще один пример совсем другого рода — это электронная почта с веб-интерфейсом, сервис, популярность которого с течением времени неуклонно возрастает, поскольку с его помощью владельцы компьютеров и портативных устройств могут получать доступ к своей электронной почте через браузер. В этом случае веб-сервер выступает как посредник между клиентом и его почтовым сервером. В данной ситуации веб-сервер создает веб-страницы, которые содержат информацию от почтового сервера (действия на стороне сервера), а затем отсылает эти страницы на устройство клиента, где его браузер отображает их (действия на стороне клиента). И наоборот, браузер предоставляет пользователю возможность создавать почтовые сообщения (действия на стороне клиента), а затем пересылает

их на веб-сервер, который далее отправляет эти сообщения на почтовый сервер (действия на стороне сервера) для рассылки адресатам.

Существует множество систем для реализации действий на стороне клиента и на стороне сервера, которые активно конкурируют. Один из ранних, но все еще популярный способ реализации действий на стороне клиента состоит во включении в исходный HTML-документ веб-страницы программ, написанных на языке JavaScript (создан корпорацией Netscape Communications). Браузер может извлечь этот код из текста страницы и выполнить все определяемые в нем действия. Другой подход (разработанный корпорацией Sun Microsystems) состоит в следующем: сначала браузеру клиента отсылается текст веб-страницы, а затем пересылаются дополнительные программные компоненты (написанные на языке Java), получившие название *аплеты*, — по мере поступления соответствующих запросов от браузера, обрабатывающего исходный HTML-документ. Совершенно другой подход используется в системе Flash (разработанной компанией Macromedia), позволяющей организовать на стороне клиента выполнение пространных презентаций мультимедиа.

Изначальный подход к реализации действий, выполняемых на стороне сервера, предполагал использование набора стандартов, получившего название CGI (*Common Gateway Interface* — общий интерфейс шлюза). С его помощью пользователь мог потребовать выполнения предварительно сохраненных на сервере программ. Вариантом такого подхода (разработка компании Sun Microsystems) является предоставление клиентам возможности вызывать на выполнение на стороне сервера программные элементы, называемые *сервлетами*. Упрощенная версия метода использования сервлетов применяется, когда запрошенные на стороне сервера действия заключаются в построении динамически настраиваемой веб-страницы, как в случае приведенного выше примера с туристическим агентством. В этой ситуации шаблоны таких веб-страниц, называемые JavaServer Pages (JSP), сохраняются на веб-сервере и просто заполняются полученной от клиента информацией с целью получения готовой, динамически настроенной веб-страницы, отправляемой клиенту. Аналогичный подход использован и в технологии корпорации Microsoft, в которой шаблоны, предназначенные для создания динамически настраиваемых веб-страниц, называют Active Server Pages (ASP; позднее — ASP.NET). В противоположность этим запатентованным технологиям система PHP (исходно понимаемая как *Personal Home Page* — личная домашняя страница, а сейчас расширенная до *PHP Hypertext Preprocessor* — препроцессор гипертекста PHP) является открытой системой, предназначенной для реализации действий на стороне сервера. К широко распространенным языкам сценариев серверной стороны можно отнести Python, Ruby, Perl и многие другие.

И наконец, мы просто не имеем права обойти молчанием проблемы безопасности и этические проблемы, возникающие из-за того, что клиентам и серверам разрешено запускать программы на чужой машине. Тот факт, что веб-серверы регулярно отсылают программы клиентам, на которых они и выполняются, приводит к появлению этических проблем на стороне сервера и проблем безопасности на стороне клиента. Если клиент будет слепо выполнять любую программу, отправленную ему веб-сервером, он будет полностью открыт для злонамеренных действий со стороны сервера. Аналогичным образом тот факт, что клиент может вызвать выполнение программ на стороне сервера, ведет к возникновению этических проблем на стороне клиента и проблем безопасности на стороне сервера. Если сервер будет слепо выполнять любую программу, отправленную ему со стороны клиента, это может стать причиной появления брешей в системе защиты и потенциальных разрушений на сервере.

4.4. Вопросы и упражнения

1. Что такое URL-адрес? Что такое браузер?
2. Что такое язык разметки?
3. В чем состоит различие между HTML и XML?
4. Каково назначение каждого из приведенных ниже тегов языка HTML?
   а. `<html>`
   б. `<head>`
   в. `</p>`
   г. `</a>`
5. Что определяют термины *сторона клиента* и *сторона сервера*?

4.4. Протоколы Интернета

В этом разделе мы рассмотрим, как сообщения передаются через Интернет. Этот процесс передачи требует скоординированных усилий от всех компьютеров в системе, поэтому программное обеспечение для управления данным процессом имеется на всех без исключения компьютерах и устройствах, подключенных к Интернету. Поэтому обсуждение мы начнем с ознакомления с общей структурой этого программного обеспечения.

Многоуровневый подход к организации сетевого ПО

Главной задачей сетевого программного обеспечения является предоставление инфраструктуры, необходимой для передачи сообщений от одной машины к другой. В Интернете задача передачи сообщений решается посредством использования некоторой иерархии программных элементов, которая в совокупности поэтапно решает задачу, подобную той, которую требуется решить при отправке посылки с подарком из одного города в другой, расположенный на другом краю страны (рис. 4.12). Прежде всего нужно будет упаковать подарок и написать на пакете адрес получателя. Затем посылку нужно будет отнести в почтовое отделение и отправить авиапочтой на адрес получателя. Почтовая служба поместит эту посылку в большой контейнер вместе с другими пакетами, отправляемыми авиапочтой, и доставит его в аэропорт. Служащие аэропорта погрузят контейнер в самолет, который отправится в нужный город, возможно, с промежуточными остановками по пути. В пункте назначения служащие аэропорта выгрузят контейнер из самолета и отправят его на сортировочный пункт почтовой службы. Наконец работники почтовой службы извлекут посылку из контейнера и доставят ее адресату.

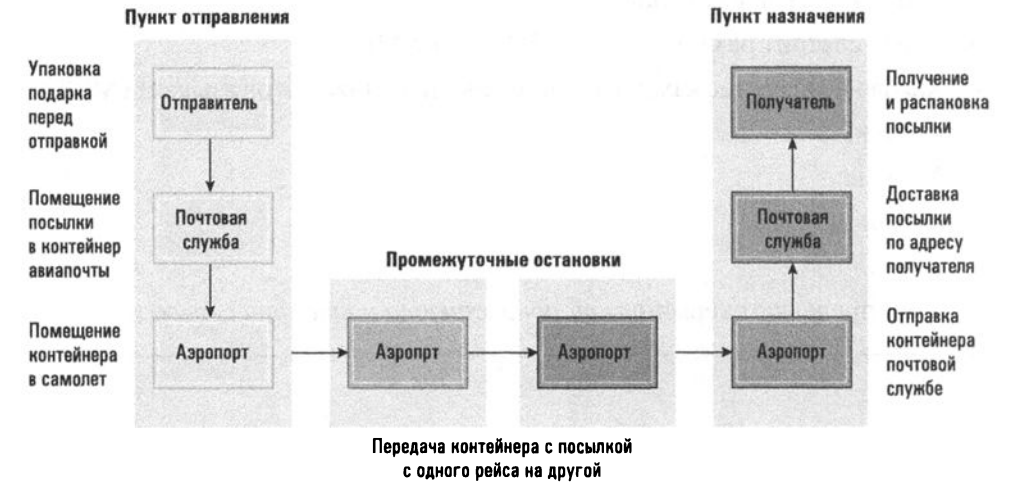


Рис. 4.12. Пример доставки посылки с подарком

Говоря коротко, транспортировка почтовых отправлений осуществляется с помощью трехуровневой иерархии служб доставки. Первый уровень — это уровень пользователя, он состоит из отправителя и получателя. Второй уровень представлен почтовой службой, а третий — авиалиниями. Каждый из уровней рассматривает следующий более низкий уровень как некий абстрактный инструмент. (Отправитель не вникает в детали работы почтовой службы, а эта служба

не интересуется внутренними делами авиалиний.) На каждом уровне этой иерархии имеются и отправители, и получатели, причем действия получателей противоположны действиям соответствующих отправителей.

Сходным образом организовано и программное обеспечение, управляющее взаимодействиями через Интернет, только в нем имеется четыре уровня вместо трех, и каждый уровень представляет собой набор программ, а не людей или организаций. Четыре уровня программного обеспечения Интернета, которые носят названия **прикладной**, **транспортный**, **сетевой** и **канальный**, представлены на рис. 4.13. Сообщение обычно генерируется на прикладном уровне. Отсюда оно передается вниз через транспортный и сетевой уровни, где осуществляется его подготовка к отправке, после чего оно передается на канальный уровень, где и осуществляется его пересылка по назначению. Доставленное сообщение с канального уровня получателя последовательно поднимается по тем же уровням иерархии, пока не достигнет прикладного уровня на машине получателя.



Рис. 4.13. Уровни сетевого программного обеспечения в Интернете

Давайте ближе познакомимся с этим процессом, проследив за тем, как сообщение проходит свой путь через Интернет от пункта отправления до пункта назначения (рис. 4.14). Свое путешествие оно начинает на прикладном уровне.

Прикладной уровень образуют программы приложений, таких как клиент (например, браузер) или сервер (например, почтовый или веб-сервер), использующие взаимодействие через Интернет для решения стоящих перед ними задач. Хотя название и похоже, этот уровень вовсе не ограничивается прикладным программным обеспечением, как оно ранее классифицировалось нами в разделе 3.2. На прикладном уровне также функционирует множество пакетов обслуживающих программ. Например, программы передачи файлов

или воспроизведения медиаданных стали уже настолько общим явлением, что обычно рассматриваются как обслуживающее программное обеспечение.

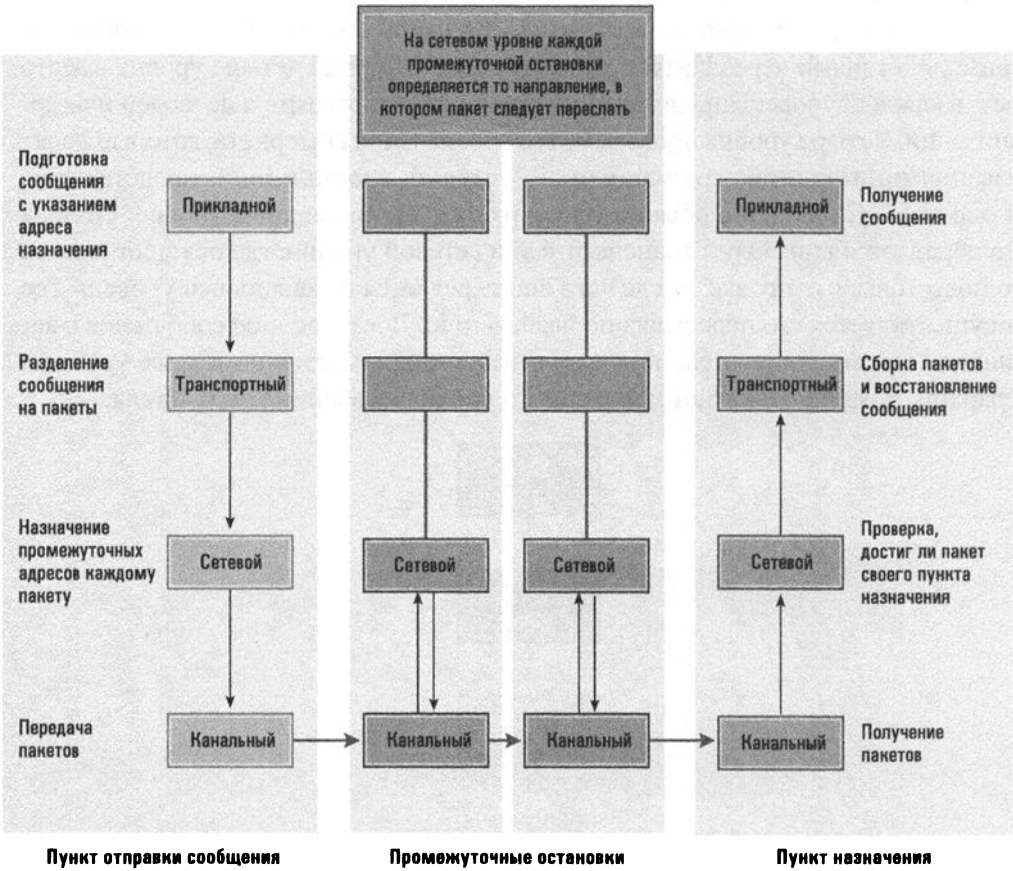


Рис. 4.14. Прохождение сообщения через Интернет

Прикладной уровень использует транспортный уровень для отправки и получения сообщений через Интернет в точности так, как мы используем почтовую службу для отправки и получения посылок. И так же, как в обязанности отправителя посылки входит написание адреса получателя в форме, соответствующей требованиям почтовой службы, обязанностью прикладного уровня является предоставление адреса доставки в формате, отвечающем требованиям инфраструктуры Интернета. Именно для решения этой задачи прикладной уровень нуждается в услугах серверов имен Интернета, к которым он обращается для перевода мнемонических адресов, понятных людям, в IP-адреса, совместимые с программным обеспечением Интернета.

Важная задача транспортного уровня — получить сообщение от программы прикладного уровня и гарантировать, что это сообщение будет правильно

отформатировано для передачи через Интернет. Для достижения последней цели длинные сообщения на транспортном уровне разбиваются на небольшие сегменты, которые пересылаются через Интернет по отдельности, независимо один от другого. Такое разбиение необходимо по той причине, что одно длинное сообщение может препятствовать нормальному прохождению потока других сообщений через маршрутизаторы Интернета, в которых пересекаются пути множества сообщений. Действительно, небольшие сегменты сообщений без затруднений проходят через эти точки практически не мешая друг другу, тогда как длинные сообщения вынуждают все остальные ожидать, пока их прохождение через маршрутизатор не завершится полностью (похоже на то, как автомашины ожидают на железнодорожных переездах прохождения длинных составов).

На транспортном уровне отдельные сегменты сообщений последовательно нумеруются, что позволяет машине-получателю воссоздать первоначальное сообщение в исходном виде после завершения пересылки всех его частей через Интернет. Затем транспортный уровень присоединяет к каждому сегменту адрес назначения и передает сформированные блоки данных, называемые **пакетами**, на сетевой уровень. С этого момента пакеты воспринимаются как отдельные, независимые друг от друга сообщения вплоть до тех пор, пока они не достигнут транспортного уровня в конечном пункте назначения. Вполне возможно, что каждый пакет, входящий в состав одного исходного сообщения, физически пройдет собственный путь через Интернет, отличающийся от всех прочих.



Основные положения для запоминания

- **Интернет является системой с коммутацией пакетов, в которой цифровые данные пересылаются с предварительным их разбиением на битовые блоки, называемые пакетами. Каждый пакет содержит как собственно передаваемые данные, так и дополнительную управляющую информацию, необходимую для правильной доставки данных.**

В обязанности сетевого уровня входит принятие решения о том, в каком направлении следует отправить пакет на каждом этапе того пути, по которому он проходит через Интернет. На практике сочетание программ сетевого уровня и находящегося ниже его канального уровня и образует то программное обеспечение, которое используется на маршрутизаторах Интернета. Сетевой уровень отвечает за поддержание таблицы маршрутизации и ее использование для определения направления, в котором будут пересылаться пакеты. Канальный

уровень на маршрутизаторах отвечает непосредственно за получение и передачу пакетов.

Таким образом, когда сетевой уровень на машине отправителя получает пакет от транспортного уровня, он использует свою таблицу маршрутизации для определения, куда этот пакет следует направить, чтобы он начал свое путешествие по Сети. Определив правильное направление, сетевой уровень передает пакет на канальный уровень, где и происходит его отправка.

Канальный уровень отвечает за пересылку пакетов. Следовательно, в его обязанности входит учет всех деталей установки соединений, присущих той сети, в которой находится данная машина. Например, если это сеть Ethernet, канальный уровень должен использовать протокол CSMA/CD. Если же это сеть Wi-Fi, канальный уровень применит протокол CSMA/CA.

Когда пакет будет передан, он поступит на канальный уровень компьютера на другом конце соединения. Здесь канальный уровень передаст пакет на сетевой уровень, на котором адрес конечного пункта назначения пакета сравнивается с элементами таблицы маршрутизации этой машины для определения направления пересылки пакета на следующем этапе. Как только решение будет принято, сетевой уровень возвращает пакет на канальный уровень для отправки в указанном направлении. Подобным образом каждый пакет совершает "прыжки" от машины к машине на пути к своему пункту назначения.

Обратите внимание, что на каждой промежуточной остановке по всему пути следования пакета только канальный и сетевой уровни принимают участие в его обработке (см. рис. 4.14). Следовательно, на маршрутизаторах достаточно иметь только эти два уровня, что уже отмечалось выше. Более того, для минимизации задержки на каждой такой промежуточной "остановке" на маршрутизаторах функции организации пересылки сетевого уровня тесно интегрированы непосредственно в канальный уровень. В результате время, необходимое современным маршрутизаторам для пересылки пакета, измеряется миллионными долями секунды.

Когда пакет достигает пункта своего назначения, именно на сетевом уровне выясняется, что путешествие этого пакета уже завершилось. В этом случае сетевой уровень вместо организации очередной пересылки передает поступивший пакет на транспортный уровень. Транспортный уровень, получая пакеты от сетевого уровня, извлекает из них сегменты передаваемого сообщения и восстанавливает это сообщение в исходном виде, руководствуясь номерами, присвоенными пакетам на транспортном уровне машины-отправителя. Как только все сообщение собрано, транспортный уровень передает его соответствующему компоненту прикладного уровня, и на этом процесс пересылки сообщения завершается.

Определение, какому именно компоненту (службе) прикладного уровня следует передать поступившее сообщение, является важной задачей транспортного

уровня. Этот процесс контролируется посредством назначения различным компонентам уникальных **номеров портов** (не имеющих отношения к портам ввода-вывода, речь о которых шла в главе 2). Для передачи сообщения определенному программному компоненту на машине получателя необходимо, чтобы соответствующий ему номер порта был прикреплен к адресу сообщения еще до его отправки. Тогда, как только сообщение поступит на транспортный уровень точки назначения, оно просто будет передано тому программному компоненту прикладного уровня, к которому относится указанный порт.

Пользователям Интернета редко приходится иметь дело с номерами портов, поскольку обычные приложения используют общепринятые номера портов. Например, когда веб-браузеру требуется запросить отправку в его адрес документа с URL-адресом `http://www.zoo.org/animals/frog.html`, он полагает, что должен установить соединение с HTTP-сервером на машине с адресом `www.zoo.org` с использованием порта номер 80. Аналогично, отправляя почту, SMTP-клиент полагает, что он должен установить соединение с почтовым SMTP-сервером, пользуясь портом с номером 25.



Основные положения для запоминания

- **Маршрутизация в Интернете является отказоустойчивой благодаря избыточности.**
- **Избыточность маршрутизации (т.е. наличие более одного маршрута доставки данных) между двумя точками в Сети повышает общую надежность Интернета и позволяет масштабировать его с целью охвата новых устройств и новых пользователей.**

Итак, коммуникация через Интернет предусматривает взаимодействие четырех уровней программного обеспечения. Прикладной уровень имеет дело с сообщениями с точки зрения приложений. Транспортный уровень преобразует эти сообщения в совокупность сегментов, совместимых с требованиями инфраструктуры Интернета, и обеспечивает восстановление исходного сообщения из поступивших сегментов перед тем, как передать его соответствующему приложению на стороне получателя. Сетевой уровень обеспечивает требуемое направление пересылки сегментов в Интернете. На канальном уровне осуществляется процесс собственно передачи отдельных сегментов от машины к машине. Принимая во внимание всю совокупность упомянутых выше действий, нельзя без восхищения принимать тот факт, что среднее время ответа в Интернете измеряется миллисекундами, и поэтому многие транзакции здесь осуществляются просто мгновенно.

Семейство протоколов TCP/IP

Спрос на открытые сети вызвал потребность в разработке открытых стандартов, следуя которым, производители могли бы выпускать оборудование и программное обеспечение, корректно взаимодействующее с продуктами других фирм-производителей. Одним из таких стандартов является модель **OSI** (*Open System Interconnection* — взаимодействие открытых систем), разработанная Международной организацией по стандартизации (ISO). Этот стандарт предусматривает иерархию из семи уровней в отличие от описанной выше четырехуровневой системы. На модель OSI часто ссылаются, так как ее поддерживает авторитет международной организации, но ее внедрение вместо четырехуровневой модели, рассматривавшейся выше, идет медленными темпами, главным образом из-за того, что она появилась уже после того, как четырехуровневая иерархия фактически стала стандартом для Интернета.

Семейство протоколов TCP/IP представляет собой набор протоколов, определяющих четырехуровневую иерархию, используемую в Интернете. В действительности "TCP/IP" — это название только двух протоколов: **TCP** (*Transmission Control Protocol* — протокол управления передачей) и **IP** (*Internet Protocol* — протокол Интернета); так что по отношению ко всему этому довольно обширному семейству такое название несколько неточно. Говоря точнее, протокол TCP определяет одну из версий транспортного уровня. Мы употребили слово *версия*, так как в семействе протоколов TCP/IP для представления транспортного уровня существует более одного способа; один из которых определяется протоколом **UDP** (*User Datagram Protocol* — протокол датаграмм пользователя). Эта неоднозначность напоминает ситуацию, в которой при отправке посылки вы имеете возможность выбора среди различных компаний по доставке грузов, каждая из которых предлагает одинаковый набор основных услуг, но имеет собственные уникальные характеристики. Таким образом, в зависимости от специфики требуемого обслуживания программное обеспечение прикладного уровня может выбирать, с помощью какой версии транспортного уровня, протокола TCP или UDP, будут отправлены данные (рис. 4.15).



Основные положения для запоминания

- **Стандарты пакетирования и маршрутизации включают семейство протоколов управления передачей и протокол Интернета (TCP/IP).**

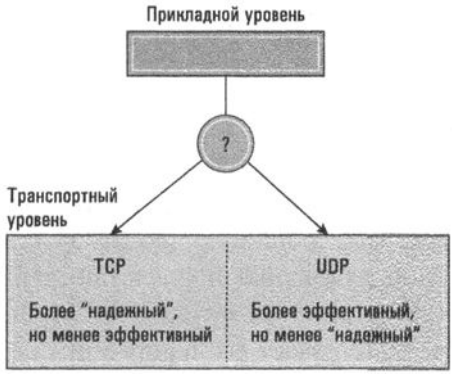


Рис. 4.15. Выбор между протоколами TCP и UDP

Между протоколами TCP и UDP существует несколько различий. Первое состоит в том, что, перед тем как передать сообщение, поступившее от прикладного уровня, транспортный уровень, базирующийся на протоколе TCP, отсылает собственное сообщение транспортному уровню машины-получателя, уведомляющее о том, что предназначенные ему данные готовы к отправке, и извещающее, какое именно программное обеспечение прикладного уровня должно их принять. Затем транспортный уровень машины-отправителя ожидает уведомления о получении адресатом этого предварительного сообщения и только после его прихода приступает к передаче сообщения, поступившего от прикладного уровня. В соответствии с этим говорят, что транспортный уровень с протоколом TCP сначала устанавливает соединение, а затем отсылает данные. Транспортный уровень, основанный на протоколе UDP, не устанавливает никаких соединений, прежде чем отослать данные. Он просто отсылает данные по указанному адресу и забывает о них. Для него не имеет значения, работает ли вообще в данный момент машина, указанная как получатель. Поэтому протокол UDP называют протоколом без установки соединения.

Второе существенное различие между протоколами TCP и UDP заключается в том, что транспортные уровни отправителя и получателя, использующие протокол TCP, совместно работают над обеспечением целостности передаваемых сообщений. Для этой цели используется механизм уведомлений и повторных передач сегментов, выполняемых до тех пор, пока не появится уверенность в том, что все сегменты сообщения были успешно переданы. Поэтому протокол TCP называют *надежным*, в то время как о протоколе UDP, который не предоставляет услуг по повторной передаче сегментов, говорят как о ненадежном.

Еще одно различие между протоколами TCP и UDP состоит в том, что протокол TCP обеспечивает как **управление потоком**, означающее, что транспортный уровень протокола TCP на машине-отправителе может снижать скорость отправки сегментов, чтобы предохранить от перегрузки их получателя в пункте

назначения, так и **контроль перегрузок**, означающий, что транспортный уровень протокола TCP на машине-отправителе может настраивать скорость передачи данных для исключения перегрузки в сети между ним и пунктом назначения.

Все это не означает, что протокол UDP вообще плох. В действительности транспортный уровень, основанный на UDP, работает существенно быстрее, чем транспортный уровень, работающий по протоколу TCP, и если прикладной уровень готов к устранению возможных последствий использования протокола UDP, то последний может оказаться лучшим выбором. Например, высокая эффективность протокола UDP делает его наилучшим выбором при осуществлении DNS-поиска, а также во многих случаях передачи потоковых данных. С другой стороны, поскольку электронная почта менее чувствительна к затратам времени, почтовые серверы для передачи почты, безусловно, должны использовать протокол TCP.

Протокол IP — это стандарт Интернета для решения задач, относящихся к сетевому уровню. Мы уже упоминали о том, что эти задачи включают **пересылку**, которая предполагает передачу пакетов через Интернет, и **маршрутизацию**, предполагающую обновление таблицы маршрутизации уровня для отражения изменяющихся условий. Например, некий маршрутизатор может выйти из строя, а это означает, что поступающие пакеты уже нельзя пересылать в этом направлении и сетевой трафик следует перенаправить в обход этой преграды. Большая часть стандарта IP, связанного с маршрутизацией, касается протоколов, используемых для взаимодействия между соседними сетевыми уровнями с целью обмена информацией о маршрутизации.

Интересная функция, связанная с пересылкой пакетов, состоит в том, что каждый раз, когда использующий протокол IP сетевой уровень на машине-отправителе готовит пакет к передаче на канальный уровень, он прикрепляет к пакету некоторое числовое значение, называемое **счетчиком переходов** по сети или временем жизни пакета. Это значение определяет, сколько раз пакет может пересылаться между машинами в попытках проложить свой путь через Интернет. Каждый раз, когда использующий протокол IP сетевой уровень пересылает некоторый пакет, он уменьшает значение его счетчика переходов на единицу. С помощью этого механизма сетевой уровень защищает Интернет от бесконечной циркуляции пакетов в системе. Хотя размеры Интернета продолжают расти с каждым днем, начальное значение счетчика переходов по сети, равное 64, остается более чем достаточным для того, чтобы позволить пакету найти свой путь в лабиринте маршрутизаторов существующих локальных и глобальных сетей.

Общий эффект от использования всех этих функций подготовки к передаче, маршрутизации и пересылке пакетов состоит в том, что Интернет устойчив в отношении многих типов отказов и способен перейти к использованию других, в обычной ситуации избыточных путей доставки данных, если где-то возникает перегрузка или имеет место отказ.

Многие годы для реализации сетевого уровня во всем Интернете использовалась версия протокола IP, известная как IPv4 (IP версии четыре). Тем не менее довольно скоро Интернет перерос возможности 32-битовой системы интернет-адресации, диктуемой протоколом IPv4. Для решения этой проблемы, а также для реализации других улучшений, таких как групповая передача данных, была разработана новая версия протокола IP, получившая название "IPv6", в которой используется система интернет-адресации с длиной адреса в 128 бит. Процесс перехода от протокола IPv4 к протоколу IPv6 в данный момент уже запущен — об этом упоминалось ранее, при обсуждении принципов интернет-адресации в разделе 4.2. Ожидается, что полный отказ от использования в Интернете 32-разрядных адресов может быть достигнут к 2025 году.



Основные положения для запоминания

- **Количество устройств, способных использовать IP-адреса, возрастало с такой скоростью, что для обеспечения возможности маршрутизации среди такого большого количества устройств был разработан новый протокол (IPv6).**

4.4. Вопросы и упражнения

1. Какие уровни иерархии программного обеспечения Интернета не требуются маршрутизатору?
2. В чем состоят различия между транспортным уровнем, основанным на протоколе TCP, и транспортным уровнем, основанным на UDP?
3. Каким образом транспортный уровень определяет, какому именно компоненту прикладного уровня следует передать поступившее сообщение?
4. Что удерживает компьютер в Интернете от записи копий всех проходящих через него сообщений?

4.5. Простой пример модели "клиент/сервер"

В этом разделе вы познакомитесь с двумя очень простыми сценариями на языке Python, которые совместно представляют минимально возможный вариант сетевой пары из клиента и сервера.

Сценарий "В бесконечность и далее"

Чтобы этот пример был действительно простым, мы ознакомимся с двумя сценариями на языке Python, которые, возможно, не слишком полезны сами по себе. При выполнении сценарий-клиент устанавливает сетевое соединение с заранее определенной точкой и отправляет сообщение "В бесконечность..." любому, кто будет готов получить его на той стороне. Затем он ожидает ответа и печатает любое поступившее в ответ сообщение. Полный текст этого клиентского сценария на языке Python приведен ниже.

```
import socket

# Создание сокета TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Подключение к серверу, прослушивающему порт localhost:1313
server_address = ('localhost', 1313)
print('Подключение к %s:%s' % server_address)
sock.connect(server_address)
# Отправка сообщения
message = b'В бесконечность...'
print('Клиент отправил "%s"' % message.decode())
sock.sendall(message)
# Получение ответа от сервера
reply = sock.recv(11)
print('Клиент получил "%s"' % reply.decode())

# Закрытие соединения с сервером
sock.close
```

Оператор `import` уведомляет интерпретатор языка Python о том, что сценарий содержит ссылки на функции, определенные в библиотеке с названием "`socket`". В контексте сетевого программного обеспечения термин **сокет** (*socket*) представляет собой абстракцию для процессов на прикладном уровне сетевого стека для соединения с другими процессами в сети через расположенный ниже транспортный уровень.

В следующей строке оператор присвоения создает новый сокет с использованием упомянутой выше библиотеки и присваивает ему имя "`sock`". Параметры, передаваемые функции `socket()` из библиотеки `socket`, указывают, что сокет `sock` должен использовать сетевой протокол IP версии 4 из адресного семейства

Интернета ("AF_INET"), а также должен использовать транспортный протокол TCP ("SOCK_STREAM"), а не какой-либо иной из доступных, например UDP.

Следующие три оператора определяют точку назначения для соединения через сокет, выводят пользователю уведомительное сообщение об этой точке, а затем предпринимают попытку установить соединение с серверным процессом в этой предварительно определенной точке. Чтобы наш пример был действительно простым, в нем используется сервер с именем "localhost", под которым любой узел в Интернете понимает самого себя. В результате этот простой клиент на самом деле вовсе не предполагает отправки своего сообщения в сеть. Наш пример построен таким образом, что и клиентский процесс, и серверный процесс функционируют на одной и той же машине, а именно — на машине данного локального узла. Причина, по которой был выбран именно такой подход, состоит лишь в том, что у многих читателей этой книги, вероятно, просто не будет доступа одновременно к двум независимым узлам, подключенным к Интернету, чтобы они могли собственноручно выполнить на них этот пример. Более того, весьма вероятно, что на подавляющем большинстве узлов непременно будут приняты определенные меры защиты (см. обсуждение концепции брандмауэра в следующем разделе этой главы), которые не позволят рядовым пользователям установить соединение между их собственными клиентами и серверами без предоставления им специального разрешения со стороны системного администратора. Расширение возможностей этого простого примера программы в сторону поиска IP-адреса удаленного узла (и получения от системного администратора разрешения на доступ к нему) мы оставим в качестве упражнения для тех читателей, которые этого пожелают.

В номере порта 1313, который был выбран нами для демонстрационных целей, на самом деле нет ничего особенного, не считая того, что это номер издания данной книги, в котором впервые появился этот демонстрационный пример. Любое числовое значение между 1023 и 65 535 может быть использовано на большинстве узлов, определяя порт, который еще не используется другим процессом.

Оператор `sock.connect(server_address)` требует от программ лежащего ниже транспортного уровня и операционной системы попытаться установить соединение с процессом, прослушивающим на узле `localhost` порт с номером 1313. Поскольку в сокете мы используем надежное соединение по протоколу TCP, этот оператор приведет к программной ошибке, если при выполнении клиентского сценария не будет обнаружен серверный процесс, прослушивающий указанный порт для установки требуемого соединения.

Следующие три оператора предназначены для подготовки клиентской программой простого сообщения "`В бесконечность...`", вывода на печать уведомляющего сообщения для пользователя и выполнения попытки отправить это

сообщение в полном объеме процессу на другой стороне сокетного соединения. Стоящий перед строкой сообщения символ "b" и дополнительная функция decode() в операторе печати подчеркивают одну из тонких особенностей, имеющих место при отправке сообщений по сети. Функция sendall() из библиотеки socket языка Python предполагает использование строки символов, каждый из которых представлен одним байтом, тогда как функция print() ожидает получения символьной строки, представленной в иной системе кодирования, подобной UTF-8 и использующей несколько байтов для представления одного символа. Язык Python предоставляет средства для преобразования представления строк символов из одной формы в другую, но мы не будем останавливаться на подобных деталях в этом простом примере сетевых соединений.

Следующие два оператора обеспечивают ожидание ответа от сервера, который при получении помещается в переменную "reply", и распечатку поступившего ответного сообщения в кодах UTF-8 с помощью функции print(). Процедура ожидания реализована в виде функции recv() непосредственно в библиотеке socket, и если наш простой клиент не сможет установить сообщение с процессом, который ответит на отправленное ему клиентом сообщение, то данный сценарий будет ожидать этого ответа неограниченно долго.

Последний оператор в демонстрационном примере клиентского сценария закрывает установленное сокетом соединение с сервером, возвращая операционной системе ресурсы транспортного уровня, выделенные этому сокету, для их последующего повторного использования.



Основные положения для запоминания

- **Данные могут сохраняться в различных форматах в зависимости от их характера (например, размера или предполагаемого способа использования).**

Ниже приведен простой серверный сценарий, предназначенный для обслуживания только что рассмотренного примера сценария клиента.

```
import socket

# Создание сокета TCP/IP
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Связывание сокета с портом localhost:1313
server_address = ('localhost', 1313)
sock.bind(server_address)
# Прослушивание входящих клиентских соединений
sock.listen(1)
```

```
print('Ожидание установки клиентом соединения')
client, client_port = sock.accept()
print('Соединение с клиентом %s:%s' % client_port)

# Получение от клиента запроса
request = client.recv(18)
print('Клиент отправил "%s"' % request.decode())
reply = b'И далее!'
print('Сервер отвечает "%s"' % reply.decode())
client.sendall(reply)

# Закрытие соединения с клиентом
client.close()
```

Детали построения этого примера сценария сетевого сервера очень схожи с деталями соответствующего клиентского сценария. Первое различие можно обнаружить там, клиентский сценарий вызывает функцию сокета `connect()`, — сервер вместо нее вызывает функции сокета `bind()` и `listen()`. Первый вызов требует от лежащих ниже уровней операционной системы использования порта 1313 и, если он доступен, связывает его с данным серверным процессом. Если запрашиваемый порт будет недоступен, выполнение оператора `bind()` вызовет сообщение об ошибке, и работа серверного сценария будет прекращена. Функция `listen()` помечает этот сокет как готовый к приему входящих соединений от клиентов.

Функция `accept()` ожидает, пока клиентский процесс обратится к серверному процессу через сетевое соединение с этим узлом и этим портом. Она возвращает два значения: адрес узла и номер порта клиента, установившего соединение.

Задача функции `recv()`, принимающей сообщение от клиента, в нашем случае существенно упрощена, поскольку мы знаем, что отправляемое им сообщение "`b'В бесконечность...'`" имеет длину 18 байт. Получив его, сервер выводит соответствующую информацию на печать, готовит ответное сообщение "`И далее!`", распечатывает его, а затем отправляет это сообщение клиенту и закрывает сокетное соединение. Если у вас серверный сценарий по какой-то причине прекратит работу *до* выполнения оператора `close()`, номер порта, который им прослушивался, может остаться недоступным еще несколько минут, пока операционная система восстановит ресурс.

Чтобы на практике опробовать работу этого простого демонстрационного примера, сначала в окне интерпретатора языка Python следует запустить серверный сценарий. Он выведет в окно сообщение

```
Ожидание установки клиентом соединения
```

и перейдет в состояние ожидания.

Затем в другом окне интерпретатора языка Python запустите на выполнение клиентский сценарий. Он выведет в свое окно такие сообщения:

```
Подключение к localhost:1313
Клиент отправил "В бесконечность..."
Клиент получил "И далее!"
```

Тем временем серверный сценарий также выведет несколько сообщений:

```
Соединение с клиентом 127.0.0.1:53233
Клиент отправил "В бесконечность..."
Сервер отвечает "И далее!"
```

На этом выполнение обоих сценариев должно благополучно завершиться. Чтобы эта демонстрация прошла успешно, вам потребуются определенное время и удача. Если на любой из сторон возникнут затруднения с доступом к порту 1313, вы можете попробовать воспользоваться любым другим портом с номером меньше 65 536 при условии, что как на клиентской, так и на серверной сторонах будет получено разрешение использовать порт с одним и тем же номером.

Этот простой пример наглядно демонстрирует, как клиентский и серверный процессы могут обмениваться информацией через транспортный уровень системы сокетов, которая достаточно часто используется в клиентском и серверном программном обеспечении в Интернете. Поскольку основное внимание в этом примере уделяется вовсе не расширенным возможностям синтаксиса и библиотек языка Python, мы опустили многие важные детали реальных систем "клиент/сервер". Реальные серверные процессы обычно работают в цикле, вновь и вновь устанавливая соединения с клиентами. Также наш серверный процесс не анализирует поступивший от клиента запрос и всегда отвечает на него сообщением "И далее!", что бы клиент ни отправил в его адрес. Поскольку конструкции языка Python для выполнения некоторых похожих действий потребуются вам и в последующих главах этой книги, мы еще вернемся в этому примеру приложения "клиент/сервер" и попробуем его расширить. Несколько

4.5. Вопросы и упражнения

1. Что такое сетевой сокет?
2. Какая информация необходима для создания нового сокета?
3. В чем состоит основное различие между процедурами создания клиентского соединения и серверного соединения?
4. Какие варианты представления символьных данных можно использовать для их отправки через сетевое соединение?

дополнительных строк кода позволят создать простой серверный сценарий, который будет способен обрабатывать различные типы запросов и отправлять клиентам действительно полезную информацию.

4.6. Кибербезопасность

Когда машина подключена к сети, она становится доступной для несанкционированного доступа и вандализма, расценивающихся как формы киберпреступности. В этом разделе будут рассмотрены темы, связанные с этими проблемами. Под термином **кибербезопасность** (*cybersecurity*) понимается обеспечение безопасности компьютеров, сетей и данных с целью их защиты от подобных атак.

Типы атак

Существует множество способов, посредством которых компьютерная система и ее содержимое могут быть атакованы через сетевые соединения. Чаще всего атаки фокусируются на программном обеспечении компьютерной системы, однако и оборудование системы, и даже человек, который ею пользуется, также могут оказаться важными слабыми точками в этой системе. Для обеспечения безопасности таких систем может потребоваться вмешательство в каждую из этих областей.



Основные положения для запоминания

- **Реализация кибербезопасности включает программный, аппаратный и человеческий компоненты.**

Большинство атак предполагает использование вредоносного программного обеспечения (чаще их называют просто **вредоносными программами**). Подобное программное обеспечение может быть передано по сети, а затем выполнено на самом компьютере, либо оно может атаковать компьютер на расстоянии. Примерами вредоносных программ, передаваемых, а затем выполняемых на атакуемых компьютерах, могут быть вирусы, черви, троянские программы или шпионские программы, названия которых отражают основные характеристики этого вредоносного программного кода.

Вирус — это вредоносное программное обеспечение, которое заражает компьютер посредством вставки своего кода в программы, которые уже присутствуют в компьютере. Затем при выполнении программы-носителя выполняется и

программный код вируса. В процессе своего выполнения многие вирусы ограничиваются лишь тем, что копируют самих себя в другие программы на этом компьютере. Однако некоторые вирусы могут выполнять разрушительные действия, такие как частичное повреждение операционной системы, стирание больших блоков массовой памяти или иное повреждение данных и других программ.

Червь обычно представляет собой автономную программу, которая распространяет саму себя по сети, самовольно устанавливаясь на компьютере и рассылая свои копии на другие машины. Как и в случае с вирусами, такие программы могут быть созданы как для того, чтобы просто рассылать свои копии, так и для нанесения определенного ущерба. Типичным следствием появления в системе червя является взрывное распространение его копий по сети, снижающее производительность легитимных приложений и способное в конечном итоге вызвать перегрузку всей компьютерной сети или даже корпоративного интранета.

Троянской программой (иначе — троян, троянский конь) является программа, которая попадает в компьютерную систему под видом некоторого привлекательного ПО, такого как безобидная игра или пакет полезных утилит, охотно и по своей воле переносимого жертвой на собственный компьютер. Однако, появившись на компьютере, троянская программа начинает проявлять дополнительную активность, которая может давать и вредоносный эффект. Иногда такая дополнительная активность начинает проявляться немедленно, тогда как в других случаях троянская программа остается в этом смысле бездействующей до наступления какого-либо определенного события, например наступления заранее выбранной даты. Троянские программы часто попадают на компьютеры в виде вложений в сообщения электронной почты с каким-либо заманчивым предложением. При открытии такого вложения (т.е. когда получатель решает просмотреть это вложение), вредоносные действия троянской программы активизируются. Следовательно, с вашей стороны будет благоразумно *никогда не открывать* вложения в сообщениях электронной почты, поступивших от неизвестных вам отправителей.

Еще одной формой вредоносного программного обеспечения являются **шпионские программы**, представляющие собой специальный программный код, предназначенный для сбора информации о различных действиях, выполняемых на компьютере, где он обосновался. Собранную информацию шпионские программы обычно отправляют злоумышленникам, организовавшим нападение. Некоторые компании используют шпионское программное обеспечение как инструмент создания профилей пользовательских предпочтений, например для организации эффективной контекстной рекламы, и в этом случае такой подход также имеет сомнительную этическую окраску. В других случаях шпионские программы используются для откровенно вредоносных целей, таких как запись последовательностей символов, вводимых с клавиатуры, с целью выявления среди них паролей или номеров кредитных карт.

Противоположный подход к тайному сбору информации состоит в использовании шпионских программ, применяющих метод **фишинга** (*phishing* — "выуживание"), когда требуемую информацию получают явно, просто попросив ее указать. Сам термин "фишинг" указывает на то, что в этом случае предполагается забросить в Сеть множество "удочек" с приманкой в надежде на то, что кто-то на нее "клюнет". Фишинг часто осуществляется посредством массовой рассылки сообщений электронной почты, и в этой форме он мало отличается от давней практики мошеннических звонков по телефону. Исполнитель рассылает сообщения электронной почты, выдавая себя за финансовое учреждение, правительственное бюро или даже за правоохранительный орган. В этих письмах потенциальную жертву просят сообщить конфиденциальную информацию, которая якобы требуется для неких законных целей. Однако собранная информация используется мошенником исключительно для вредоносных действий.

Помимо того что компьютер может пострадать от различных вирусов, троянских или шпионских программ, действующих изнутри, он может быть атакован в сети программным обеспечением, выполняемым на других компьютерах в системе. В качестве примера можно указать атаку типа **отказ в обслуживании** (*Denial of Service* — **DoS**), представляющую собой искусственный процесс масштабной перегрузки компьютера поступающими сообщениями. Такие атаки неоднократно организовывались в Интернете против крупных коммерческих веб-серверов с целью нарушить бизнес соответствующей компании и в некоторых случаях действительно останавливали коммерческую деятельность атакуемой организации.

Группа быстрого реагирования на инциденты в компьютерной системе

В ноябре 1988 года запущенная в Интернете программа-червь вызвала серьезные сбои в работе сети. Как следствие агентство DARPA (*Defense Advanced Research Projects Agency* — Управление перспективных исследовательских проектов) Министерства обороны США создало группу быстрого реагирования на инциденты в этой глобальной компьютерной системе, получившую название CERT (*Computer Emergency Response Team* — Компьютерная группа реагирования на чрезвычайные ситуации), размещенную в координационном центре CERT Университета Карнеги–Меллона. Группа CERT выступает в роли наблюдателя за безопасностью в Интернете. В ее обязанности входит изучение проблем безопасности, распространение предупреждений о возможной угрозе, проведение кампаний по повышению доверия пользователей к безопасности работы в Интернете и т.д. Координационный центр CERT поддерживает веб-сайт (`http://www.cert.org`), на котором размещает сообщения о своей деятельности.

Организация атаки типа "отказ в обслуживании" предполагает генерацию и отправку на целевой сервер громадного количества сообщений за как можно более короткий период времени с целью вызвать его критическую перегрузку. Для решения этой задачи атакующий обычно распространяет специальное программное обеспечение среди множества компьютеров, владельцы которых ни о чем не подозревают. Всю совокупность этих зараженных вредоносным ПО компьютеров часто называют **ботнет** (*botnet* — сеть роботов). В назначенное время им выдается сигнал, по которому все эти компьютеры начинают генерировать целый шквал бесполезных сообщений, в котором и тонет атакуемый сервер. В случае, когда в ботнет входит множество различных узлов, атаку называют **распределенной атакой отказа в обслуживании** (*Distributed Denial of Service* — **DDoS**). Как следствие в атаках DDoS очень велика вероятность использования в качестве исполнителей компьютеров множества пользователей, не имеющих никакого представления о своем участии в незаконных разрушительных действиях. Именно по этой причине всем пользователям компьютеров рекомендуется устанавливать все обновления в системах защиты, предоставляемых разработчиками операционных систем их компьютеров, а также обеспечивать организацию необходимой защиты на любом устройстве, подключенном к Интернету. Было установлено, что если компьютер подключен к Интернету, то каждые 20 минут по крайней мере один злоумышленник попытается использовать его в своих целях. И если взглянуть с другой стороны, то напрашивается вывод, что каждый плохо защищенный узел представляет собой серьезную угрозу целостности Интернета.



Основные положения для запоминания

- **Распределенные атаки отказа в обслуживании (DDoS) нарушают работу целевых серверов, заваливая их запросами, одновременно поступающими от множества систем.**

Другой проблемой, связанной с обилием нежелательных сообщений, является процветающая практика рассылки множества бесполезных сообщений электронной почты, получивших название **спам**. Однако в этом случае, в отличие от DoS-атак, объем спама редко оказывается достаточным, чтобы вызвать перегрузку в работе компьютерной системы. Спам может вызвать перегрузку не компьютера, а человека, который его получает. Эта проблема усугубляется тем фактом, что, как мы уже видели, спам является весьма популярным инструментом для фишинга и внедрения троянских программ, а также может использоваться для распространения вирусов и другого вредоносного программного обеспечения.



Основные положения для запоминания

- **Фишинг, вирусы и другие атаки всегда включают человеческий и программный компоненты.**

Защита систем и их лечение

Старая пословица "Береженного Бог бережет", безусловно, верна и в контексте борьбы с вандализмом в компьютерных сетях и Интернете. Метод, используемый в этом случае в качестве первичной профилактики, состоит в фильтрации сетевого трафика, проходящего через некоторую точку в сети, — обычно с использованием специального программного обеспечения, называемого **брандмауэром**. Например, брандмауэр может быть установлен в шлюзе внутренней сети организации с целью фильтрации сообщений, входящих и выходящих из этой сети. Такие брандмауэры могут быть настроены для блокирования исходящих сообщений с определенными адресами назначения или для блокирования входящих сообщений из источников, о которых заранее известно, что они могут стать источником проблем. Последняя функция представляет собой инструмент для прекращения DoS-атак, поскольку обеспечивает возможность блокировки трафика с атакующих компьютеров. Еще один распространенный прием использования брандмауэров на шлюзе состоит в блокировании всех *входящих* сообщений, адрес отправителя которых находится в той сети, которая связывается с внешним миром через этот шлюз, — поскольку появление такого сообщения указывает на то, что кто-то извне внутренней сети организации предпринимает попытку выдать себя за ее члена. Подобный маскарад, когда кто-то пытается выдать себя за другого, называется **спуфингом**.



Основные положения для запоминания

- **Антивирусное программное обеспечение и брандмауэры позволяют организовать защиту личных данных от неавторизованного доступа.**

Брандмауэры также используют для защиты отдельных компьютеров, а не только сетей в целом или определенных доменов. Например, если некоторый компьютер не используется как веб-сервер, сервер имен или почтовый сервер, то на нем можно установить брандмауэр, который будет блокировать весь входящий трафик, адресованный соответствующим приложениям. И действительно, один из способов, посредством которого злоумышленник может получить

доступ к компьютеру, заключается в подключении к нему через "дыру", созданную несуществующим серверным приложением. В частности, эффективный метод получения информации, собранной шпионской программой, состоит в установке на зараженном компьютере секретного сервера, через который вредоносные клиентские приложения смогут считывать накопленные шпионские данные. Тем не менее правильно настроенный брандмауэр способен успешно блокировать любые сообщения, поступающие от вредоносных клиентских программ.

Существуют варианты брандмауэров, разработанных для конкретных целей; примером могут служить **фильтры спама**, представляющие собой брандмауэры, предназначенные исключительно для блокирования нежелательных почтовых сообщений. Многие фильтры спама используют достаточно сложные приемы, чтобы отличать нежелательную почту от желательных почтовых сообщений. Так, некоторые фильтры обучаются, чтобы правильно выполнять этот анализ посредством тренировочного процесса, при котором пользователь вручную отмечает письма со спамом до тех пор, пока программный фильтр не соберет достаточно примеров, чтобы принимать решения самостоятельно. Эти фильтры спама являются примерами того, как результаты, полученные в различных областях науки (теория вероятностей, искусственный интеллект и т.д.), можно успешно объединять для достижения успехов в других областях.

Другим превентивным инструментом, который также осуществляет фильтрацию, является **прокси-сервер**. Прокси-сервер представляет собой программный блок, который выступает посредником между клиентом и сервером с целью защиты клиента от враждебных действий со стороны сервера. При отсутствии прокси-сервера клиент взаимодействует непосредственно с сервером, а это означает, что у сервера есть возможность получить определенные сведения об этом клиенте. С течением времени, когда много клиентов в корпоративной сети организации войдут в контакт с этим удаленным сервером, последний сможет собрать множество информации о структуре внутренней сети организации, причем такой, которая позднее сможет быть использована для проведения злонамеренных действий. С учетом такой возможности организация может установить прокси-сервер для определенного типа сервисов — FTP, HTTP и т.д. В результате каждый раз, когда клиент в сети организации предпримет попытку установить контакт с сервером такого типа, на самом деле клиент установит контакт лишь с соответствующим прокси-сервером организации. Этот прокси-сервер, в свою очередь, сыграет роль клиента и установит контакт с требуемым сервером вне сети организации. В результате прокси-сервер будет выполнять функции посредника между истинным клиентом и истинным сервером, пересылая сообщения в прямом и обратном направлениях. Первым преимуществом такой схемы является то, что у истинного сервера не будет никакой

возможности узнать, что прокси-сервер вовсе не является тем истинным клиентом, который с ним взаимодействует; фактически у него даже не возникнет подозрения о том, что может существовать и иной, истинный клиент. А это означает, что внешний сервер будет полностью лишен возможности изучать какие-либо характеристики внутренней сети организации. Второе преимущество заключается в том, что благодаря своей позиции прокси-сервер имеет возможность фильтровать все сообщения, отправляемые сервером клиенту. Например, прокси-сервер, контролирующий службу FTP, может проверять все пересылаемые клиентам файлы на наличие известных вирусов и блокировать передачу любых зараженных файлов.

Совершенно иным инструментом предотвращения различных проблем в сетевом окружении является аудит программного обеспечения, подобный тому, речь о котором шла при обсуждении безопасности операционных систем в разделе 3.5. Пользуясь программным обеспечением для ведения сетевого аудита, системный администратор сможет выявить неожиданное возрастание количества передаваемых сообщений в различных точках в пределах контролируемой им области, отслеживать активность сетевых брандмауэров и анализировать комбинации запросов, отправленных отдельными компьютерами сети с целью обнаружения нарушений их нормальной работы. Несомненно, программное обеспечение сетевого аудита можно считать основным инструментом сетевого администратора, позволяющим ему выявлять угрозы безопасности сети еще то того, как события выйдут из-под контроля.

Еще одним средством защиты от вторжений из сетевых соединений является программное обеспечение, получившее название **антивирусные программы**. Оно предназначено для выявления и удаления программ, зараженных известными вирусами, а также иных вредоносных элементов. (В действительности антивирусное программное обеспечение представляет собой обширный класс программных продуктов, каждый из которых предназначен для выявления и удаления объектов с определенным типом заражения. Например, в то время как одни продукты специализируются именно на выявлении вирусов, другие предназначены для обнаружения шпионского программного обеспечения.) Для пользователей подобных пакетов очень важно понимать, что, как и в случае биологических систем, в мире постоянно появляются все новые и новые типы компьютерных "инфекций", которые требуют использования новых типов "вакцин". Поэтому антивирусное программное обеспечение необходимо постоянно актуализировать, своевременно загружая обновления, предоставляемые разработчиками этих продуктов. И даже в этом случае у вас не будет абсолютной гарантии в отношении безопасности вашего компьютера. В конце концов, новый вирус должен сначала заразить несколько компьютеров, прежде чем его появление можно будет обнаружить, а затем и разработать соответствующую

"вакцину". Поэтому грамотный пользователь компьютера никогда не станет открывать вложения в сообщениях электронной почты, поступивших от неизвестных ему отправителей, никогда не станет загружать программное обеспечение, не убедившись предварительно в его надежности, никогда не станет отвечать на всплывающие рекламные объявления и не оставит компьютер подключенным к Интернету, когда это соединение ему больше не требуется.

Криптография

В некоторых случаях сетевой вандализм ставит своей целью разрушение системы (как в случае атак отказа от обслуживания), но в иных случаях его целью может быть получение доступа к определенной информации. Традиционным методом защиты информации является управление доступом к ней с помощью паролей. К сожалению, пароли можно узнать многими способами, а само их использование вообще не имеет смысла при передаче данных через Интернет, где этот процесс осуществляется при участии неизвестных посредников. Изучением вопроса безопасной отправки и получения сообщений в присутствии угроз занимается **криптография**. Инструменты криптографии, такие как **шифрование**, могут использоваться для преобразования сообщения в его закодированную версию, такую, что даже если она попадет во враждебные руки, зашифрованная в сообщении информация останется недоступной. В настоящее время многие традиционные интернет-приложения были доработаны с целью включения в них криптографических функций и получения тем самым "защищенных версий" этих приложений.



Основные положения для запоминания

- **Криптография является важнейшим элементом многих моделей обеспечения кибербезопасности.**

Ярким примером такого подхода является защищенная версия протокола HTTP, получившая название **HTTPS**. Сейчас эта версия используется в большинстве финансовых учреждений с целью предоставления своим клиентам защищенного доступа к их счетам через Интернет. Костяк стандарта HTTPS образует его **протокол защиты транспортного уровня** (*Transport Layer Security* — **TLS**), являющийся прямым наследником протокола, получившего название **уровень защищенных сокетов** (*Secure Sockets Layer* — **SSL**), исходно разработанного компанией Netscape для обеспечения защищенных коммуникационных соединений между веб-клиентами и веб-серверами. В большинстве

браузеров использование протокола TLS индицируется посредством отображения пиктограммы замочка в своей адресной строке, причем в некоторых случаях на использование или не использование этого протокола TLS указывает наличие или отсутствие этой пиктограммы, тогда как в других случаях замочек отображается всегда, но он может быть либо закрыт (использование защиты), либо открыт (защита отсутствует).



Основные положения для запоминания

- **Стандарты на обмен информацией и взаимодействие между браузерами и веб-серверами включают протокол HTTP и протоколы уровня защищенных сокетов/защиты транспортного уровня (SSL/TLS).**

Столетиями для защиты сообщений шифровальщики использовали схемы шифрования, зависящие от секретности алгоритма кодирования или декодирования и/или секретности паролей (иначе называемых ключами). Поскольку математические принципы, на которых строились многие криптографические схемы, со временем были изучены достаточно глубоко, компьютеры оказались грозными инструментами взлома для многих традиционных систем шифрования, — иногда просто потому, что позволяли опробовать *все* возможные ключи несравненно быстрее, чем это мог бы сделать человек. Системы **шифрования с симметричным ключом**, в которых один и тот же ключ используется как при шифровании, так и при расшифровке сообщений, в целом обеспечивают повышенную безопасность, поскольку компьютеры могут работать с большими и, казалось бы, случайными ключами. В идеальных условиях каждый бит, на который увеличивается длина ключа в схеме шифрования, удваивает количество возможных сочетаний, которые должен проверить злоумышленник при попытке его взлома. В протоколах HTTPS и SSL для защиты транзакций в Интернете используются методы шифрования с симметричным ключом. В основание этих систем положены математические операции, разработанные таким образом, что знания о том, как шифруются сообщения (алгоритма шифрования), будет недостаточно для расшифровки сообщений. Эта характеристика кажется несколько противоречивой: в конце концов, интуиция подсказывает нам, что если человек знает, как шифруются сообщения, то у него всегда будет возможность просто выполнить те же самые математические операции, которые использовались в процессе шифрования, но в обратном порядке, и тем самым расшифровать сообщение. Однако системы шифрования с симметричным ключом опровергают эти интуитивные предположения. Тем не менее проблема все же остается и состоит она в том, что при реализации безопасной связи в системах

с симметричным ключом сам секретный ключ должен быть секретным же образом *согласован* между двумя сторонами, отправителем и получателем, но передавать по сети его следует так, что значение ключа должно остаться секретом для любого злоумышленника.

Основные положения для запоминания

- **Криптографические системы опираются на математическое обоснование эффективности их алгоритмов.**
- **Симметричное шифрование представляет собой метод, предполагающий использование одного и того же ключа для процедур шифрования и дешифрования.**

Одним из более интересных подходов в области криптографии является **шифрование с открытым ключом**, при котором алгоритмы как шифрования, так и дешифрования являются общеизвестными, как и один из двух используемых ключей, но при этом сообщение остается надежно защищенным.

При шифровании с открытым ключом используются два значения, называемые ключами. Один из них, называемый **открытым ключом**, используется для кодирования сообщений, тогда как второй, называемый **закрытым ключом**, необходим для его расшифровки. При использовании такой схемы открытый ключ рассылается всем, у кого может возникнуть необходимость отправить сообщение на определенный адрес. Закрытый ключ хранится в секрете у того, кому предстоит получать эти сообщения. Таким образом, когда отправителю потребуется отправить сообщение, он зашифрует его с использованием открытого ключа *получателя*, а затем отправит ему в полной уверенности, что содержимое сообщения останется конфиденциальным даже в том случае, если оно будет перехвачено на промежуточном этапе лицом, которому также известен этот открытый ключ. Дело в том, что знание лишь открытого ключа не позволяет декодировать сообщения; это может сделать только тот, кто владеет *закрытым* ключом. Таким образом, если Петр создаст систему шифрования с открытым ключом и сообщит открытый ключ Василию и Татьяне, то и Василий, и Татьяна смогут отправлять зашифрованные сообщения Петру, но каждый из них не сможет читать адресованные Петру сообщения, отправленные ему другим, хотя знает тот ключ, который был использован при их шифровании (рис. 4.16). И действительно, хотя интуитивно может показаться, что схема с секретными правилами шифрования должна обеспечивать лучшую защиту сообщению, направленному в адрес Петра, большинство наиболее надежных схем шифрования, используемых в Интернете, являются открытыми стандартами и любой может знать алгоритм, применяемый в них для шифрования, но

при этом сообщения остаются надежно защищенными до тех пор, пока закрытый ключ остается известным только его владельцу.

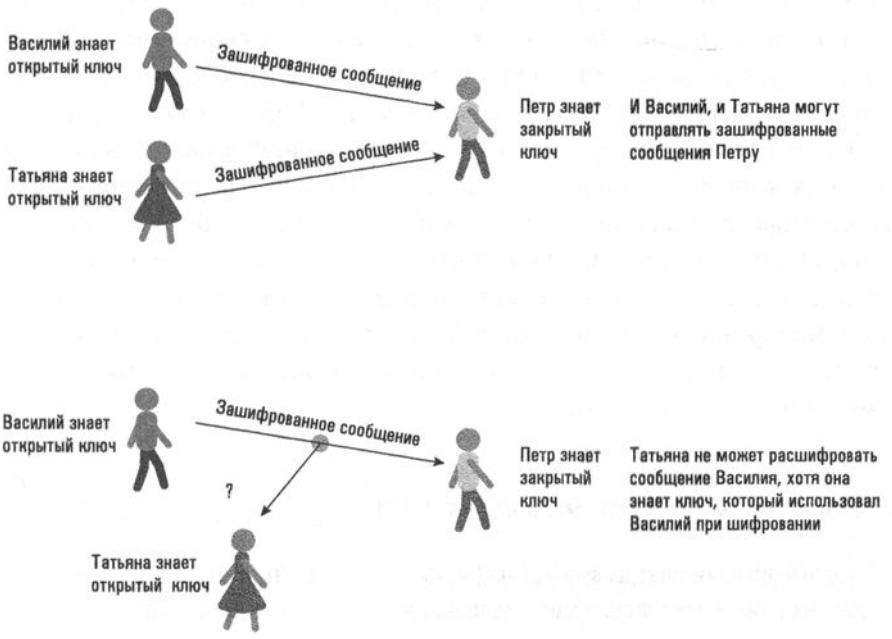


Рис. 4.16. Шифрование с открытым ключом

Основные положения для запоминания

- **Открытые стандарты обеспечивают безопасность шифрования.**
- **Шифрование с открытым ключом, не являющееся симметричным, представляет собой метод шифрования, получивший самое широкое распространение благодаря высокой степени защищенности, которую он обеспечивает.**

Безусловно, существуют и некоторые тонкие проблемы, связанные с использованием систем с открытым ключом. Одна из них состоит в получении гарантии, что открытый ключ, который был использован при шифровании, действительно является правильным открытым ключом для получающей стороны. Например, ведя переписку со своим банком, вы должны быть уверены, что открытый ключ, который вы используете при шифровании, является именно тем, который был получен именно от банка, а не от некоего самозванца. Если самозванец представит себя вашим банком (это яркий пример спуфинга) и передаст вам свой открытый ключ, все сообщения, которые вы зашифровали и отправили "банку", будут понятны только мошеннику, но не вашему банку.

Следовательно, проблема ассоциирования открытых ключей с соответствующей получающей стороной является очень важной.

Один из подходов к решению этой проблемы состоит в создании доверительных интернет-сайтов, называемых **центрами сертификации** (*certificate authorities* — удостоверяющий центр), задача которых состоит в поддержании точных списков сторон и их открытых ключей. Эти центры, функционирующие как серверы, предоставляют достоверную информацию об открытых ключах своих клиентов в виде пакетов, получивших название **сертификаты**. Каждый сертификат представляет собой пакет, содержащий название получающей стороны и ее открытый ключ. В настоящее время в Интернете доступно множество коммерческих центров сертификации, хотя для организаций также является общепринятой поддержка собственных удостоверяющих центров с целью обеспечения более жесткого контроля в отношении уровня безопасности их внешних коммуникаций.



Основные положения для запоминания

- **Удостоверяющие центры выдают цифровые сертификаты, которые подтверждают владение ключами шифрования, используемыми в защищенных коммуникациях, основывая свою работу на модели доверия.**

Другая проблема, присущая системам с открытым ключом, состоит в их большой вычислительной сложности, в результате чего их процедуры шифрования требуют существенно больше времени в сравнении со схожими системами с симметричным ключом. Решение, часто применяемое в Интернете, заключается в использовании комбинации этих криптографических методов, сочетающей уникальные преимущества каждого из них. Программный код большинства браузеров обычно включает список доверенных центров сертификации, поэтому, получив с любого из них сертификат, браузер клиента может с уверенностью полагаться на то, что будет располагать истинным открытым ключом требуемого удаленного сервера. Далее браузер использует относительно медленную процедуру шифрования с открытым ключом, чтобы договориться с удаленным сервером об использовании в предстоящей сессии совершенно нового секретного ключа. Обменявшись новым секретным ключом в режиме, обеспечивающим невозможность его раскрытия любым из подслушивающих устройств, браузер и сервер затем обращаются к быстрой технологии криптографии с симметричным ключом для шифрования сообщений, отправляемых с использованием протокола HTTPS, скажем, в процессе выполнения покупки через Интернет с использованием кредитной карты.

И наконец, следует сказать несколько слов о роли систем шифрования с открытым ключом в решении проблем **аутентификации**, т.е. получения гарантий, что автор сообщения на самом деле является тем, за кого себя выдает. Критическим моментом здесь является то, что в некоторых системах шифрования с открытым ключом роли открытого и закрытого ключей можно поменять на обратные. То есть отправляемое сообщение шифруется с использованием закрытого ключа, и, поскольку только одна из сторон имеет к нему доступ, любой текст, зашифрованный подобным образом, может поступить только от этой стороны. Следуя приведенной логике, держатель закрытого ключа может выбрать последовательность битов, обычно называемую **цифровой подписью**, которую в зашифрованном виде сможет предоставить только он сам. Далее, прикрепляя эту подпись к сообщению, отправитель получает возможность маркировать все свои отправляемые сообщения как аутентичные. В самом простом случае цифровая подпись может быть такой же простой, как зашифрованная версия самого сообщения. Тогда все, что отправитель должен сделать, — это зашифровать предназначенное для передачи сообщение с использованием его закрытого ключа (того самого, который обычно используется для дешифрования). Когда сообщение будет доставлено, получатель воспользуется открытым ключом отправителя, чтобы расшифровать цифровую подпись. Полученное в результате сообщение будет гарантированно аутентичным, поскольку только владелец закрытого ключа сможет создать его зашифрованную версию, поступившую в адрес получателя.

К теме шифрования с открытым ключом мы еще вернемся позднее, в конце главы 12, где она будет рассматриваться как пример сложного компьютерного алгоритма.

Правовое регулирование безопасности сетей

Другим способом повышения защищенности систем из компьютерных сетей является использование правовых инструментов. Однако в этом случае необходимо учитывать два обстоятельства. Первое состоит в том, что объявление действия незаконным не предотвращает его совершения. Все, что обеспечивается в этом случае, — это предоставление юридических обоснований для подобной его классификации. Второе обстоятельство заключается в интернациональной природе сетевых систем, означающей, что получение необходимых юридических обоснований часто оказывается чрезвычайно сложным. То, что является незаконным в одной стране, может быть совершенно законным в другой. В конечном счете повышение безопасности сетей правовыми методами является сугубо международной задачей, а следовательно, должно контролироваться международными правовыми учреждениями, например потенциальным претендентом на эту роль может быть Международный суд в Гааге.

Даже приняв во внимание сказанное выше, следует признать, что, пусть даже не в полной мере, юридические силы, тем не менее, имеют огромное влияние, а значит, нам будет полезно познакомиться с некоторыми юридическими мерами, предпринятыми для разрешения конфликтных ситуаций в области компьютерных сетей. С этой целью мы воспользуемся выдержками из федерального законодательства Соединенных Штатов. Схожие решения могут быть найдены и у соответствующих органов других государств, например стран Европейского Союза.

Мы начнем с проблемы распространения вредоносного программного обеспечения. В США эта проблема решается Законом о борьбе с компьютерным мошенничеством и злоупотреблениями, который в исходном варианте был принят в 1984 году, а затем несколько раз дорабатывался. На основании именно этого закона в судебном порядке рассматривалось большинство случаев, связанных с внедрением в сети червей и вирусов. Говоря коротко, закон требует предоставления доказательств в отношении того, что ответчик сознательно осуществил распространение программы или данных, способных преднамеренно причинять ущерб. Закон о борьбе с компьютерным мошенничеством и злоупотреблениями также распространяется на случаи похищения информации. В частности, в нем объявляется незаконным получение чего-нибудь ценного посредством неавторизованного доступа к компьютеру. В судах отмечалась склонность воспринимать формулировку "что-нибудь ценное" в расширенной интерпретации, вследствие чего данный закон применяли не только к похищению информации. Например, суд мог постановить, что даже простое *использование* компьютера можно рассматривать как "что-нибудь ценное".

Право на неприкосновенность частной жизни — другой и, вероятно, наиболее спорный и противоречивый аспект правового урегулирования в области компьютерных сетей. Тщательному рассмотрению были подвергнуты такие вопросы, как право работодателя отслеживать обмен информацией, осуществляемый его работниками, или то, в каких пределах провайдеру услуг Интернета разрешается доступ к информации, которой обмениваются его клиенты. В Соединенных Штатах Америки многие из подобных вопросов регулируются законом о тайне обмена электронной информацией (Electronic Communication Privacy Act — ECPA), принятым в 1986 году. В качестве его основы использовалось законодательство о перехвате информации. Хотя этот документ достаточно объемный, его содержание можно передать несколькими короткими выдержками. В частности, в нем утверждается следующее.

> За исключением особо оговоренных случаев, любое лицо, которое намеренно перехватывает, пытается перехватить или склоняет другое лицо к перехвату или попытке перехвата любой информации по проводам, в устной форме или в виде электронного сообщения... будет наказано в соответствии с подразделом 4 или подвергнуто судебному преследованию в соответствии с подразделом 5.

Вот еще одна выдержка.

> ...любому лицу или организации, предоставляющей услуги электронного сообщения для общественного пользования, запрещается намеренное раскрытие содержания любого обмена информацией... любому иному лицу или организации, кроме получателя, которому эта информация предназначена, или его официального представителя.



Основные положения для запоминания

- **Проблемы конфиденциальности и безопасности возникают как при разработке, так и при использовании вычислительных систем и их программного обеспечения.**

В целом, закон ECPA подтверждает право индивидуума на тайну общения — незаконными считаются действия провайдера сетевых услуг, связанные с распространением сведений о содержании информации, которой обмениваются его клиенты, а также службы, не уполномоченные на то специально, не имеют права прослушивать информацию, которой обмениваются третьи лица. Однако закон ECPA также оставляет место для дебатов. Например, вопрос в отношении права работодателя отслеживать обмен информацией, осуществляемый его работниками, сводится к вопросу о полномочиях, которые суды, как правило, склонны предоставлять работодателю, когда обмен информацией осуществляется с использованием принадлежащего ему оборудования.

Более того, этот закон идет еще дальше и предоставляет некоторым государственным службам право проводить мониторинг электронных сообщений, но с некоторыми ограничениями. Эти положения также стали источником бурных дискуссий. Например, в 2000 году ФБР обнародовало сведения о существовании принадлежащей ему системы под названием "Carnivore", которая выдавала отчеты об обмене информацией *всеми* абонентами провайдера услуг Интернета, а не только теми, которые были указаны судом. Более того, в 2001 году в ответ на атаку террористов на Всемирный торговый центр, в конгрессе прошел спорный законопроект USA PATRIOT (*Uniting and Strengthening America by Providing Appropriate Tools Required to Intercept and Obstruct Terrorism* — Объединение и усиление Америки предоставлением соответствующих инструментов, необходимых для перехвата и препятствия терроризму), в котором были изменены ограничения, установленные в отношении прав, ограничивающих деятельность государственных служб. В 2013 году было установлено, что эти законы были истолкованы так, что разрешали неизбирательный сбор огромного количества информации об использовании телефонов и Интернета простыми американцами.

В дополнение к правовым и этическим дискуссиями, разгоревшимся вокруг этих нововведений, предоставление прав на мониторинг вызвало появление некоторых технических проблем, которые ближе к основной тематике этой книги. Одна из них состоит в том, что должны быть обеспечены соответствующие технические возможности, — иначе говоря, системы связи должны разрабатываться и программироваться так, чтобы наблюдение за ними было возможным. Для обеспечения таких возможностей был принят акт о содействии средств связи выполнению закона (Communications Assistance for Law Enforcement Act — CALEA). Он требует от владельцев средств телекоммуникации модифицировать их оборудование таким образом, чтобы оно допускало перехваты, требуемые по закону. Однако практическая реализация этого закона оказалась делом весьма сложным и дорогостоящим.



Основные положения для запоминания

- **Существующие технологии позволяют проводить сбор, изучение и использование информации "о", "полученной от" и "предназначенной для" отдельных лиц, групп и учреждений.**

Еще более спорная проблема заключается в конфликте между правом комиссии FCC осуществлять контроль за информацией и правом пользователей использовать шифрование. Ведь если контролируемые сообщения хорошо зашифрованы, то простой их перехват в линиях связи практически бесполезен для служителей закона. Поэтому правительства США, Канады и многих европейских стран пошли по пути создания системы регистрации, требующей обязательной регистрации ключей шифрования, но это требование вызвало яростный отпор у корпораций. В конечном счете это можно понять, приняв во внимание размах промышленного шпионажа в нашем мире. Не вызывает сомнения, что требование регистрации ключей шифрования ставит многие законопослушные корпорации, как, впрочем, и отдельных граждан, в затруднительное положение. Насколько надежной будет сама регистрирующая система? Поскольку разработчики программного обеспечения перешли к использованию более мощных средств шифрования, включив их как базовые функции в свои продукты, некоторые правоохранительные органы отреагировали на это, требуя принятия значительных исключений. В 2016 году ФБР предложило компании Apple разработать новое программное обеспечение, которое позволит им взломать айфон одного из нападавших в перестрелке в Сан-Бернадино в предыдущем году. Компания Apple отклонила это предложение, ссылаясь на то, что это создаст опасный прецедент, а также на то, что подобные действия могут

подорвать доверие к системе защиты устройств от Apple во всем мире. Тем не менее ФБР нашло альтернативный способ проникнуть в запертый телефон, прежде чем этот спорный случай был решен в судебном порядке.

В завершение для демонстрации всей обширности области правовых вопросов, имеющих отношение к Интернету, мы обратимся к закону о защите прав потребителей от киберсквоттинга (*Anticybersquatting Consumer Protection Act* — ACPA) от 1999 года, который был разработан для защиты организаций от самозванцев, которые в противном случае могли бы зарегистрировать похожие на названия этих организаций доменные имена для последующей перепродажи (практика, получившая название "киберсквоттинг"). Этот закон запрещает использование доменных имен, которые идентичны или достаточно похожи на чужую торговую марку или "товарный знак общего права". Одним из следствий является тот факт, что хотя закон не запрещает спекуляцию доменными именами (т.е. регистрацию потенциально привлекательного доменного имени с последующей перепродажей прав на него), он ограничивает эту практику лишь общими доменными именами. Иначе говоря, некто, спекулирующий доменными именами, имеет право зарегистрировать общее имя, например `GreatUsedCars.com`, но не имеет права объявить свои права на доменное имя, скажем, `BigAlUsedCars.com`, если название "Big Al" уже используется в автомобильном бизнесе. Подобные различия часто являются предметом дебатов в судебных процессах, основанных на применении закона ACPA.

4.6. Вопросы и упражнения

1. Что такое "фишинг"? Как можно защитить от него компьютеры?
2. В чем состоит различие между брандмауэрами, которые могут быть установлены на шлюзах доменов и которые могут быть установлены на отдельных узлах в пределах домена?
3. С технической точки зрения термин "данные" относится к представлению информации, тогда как термин "информация" ссылается на основное значение этих данных. Что защищает использование пароля — данные или информацию? Что защищает шифрование — информацию или данные?
4. В чем состоит преимущество шифрования с открытым ключом в отношении традиционных методов шифрования?
5. Какие проблемы возникают при попытках обеспечить правовой подход в отношении защиты компьютерных сетей?

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. Что такое протокол? Укажите три протокола из числа тех, с которыми вы познакомились в этой главе, и опишите назначение каждого из них.
2. Опишите модель "клиент/сервер".
3. Дайте описание одноранговой модели.
4. Опишите три типа распределенных компьютерных систем
5. В чем состоит различие между открытой и закрытой сетями?
6. Почему протокол CSMA/CD не применим в беспроводных сетях?
7. Опишите последовательность выполняемых машиной действий, когда ей необходимо передать сообщение в сети, работающей по протоколу CSMA/CD.
8. В чем заключается проблема скрытого терминала? Опишите способ ее решения.
9. Чем сетевой концентратор отличается от повторителя?
10. Что отличает маршрутизатор от таких сетевых устройств, как повторители, мосты или сетевые коммутаторы?
11. В чем состоит различие между сетью и сетью сетей (internet)?
12. Укажите два протокола для управления правом передачи сообщения в сети.
13. Первоначально предполагалось, что использование 32-разрядных интернет-адресов оставляет достаточно места для расширения Интернета, однако это предположение оказалось неверным. В стандарте IPv6 используется 128-разрядная адресация. Будет ли этого достаточно? Аргументируйте свой ответ. (Например, вы можете сравнить количество возможных адресов с численностью населения Земли.)
14. Представьте каждую из приведенных ниже битовых комбинаций в десятичной нотации с точками.

    а. `00000101000101000100011`

    б. `1000000000100000`

    в. `0011000000011000`

15. Какая битовая комбинация представлена каждым из приведенных ниже значений в десятичной нотации с точками?

    а. `0.0` **б.** `26.19.1` **в.** `8.12.20.13`

16. Предположим, что адрес узла в Интернете задан как 134.48.4.123. Каков будет его 32-битовый адрес в шестнадцатеричном представлении?
17. Что такое DNS-поиск?
18. Если мнемонический интернет-адрес компьютера имеет вид `batman.bat-cave.metropolis.gov`, какое предположение можно сделать о структуре домена, в который входит этот компьютер?
19. Укажите и опишите назначение составных частей следующего адреса электронной почты: `kermit@.animals.com`.
20. В контексте протокола VoIP в чем состоит различие между адаптером аналоговой телефонии и встроенным телефоном?
21. В чем заключается назначение почтового сервера?
22. В чем состоит различие между методами многоадресной однонаправленной передачи (N-unicast) и групповой передачей (multicast) данных?
23. Дайте определения следующих понятий.
    а. Сервер имен.
    б. Провайдер доступа к Интернету.
    в. Шлюз.
    г. Конечная система.
24. Дайте определение для каждого из следующих понятий.
    а. Гипертекст.
    б. HTML.
    в. Браузер.
25. Многие "непрофессиональные" пользователи Интернета полагают термины *Интернет* и *World Wide Web* взаимозаменяемыми. Чему соответствует каждый из этих терминов в действительности?
26. Просматривая простую веб-страницу, дайте браузеру указание отобразить исходную версию документа и определите его основную структуру. В частности, укажите заголовок и тело документа, а также перечислите несколько инструкций, которые вы найдете в каждой из этих частей.
27. Назовите пять тегов языка HTML и опишите их назначение.
28. Измените приведенный ниже HTML-документ таким образом, чтобы в новом варианте слово "Корсар" представляло собой ссылку на документ с URL-адресом `http://animals.org/pets/dogs.html`.

```
<html>
<head>
<title>Пример</title>
</head>
```

```
<body>
У меня есть собака
<p>Мою собаку завут Корсар.</p>
</body>
</html>
```

29. Нарисуйте схему, представляющую, как приведенный ниже HTML-документ будет выглядеть на экране монитора компьютера.

```
<html>
<head>
<title>Пример</title>
</head>
<body>
У меня есть собака
<img src = "Corsair.jpg">
</body>
</html>
```

30. Используя неформальный стиль языка XML, представленный выше в этой главе, разработайте язык разметки для представления простых алгебраических выражений в виде текстовых файлов.

31. Используя неформальный стиль языка XML, представленный выше в этой главе, разработайте набор тегов, которые текстовый процессор сможет использовать для разметки исходного текста. Например, как можно указать текстовому процессору, что некоторый фрагмент текста следует выделить полужирным написанием либо что он должен быть подчеркнутым, ну и так далее?

32. Используя неформальный стиль языка XML, представленный выше в этой главе, разработайте набор тегов, который можно будет использовать для разметки обзоров кинофильмов в соответствии с тем, как текстовые элементы должны отображаться на печатной странице. Затем разработайте набор тегов, которые можно будет использовать для разметки обзоров в соответствии со смыслом отдельных элементов текста.

33. Используя неформальный стиль языка XML, представленный выше в этой главе, разработайте набор тегов, который можно будет использовать для разметки статей о спортивных событиях в соответствии с тем, как текстовые элементы должны отображаться на печатной странице. Затем разработайте набор тегов, которые можно будет использовать для разметки статей в соответствии со смыслом отдельных элементов текста.

34. Укажите компоненты следующего URL-адреса и объясните назначение каждого из них.

```
http://lifeforms.com/animals/moviestars/kermit.html
```

35. Укажите компоненты каждого из следующих сокращенных URL-адресов.

а. `http://www.farmtools.org/windmills.html`

б. `http://castles.org/`

в. `www.coolstuff.com`

36. Чем будут различаться действия браузера, если потребовать у него найти документ, находящийся на веб-узле с URL-адресом `http://stargazer.universe.org` по отношению к случаю, когда URL-адрес веб-узла будет указан как `https://stargazer.universe.org`?

37. Приведите два примера действий, выполняемых в среде веба на стороне клиента. Приведите два примера действий, выполняемых в среде веба на стороне сервера.

***38.** Что такое эталонная модель OSI?

***39.** В сети с топологией шины сама шина представляет собой неразделяемый ресурс, за доступ к которому машинам приходится конкурировать каждый раз, когда им требуется отправить сообщение. Как в этом контексте удается избежать ситуаций взаимной блокировки (см. раздел 3.4)?

***40.** Перечислите четыре уровня иерархии программного обеспечения Интернета и опишите задачи, выполняемые каждым уровнем.

***41.** Почему на транспортном уровне большие сообщения разрезаются на меньшие части, упаковываемые в пакеты?

***42.** Когда приложение указывает транспортному уровню, что для передачи сообщения следует использовать протокол TCP, какие дополнительные сообщения необходимо будет отправить транспортному уровню, чтобы выполнить это требование прикладного уровня?

***43.** С какой точки зрения протокол TCP представляется более удачным протоколом транспортного уровня, чем протокол UDP? В чем заключаются преимущества протокола UDP в сравнении с TCP?

***44.** Что означает утверждение о том, что UDP является протоколом, не устанавливающим соединения?

***45.** На какой уровень иерархической модели протокола TCP/IP следует поместить брандмауэр, чтобы он мог фильтровать поступающие сообщения на основании проверки их элементов, перечисленных ниже.

а. Содержимое сообщения.

б. Адрес отправителя.

в. Тип приложения.

46. Предположим, вам требуется установить в сети брандмауэр для отфильтровывания сообщений электронной почты, содержащих определенные слова или выражения. Где следует поместить этот брандмауэр — на

шлюзе вашего домена или на почтовом сервере домена? Поясните свой ответ.

47. Что такое прокси-сервер и в чем его преимущества?
48. Изложите основные принципы шифрования с открытым ключом.
49. Чем опасен для Интернета незащищенный простаивающий ПК?
50. В каком смысле глобальная природа Интернета ограничивает возможность юридического решения возникающих в нем проблем?

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответ на предложенный вопрос. Вы должны разобраться, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.



Основные положения для запоминания

- **Коммерческая и государственная цензура цифровой информации поднимает юридические и этические проблемы.**
- **Электронная почта, SMS-сообщения и сетевой чат открыли новые возможности общения и сотрудничества.**
- **Интернет и веб оказали различное — плодотворное, положительное и отрицательное — влияние на многие области жизни человека.**
- **Видеоконференции и видеочат способствовали распространению новых способов общения и сотрудничества.**
- **Социальные сети продолжают предлагать и развивать все новые и новые способы общения.**
- **Облачные вычисления открыли новые способы общения и сотрудничества.**
- **Интернет и веб предоставляют новые методы и возможности общения и сотрудничества.**
- **Интернет и веб вызвали существенные изменения во многих областях, включая электронную коммерцию, здравоохранение, доступ к информации и развлечениям, интерактивное обучение.**
- **Интернет и развернутые на его основе системы содействуют сотрудничеству.**

1. Возможность подключать компьютеры и смартфоны к Интернету облегчила людям совместную работу по-новому и привела к росту популярности концепции работы на дому. Электронная почта, видеоконференции, текстовые и программные средства совместной работы в облаке позволили отказаться от многих традиционных способов сотрудничества и общения лицом к лицу непосредственно на рабочем месте. В чем состоят преимущества и недостатки этой идеи? Влияет ли она на уровень потребления природных ресурсов? Способствует ли она укреплению семей? Станет ли меньше "служебных интриг"? Будут ли работающие на дому обладать теми же возможностями продвижения по службе, что и те сотрудники, которые ежедневно ходят на работу? Ослабнут ли общественные связи? Какой эффект — положительный или отрицательный — будет иметь уменьшение контактов работников и работодателей?
2. Онлайн-заказ товаров неуклонно вытесняет традиционный "ручной" способ совершения покупок в магазинах из "кирпича и бетона". Какое влияние это окажет на привычки в обществе, связанные с процессом покупок? Что произойдет с супермаркетами? Что произойдет с маленькими магазинами, например книжными или магазинами одежды, по которым многие любят побродить, не делая покупок? Хорошо или плохо покупать товары по минимально возможным ценам? Существуют ли какие-то моральные обязательства платить больше за предмет с целью поддержать местный бизнес? Этично ли осматривать товары в местном магазине, выбирая понравившиеся, а потом заказывать их по более низкой цене через Интернет? Каковы отдаленные последствия такого поведения?
3. В какой степени правительство может контролировать доступ граждан к Интернету (или к другой международной сети)? Что по этому поводу можно сказать, исходя из соображений национальной безопасности? Какие вопросы, связанные с безопасностью, могут при этом возникнуть?
4. Сайты социальных сетей разрешают пользователям Интернета помещать на них свои сообщения (часто анонимно), а также читать сообщения, помещенные другими. Следует ли администратора такой службы считать ответственным за ее содержание? Должна ли телефонная компания нести ответственность за содержание телефонных переговоров? Должен ли управляющий овощным магазином нести ответственность за содержание местной доски объявлений, которая висит на стене магазина?
5. Следует ли контролировать использование Интернета? Следует ли это использование регулировать законодательно? Если да, кто должен этим заниматься и в каких пределах?

6. Сколько времени вы тратите на пользование Интернетом? С пользой ли вы проводите это время? Изменила ли возможность свободно пользоваться Интернетом на вашу социальную активность? Как вам легче общаться с людьми: через социальные сети в Интернете или лично?
7. Когда вы приобретаете пакет программного обеспечения для персонального компьютера, разработчик обычно предлагает вам зарегистрироваться, чтобы ставить вас в известность о возможных усовершенствованиях. Этот процесс регистрации все чаще осуществляется через Интернет. Вас обычно просят ввести такие данные, как имя, адрес и, возможно, сведения о том, откуда вы узнали о продукте, после чего программное обеспечение разработчика автоматически передает ему эти данные. Какие этические проблемы возникнут, если разработчик так сконструирует программу регистрации, что во время регистрационного процесса она будет посылать разработчику некоторую дополнительную информацию? Например, эта программа может сканировать содержимое вашей системы и сообщать о других имеющихся пакетах программного обеспечения.
8. Когда вы посещаете веб-сайт, у этого сайта появляется возможность записать на ваш компьютер данные, которые называют "куки" (от англ. *cookie*, буквально — "печенье"), свидетельствующие о том, что вы посещали этот сайт. Впоследствии эти записи могут использоваться для опознания посетителей, уже заходивших на данный сайт, а также для хранения сведений об их прежних действиях, чтобы последующие визиты посетителя на этот сайт можно было организовать более эффективно. Записи куки на вашем компьютере также можно использовать для сбора сведений о посещаемых вами сайтах. Следует ли предоставлять веб-сайтам возможность записывать куки на вашем компьютере? Следует ли предоставлять веб-сайтам возможность записывать куки на ваш компьютер без вашего ведома? Каковы возможные преимущества от использования записей куки? Какие проблемы могут возникнуть при использовании записей куки?
9. Если от корпораций потребуют регистрации их ключей шифрования в некотором государственном органе, обеспечит ли это решение сохранение их безопасности на прежнем уровне?
10. В общем случае правила этикета предписывают удерживаться от звонков находящимся на рабочем месте друзьям в личных или обоюдных интересах, подобных приглашению выехать на природу в ближайшие выходные. Аналогичным образом большинство из нас будет испытывать сомнения в отношении звонка клиенту на дом с целью описать ему новый продукт. Точно так мы отправляем по почте приглашения на свадьбу на домашний адрес гостя, тогда как приглашения на бизнес-конференцию принято

отправлять на рабочий адрес. Будет ли правильно отправить личное сообщение электронной почты на почтовый сервер по месту работы вашего знакомого?

11. Предположим, что владелец ПК оставил его включенным и подключенным к Интернету, в результате чего он в конечном счете был использован другой стороной для организации атаки отказа от обслуживания. В какой степени владелец ПК должен нести за это ответственность? Будет ли ваш ответ зависеть от того, был ли владельцем ПК установлен на нем надлежащий брандмауэр?
12. Будет ли этичным для компаний, выпускающих конфеты или игрушки, размещать на своих веб-сайтах такие игры, которые не только развлекают детей, но и продвигают продукцию компании? А что можно сказать в том случае, если эти игры к тому же будут разработаны так, чтобы собирать у детей определенную информацию? Где проходят границы между развлечением, рекламированием и использованием?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Antoniou G., Groth P., van Harmelem F., Hoekstra R. *A Semantic Web Primer*, 3rd ed. — Cambridge, MA: MIT Press, 2012.
2. Bishop M. *Introduction to Computer Security*. — Boston, MA: Addison-Wesley, 2005.
3. Comer D.E. *Computer Networks and Internets*, 6th ed. — Upper Saddle River, NJ: Prentice-Hall, 2014.
4. Comer D.E. *Internetworking with TCP/IP*, Vol. 1, 6th ed. — Upper Saddle River, NJ: Prentice-Hall, 2013.
5. Goldfarb C.F., Prescod P. *The XML Handbook*, 5th ed. — Upper Saddle River, NJ: Prentice-Hall, 2004.
6. Halsal F. *Computer Networking and the Internet*, 5th ed. — Boston, MA: Addison-Wesley, 2005.
7. Harrington J.L. *Network Security: A Practical Approach*. — San Francisco: Morgan Kaufmann, 2005.
8. Kurose J.F., Ross K.W. *Computer Networking: A Top Down Approach Featuring the Internet*, 6th ed. — Boston, MA: Addison-Wesley, 2012.
9. Peterson L.L., Davie B.S. *Computer Networks: A Systems Approach*, 5th ed. — San Francisco: Morgan Kaufmann, 2011.
10. Rosenoer J. *CyberLaw: The Law of the Internet*. — New York: Springer, 1997.
11. Spinello R. A., Tavani H.T. *Readings in CyberEthics*, 2nd ed. — Sudbury, MA: Jones and Bartlett, 2004.

12. Stallings W. *Cryptography and Network Security*, 5th ed. — Upper Saddle River, NJ: Prentice-Hall, 2010.
13. Stevens W.R. *TCP/IP Illustrated*, Vol. 1. — Boston, MA: Addison-Wesley, 1994.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Хантер Д., Рафтер Дж., Фаусетт Дж., Ван дер Влист Э. и др. *XML. Базовый курс*, 4-е изд. — М.: ООО "И.Д. Вильямс", 2009.
2. Одом У. *Компьютерные сети. Первый шаг.* — М.: Издательский дом "Вильямс", 2006.
3. Камер Д.Э. *Компьютерные сети и Internet*, 3-е изд. — М.: Издательский дом "Вильямс", 2002.
4. Шиндер Д.Л. *Основы компьютерных сетей.* — М.: Издательский дом "Вильямс", 2002.

Глава 5

В о введении к этой книге указывалось, что центральным звеном компьютерных наук является изучение алгоритмов. Пришло время сосредоточить свое внимание на этой важнейшей теме. Нашей целью будет достаточно глубокое изучение этой фундаментальной концепции, чтобы вы могли в полной мере понять и оценить науку о вычислениях.

Алгоритмы

5.1. ПОНЯТИЕ АЛГОРИТМА
- Неформальный обзор
- Формальное определение алгоритма
- Абстрактная природа алгоритма

5.2. ПРЕДСТАВЛЕНИЕ АЛГОРИТМА
- Примитивы
- Псевдокод

5.3. СОЗДАНИЕ АЛГОРИТМА
- Искусство решения задач
- С чего начать

5.4. ИТЕРАЦИОННЫЕ СТРУКТУРЫ
- Алгоритм последовательного поиска
- Управление циклами
- Алгоритм сортировки методом вставки

5.5. РЕКУРСИВНЫЕ СТРУКТУРЫ
- Алгоритм двоичного поиска
- Управление рекурсией

5.6. ЭФФЕКТИВНОСТЬ И ПРАВИЛЬНОСТЬ
- Эффективность алгоритма
- Верификация программ

Как было сказано выше, прежде чем компьютер сможет выполнить задачу, ему необходимо предоставить алгоритм ее решения, в точности описывающий, что и как надо делать. Следовательно, изучение алгоритмов является краеугольным камнем компьютерных наук. В этой главе мы обсудим многие фундаментальные понятия этой области знаний, включая разработку и представление алгоритмов, а также важнейшие управляющие концепции — итерацию и рекурсию. В процессе обсуждения мы рассмотрим некоторые хорошо известные алгоритмы поиска и сортировки. А начнем мы с общего ознакомления с концепцией алгоритма.

5.1. Понятие алгоритма

Во введении к этой книге алгоритм был неформально определен как последовательность этапов, описывающая способ решения поставленной задачи. В данном разделе мы рассмотрим это основополагающее понятие более детально.

Неформальный обзор

В предыдущих главах этой книги вы познакомились с множеством различных алгоритмов. Среди них можно упомянуть алгоритмы преобразования одной формы представления чисел в другую, обнаружения и исправления ошибок в данных, сжатия файлов данных и их восстановления в исходное состояние, управления вычислительным процессом в многозадачной среде и т.д. Более того, мы выяснили, что машинный цикл, в соответствии с которым работает центральный процессор любых компьютеров, также, в сущности, — не более чем очень простой алгоритм:

```
Пока не будет выполнена команда останова, продолжать
выполнение следующей последовательности этапов:
  а. Выбрать очередную команду.
  б. Декодировать команду.
  в. Выполнить эту команду.
```

Как было показано в алгоритме, описывающем карточный фокус на рис. 0.1, использование алгоритмов не ограничивается лишь наукой и техникой. В действительности они лежат в основе даже такой простой деятельности, как лущение гороховых стручков:

```
Взять корзину гороховых стручков и пустую миску.
Пока в корзине есть гороховые стручки, выполнять
следующую последовательность этапов:
  а. Взять стручок из корзины.
  б. Раскрыть створки стручка.
```

```
в. Вылущить горошины из стручка в миску.
г. Выбросить пустой стручок.
```

Следует отметить, что многие исследователи полагают, что любая активность в сознании человека, включая воображение, творчество и принятие решений, в действительности является результатом выполнения определенных алгоритмов — гипотеза, которую мы подробнее рассмотрим позднее, при исследовании искусственного интеллекта (глава 11).

Однако, прежде чем идти дальше, давайте рассмотрим формальное определение алгоритма.

Формальное определение алгоритма

Неформальные, нестрого определенные концепции обычно применимы и распространены только в повседневной жизни, но наука может оперировать только точно определенными понятиями и терминами. Поэтому обратимся к формальному определению алгоритма, приведенному на рис. 5.1.

Алгоритм — это упорядоченный набор
из недвусмысленных и выполнимых этапов,
определяющий некоторый конечный процесс

Рис. 5.1. Определение понятия алгоритма

Обратите внимание, что определение требует упорядоченности этапов алгоритма. Это означает, что отдельные этапы алгоритма должны составлять четко определенную структуру в смысле порядка их выполнения. В простейшем случае этапы алгоритма выполняются последовательно, в том порядке, в каком они приведены, т.е. строго **упорядоченно**. Однако это не означает, что этапы всегда должны выполняться строго в линейной последовательности: первый, второй и т.д. Так, некоторые алгоритмы, называемые параллельными, содержат больше одной последовательности этапов, каждая из которых разработана так, что может выполняться отдельным процессором многопроцессорной машины. В таких случаях алгоритм в целом не представляет собой единую последовательность этапов, соответствующую сценарию "первый этап, второй этап..." В действительности он содержит множество последовательностей, которые разветвляются и вновь объединяются по мере того, как разные процессоры выполняют различные части задачи. (К этой концепции мы еще раз вернемся в главе 6.) Другими примерами могут служить алгоритмы, выполняемые такими электронными схемами, как триггеры (см. главу 1), в которых каждый вентиль реализует отдельный этап всего алгоритма. В этом случае этапы упорядочены как причина и следствие, так как действие каждого вентиля распространяется на всю схему.



Основные положения для запоминания

- **Упорядоченность — это выполнение каждого этапа алгоритма в том порядке, в каком он приведен.**

Теперь рассмотрим требование, согласно которому алгоритм должен состоять из *выполнимых* этапов. Чтобы оценить важность этого условия, рассмотрим следующую инструкцию.

```
Составить список всех целых положительных чисел.
```

Не вызывает сомнения то, что выполнить ее невозможно, поскольку множество целых положительных чисел бесконечно. В результате любой набор инструкций, включающий эту, будет невыполнимым, а значит, не будет являться алгоритмом. Для понятия "выполнимый" специалисты в области компьютерных наук используют термин *эффективный*. Таким образом, если говорится, что данный этап эффективен, значит, он осуществим.

Еще одним требованием, налагаемым определением, приведенным на рис. 5.1, является недвусмысленность этапов алгоритма. Это означает, что во время выполнения алгоритма при любом состоянии процесса информации должно быть достаточно, чтобы полностью и однозначно определить действия, которые требуется осуществить на каждом его этапе. Другими словами, выполнение любого этапа алгоритма не потребует каких-либо творческих способностей. Наоборот, единственное требование состоит в способности следовать имеющимся инструкциям. (Из главы 12 вы узнаете, что "алгоритмы", не отвечающие этому ограничению и называемые недетерминированными, являются важной темой научных исследований.)

Определение на рис 5.1 также включает требование, что любой алгоритм должен определять *конечный* процесс, а это означает, что выполнение алгоритма обязательно должно приводить к его завершению. Это требование происходит из теории вычислений, в задачи которой входит получение ответа на вопросы типа "Что является предельным ограничением алгоритмов и машин?" В данном случае теория вычислений пытается разграничить вопросы, ответы на которые могут быть получены алгоритмическим путем, и вопросы, ответы на которые лежат за пределами возможностей алгоритмических систем. В этом контексте грань проводится между процессами, которые приводят к конечному результату, и процессами, которые выполняются бесконечно, не приводя к окончательному значащему результату.

Однако в действительности существуют примеры содержательных приложений, использующих бесконечные процессы, например контроль показателей

жизнедеятельности пациента в больнице или поддержание установленной высоты полета авиалайнера. Можно возразить, что на самом деле в этих приложениях многократно повторяются конечные алгоритмы, каждый из которых доходит до своего завершения, а затем автоматически начинается вновь. Тем не менее трудно возразить утверждению, что такие аргументы являются лишь попытками остаться верными ограничительному формальному определению. Как бы там ни было, результат состоит в том, что термин *алгоритм* часто используется в прикладном или неформальном окружении как ссылка на последовательность этапов, которая необязательно определяет завершающийся процесс. Примером может служить известный нам еще со школьной скамьи "алгоритм" деления в столбик, который не определяет конечный процесс в случае деления 1 на 3. Формально подобные случаи представляют собой лишь неправильное использование этого термина.

Абстрактная природа алгоритма

Вначале подчеркнем различие между алгоритмом и его представлением, что аналогично различию между сюжетом и книгой. Сюжет абстрактен или концептуален по своей природе, а книга является его физическим представлением. Если книгу перевести на другой язык или опубликовать в другом формате, изменится только представление сюжета, а сам по себе сюжет останется прежним.

Точно так алгоритм является абстракцией и отличается от своего конкретного представления. Существует много способов представления одного и того же алгоритма. Например, алгоритм для перевода показаний температуры по шкале Цельсия в показания по шкале Фаренгейта традиционно представляется в виде алгебраической формулы

$$F = (^9/_5)C + 32$$

Однако его можно представить и в виде инструкции:

> Умножить значение температуры в градусах Цельсия на $^9/_5$,
> а затем к полученному произведению прибавить 32.

Более того, эту последовательность действий можно представить даже в виде электронной схемы. Однако в каждом случае лежащий в основе алгоритм остается прежним, различаются только методы его представления.

Различие между алгоритмом и его представлением становится проблемой, когда мы предпринимаем попытку передать алгоритм другим. Типичный пример состоит в степени детализации, с которой алгоритм следует описать. Так, для метеорологов инструкция "Перевести показания в градусах Цельсия в эквивалентные им показания по шкале Фаренгейта" будет вполне

удовлетворительной, но непрофессионалы, нуждающиеся в более детальных инструкциях, сочтут ее неоднозначной. Обратите внимание, что проблема заключается не в том, что неоднозначен лежащий в основе алгоритм, а в том, что, с точки зрения непрофессионалов, он представлен недостаточно подробно. Таким образом, неоднозначность скорее присуща представлению алгоритма, а не самому алгоритму. В следующем разделе мы увидим, как концепцию примитивов можно использовать для исключения подобных проблем неоднозначности в представлении алгоритмов.

И наконец, в контексте обсуждения различий между алгоритмами и их представлениями следует также прояснить различие между двумя другими связанными с ними понятиями: программами и процессами. Программа является представлением алгоритма. (Здесь мы используем термин *алгоритм* в его менее формальном смысле, поскольку многие программы являются представлениями бесконечных "алгоритмов".) В действительности специалисты в области компьютерных наук обычно используют термин *программа* по отношению к формальному представлению алгоритма, разработанного для некоторого компьютерного приложения. В главе 3 мы определили *процесс* как деятельность, связанную с выполнением программы. Однако следует заметить, что выполнить программу означает также выполнить представленный этой программой алгоритм, поэтому процесс можно эквивалентно определить как деятельность по выполнению алгоритма. Из сказанного можно сделать вывод, что процессы, алгоритмы и программы — это различные, хотя и взаимосвязанные понятия. Программа является представлением алгоритма, тогда как процесс представляет собой деятельность по выполнению алгоритма.

5.1. Вопросы и упражнения

1. Охарактеризуйте различия между процессом, алгоритмом и программой.
2. Приведите примеры алгоритмов, с которыми вы знакомы. Действительно ли они являются алгоритмами в строгом смысле этого слова?
3. Укажите элементы неопределенности в том неформальном определении алгоритма, которое было предложено в разделе 0.1 вводной главы.
4. Каким пунктам в определении алгоритма не соответствует приведенная ниже последовательность инструкций?

   Этап 1. Возьмите монету из вашего кармана и положите ее на стол.

   Этап 2. Возвратитесь к этапу 1.

5.2. Представление алгоритма

В этом разделе мы рассмотрим вопросы, относящиеся к представлению алгоритмов. Наша задача — ввести основные понятия примитивов и псевдокода, а также разработать некоторую систему представления для собственного использования.

Примитивы

Для представления алгоритмов необходимо использовать некоторую форму языка. Если с алгоритмом работает человек, то это может быть и традиционный язык (английский, испанский, русский, японский), и, возможно, язык картинок, как на рис. 5.2. (На этом рисунке приведен алгоритм, описывающий, как сложить птичку из квадратного листа бумаги.) Однако зачастую использование этих естественных средств общения ведет к неправильному пониманию, причем причина чаще всего состоит в неоднозначности используемой терминологии. Например, предложение "Посещение внуков — это большая нагрузка на нервную систему" может иметь двоякий смысл: либо приезд внуков вызывает массу хлопот, либо поездка к ним является серьезным испытанием для пожилого человека. Другой источник проблем — это неправильное понимание алгоритма, вызванное недостаточной детализацией его описания. Мало кто из читателей сможет успешно сделать птичку, пользуясь лишь указаниями, приведенными на рис. 5.2, тогда как для тех, кто уже изучал искусство оригами, это, вероятно, будет несложно. Короче говоря, проблемы восприятия возникают в тех случаях, когда выбранный для представления алгоритма язык неточно определен или представленная в описании алгоритма информация недостаточно детальна.

В компьютерных науках эти проблемы решают путем создания четко определенного набора составных блоков, из которых могут конструироваться представления алгоритмов. Такие блоки называются **примитивами**. То, что примитивам даются точные определения, устраняет многие проблемы неоднозначности и одновременно требует одинакового уровня детализации для всех описываемых с их помощью алгоритмов. Набор примитивов вместе с набором правил, устанавливающих, как эти примитивы могут комбинироваться для представления более сложных идей, образуют **язык программирования**.



Основные положения для запоминания

- **Алгоритмы, описание которых дано на языке программирования, могут быть выполнены компьютером.**

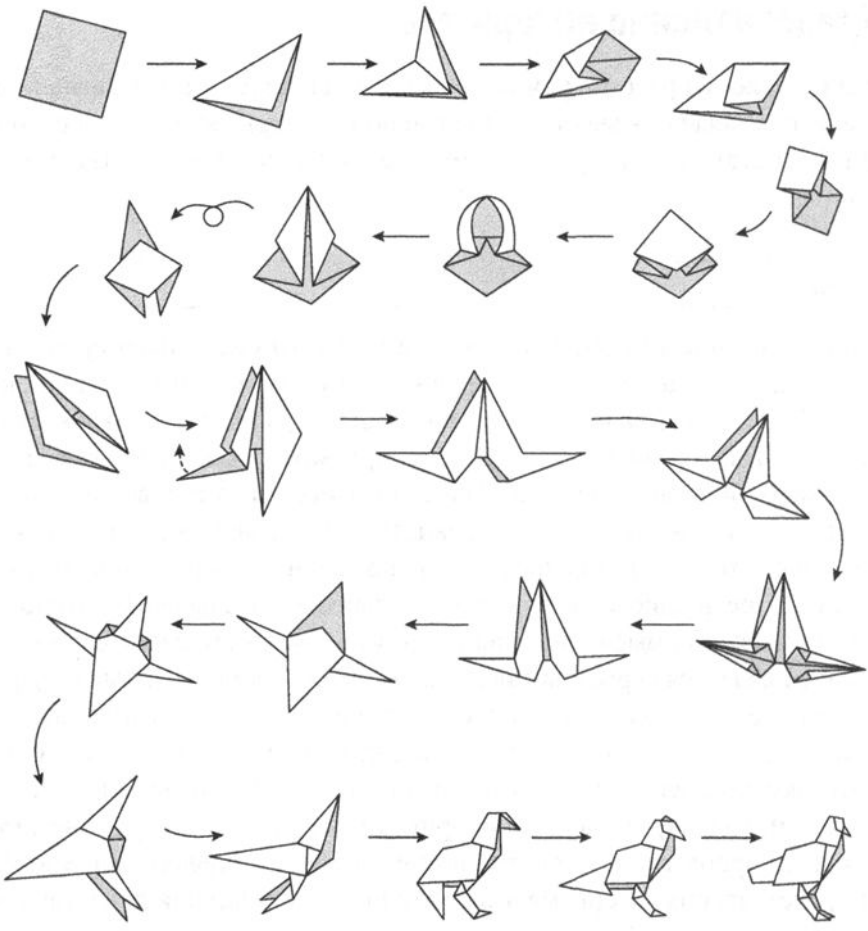


Рис. 5.2. Как сложить птичку из квадратного листа бумаги

Каждый примитив состоит из двух частей: синтаксической и семантической. Синтаксис относится к символьному представлению примитива, а семантика — к значению примитива, т.е. к представляемой им концепции. Например, синтаксис слова *воздух* включает шесть соответствующих символов, тогда как семантически это окружающая наш мир газовая субстанция. В качестве примера на рис. 5.3 представлены некоторые примитивы, используемые в искусстве оригами.

Чтобы получить набор примитивов, пригодных для представления выполняемых машиной алгоритмов, мы можем обратиться к отдельным командам, которые эта машина способна выполнять благодаря ее конструкции. Если алгоритм будет описан с подобным уровнем детализации, то, несомненно, это будет программа, пригодная для выполнения машиной. Однако описание алгоритма на таком уровне детализации весьма утомительно, поэтому обычно используется

набор примитивов более высокого уровня; каждый из них является абстрактным инструментом, сконструированным из примитивов более низкого уровня, представляемых машинным языком. В результате будет получен формальный язык программирования, позволяющий описывать алгоритмы на более высоком концептуальном уровне, чем это возможно в собственно машинном языке. Такие языки программирования мы подробно обсудим в следующей главе.

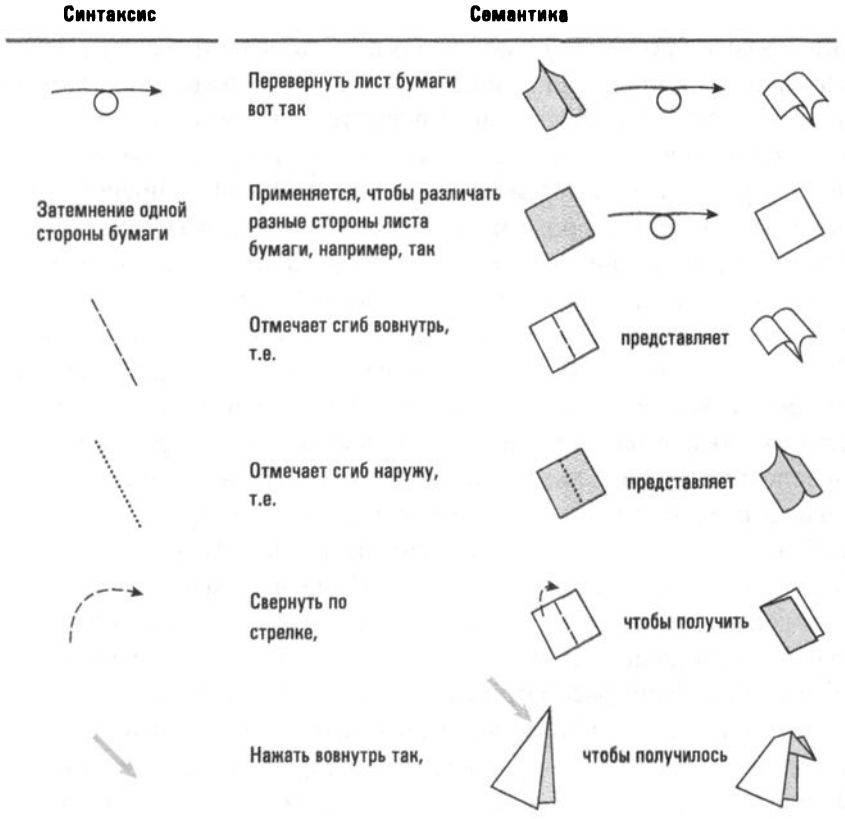


Рис. 5.3. Примитивы, используемые в оригами

Псевдокод

Начиная с этого момента мы отказываемся от использования команд формального языка программирования в пользу менее формальной и более интуитивной системы обозначений, известной как *псевдокод*. В общем случае **псевдокод** — это система обозначений, предназначенная для неформального представления идей в процессе разработки алгоритмов. Преимущество псевдокода в том, что он проще воспринимается человеком, чем любые языки программирования, но при этом он более точен, чем традиционный язык.



Основные положения для запоминания

- **Традиционный язык и псевдокод позволяют описать любой алгоритм так, что человек сможет его понять.**

Один из путей создания псевдокода состоит в простом ослаблении правил формального языка программирования при использовании его синтаксических и семантических структур, смешанных с менее формальными конструкциями. Существует множество вариантов таких псевдокодов, поскольку имеется много различных языков программирования. Двумя довольно популярными вариантами являются упрощенные версии языков Algol и Pascal — преимущественно по той причине, что они широко используются в учебниках и академических публикациях на протяжении многих десятилетий. В последнее время набирают популярности псевдокоды, напоминающие языки Java и C, и вновь по той причине, что большинство программистов могут хотя бы читать программы на этих языках. Однако независимо от того, из какого языка псевдокод заимствует свой синтаксис, он должен обладать одним важнейшим свойством, чтобы успешно выполнять свое предназначение как средства выражения алгоритмов: псевдокод должен обеспечивать непротиворечивую, краткую запись для представления часто встречающихся, повторяющихся семантических структур. Для наших целей в оставшейся части книги при записи псевдокода мы будем использовать синтаксис, близкий к синтаксису языка Python. Одни из семантических структур нашего псевдокода будут заимствованы из языковых конструкций Python, представленных в предыдущих главах, тогда как другие будут взяты из конструкций, которые формально будут рассматриваться в последующих главах.

Одной из таких часто встречающихся семантических структур является присвоение имени вычисленному значению. Например, если была вычислена сумма балансов на текущих и сберегательных счетах, может потребоваться сохранить полученный результат, присвоив ему некоторое имя, по которому на эту сумму можно будет ссылаться впоследствии. В подобных случаях мы будем использовать следующую форму записи:

```
name = выражение
```

Здесь `name` — это имя, по которому мы будем ссылаться на полученный результат в дальнейшем, а запись `выражение` описывает вычисления, результат которых должен быть сохранен под указанным именем. Эта структура псевдокода прямо соответствует эквивалентному **оператору присваивания** языка Python, с которым мы познакомились в главе 1 при рассмотрении вопроса сохранения значения в переменной. Например, оператор

```
RemainingFunds = CheckingBalance + SavingsBalance
```

представляет собой оператор присваивания, который сохраняет сумму значений переменных `CheckingBalance` и `SavingsBalance` в переменной с именем `RemainingFunds`. В результате имя `RemainingFunds` в последующих операторах можно будет использовать для ссылок на эту сумму.

Другой часто используемой семантической структурой является **выбор** одного из двух возможных действий в зависимости от истинности или ложности некоторого условия. Вот примеры ее использования.

Если валовой внутренний продукт растет, покупать обыкновенные акции; в противном случае продавать обыкновенные акции.

Покупать обыкновенные акции, если валовой внутренний продукт растет, и продавать их в противном случае.

Покупать или продавать обыкновенные акции в зависимости от того, растет или уменьшается валовой внутренний продукт.

Представление алгоритма во время его разработки

При разработке алгоритма разработчик должен хранить в памяти множество взаимосвязанных понятий, объем которых в некоторых случаях просто превосходит возможности человеческого мышления. Поэтому разработчик сложных алгоритмов нуждается в средствах записи, которые позволят ему при необходимости вспомнить особенности любой части разрабатываемого им алгоритма.

На протяжении 1950–1960-х годов самым совершенным инструментом проектирования являлись блок-схемы (в которых алгоритмы представлялись в виде геометрических фигур, соединенных стрелками). Однако зачастую они превращались в запутанную паутину пересекающихся стрелок, что затрудняло понимание структуры разрабатываемого алгоритма. Поэтому со временем блок-схемы уступили место другим методам представления. Примером может служить используемый в этой книге псевдокод, в котором алгоритмы описываются посредством точно определенных текстовых структур. Тем не менее блок-схемам по-прежнему отдают предпочтение при проведении презентаций. Например, на рис. 5.8 и 5.9 они используются для демонстрации алгоритмической структуры, присущей распространенным управляющим операторам.

Поиск более удачных средств проектирования продолжается и поныне. В главе 7 мы увидим, что существующие тенденции заключаются в использовании графических методов для задач глобального проектирования больших программных систем, в то время как псевдокод остается популярным при разработке процедурных компонентов меньшего размера, входящих в состав системы.



Основные положения для запоминания

- **В операторах выбора результат вычисления булева выражения анализируется для определения, какую из двух частей алгоритма следует использовать.**

Каждое из этих предложений можно переписать так, чтобы оно соответствовало следующей структуре:

```
if (условие):
  действие
else:
  действие
```

Здесь ключевые слова `if` (*если*) и `else` (*иначе*) используются для того, чтобы объявить различные подструктуры внутри основной структуры, а скобки позволяют обозначить границы этих подструктур. Непосредственно за выражением `условие` и ключевым словом `else` всегда должно следовать двоеточие, а соответствующие им выражения `действие` записываются с отступом. Если выражение `действие` состоит из нескольких этапов, все они должны быть записаны с одним и тем же отступом. Включив такую синтаксическую структуру в наш псевдокод, мы получаем универсальный способ описания указанной общей семантической структуры.

В качестве примера рассмотрим приведенное ниже предложение.

> В зависимости от того, является год високосным или нет, разделить итог на 366 или 365 соответственно.

Хотя приведенный выше вариант больше отвечает литературному стилю, мы будем использовать следующую его версию:

```
if (год високосный):
  ИтогоЗаДень = Итог / 366
else:
  ИтогоЗаДень = Итог / 365
```

Мы также примем сокращенный синтаксис этого конструкта:

```
if (условие):
  действие
```

Он будет использоваться в случаях, когда не предусмотрено действие для варианта `else`. Используем эту схему для следующего предложения:

> В случае уменьшения объема продаж снизить цену на 5%.

Применение сокращенного синтаксиса позволяет сократить исходный вариант следующим образом:

```
if (объем продаж снизился):
  снизить цену на 5%
```

Другая универсальная алгоритмическая структура, **итерация**, заключается в требовании повторять выполнение последовательности инструкций до тех пор, пока некоторое условие остается верным. Ниже приведены неформальные примеры этой структуры.

> До тех пор, пока есть билеты для продажи, продолжать продавать билеты.

и

> Пока есть билеты для продажи, продолжать продажу билетов.



Основные положения для запоминания

- **Итерация — это повторение некоторой части алгоритма до тех пор, пока не будет выполнено заданное условие либо пока она не будет выполнена указанное количество раз.**

Для подобных случаев в нашем псевдокоде будет применяться следующий универсальный шаблон:

```
while (условие):
  действие
```

Если говорить коротко, эта инструкция предписывает проверить условие и, если оно верно, выполнить действие, а затем вновь проверить условие. Если при очередной проверке условие оказывается неверным, следует перейти к инструкции, следующей за данной структурой. Таким образом, оба предшествующих примера при записи на псевдокоде будут выглядеть следующим образом:

```
while (имеются билеты, которые можно продать):
  продавать билеты
```

Итерация, последовательно выполняемые (упорядоченные) этапы и выбор образуют полный набор строительных блоков псевдокода, которого нам будет вполне достаточно для построения любых алгоритмов. Тем не менее использование нескольких дополнительных структур в качестве сокращения позволит увеличить полезность нашего псевдокода в отношении записи алгоритмов в более сжатой форме.



Основные положения для запоминания

- **Упорядоченные (последовательно выполняемые) инструкции, выбор и итерация являются строительными блоками алгоритмов.**
- **Любой алгоритм может быть реализован с использованием только упорядоченных инструкций, выбора и итерации.**

Во многих языках программирования отступы позволяют существенно повысить читабельность программ. В языке Python, а значит, и в нашем псевдокоде, построенном на основе этого языка, отступы являются существенным и обязательным элементом нотации. Например, в операторе

```
if (дождя нет):
  if (Температура == Жарко):
    идти купаться
  else:
    играть в гольф
else:
  смотреть телевизор
```

отступы указывают, что вопрос о том, имеет ли переменная `Температура` значение `Жарко`, даже не будет рассматриваться, если условие `дождя нет` не выполняется. Говорят, что вопрос о температуре является **вложенным** во внешний оператор `if` и анализируется только в том случае, если условие внешнего оператора `if` будет выполнено. Аналогичным образом отступы говорят нам и о том, что элемент `else: играть в гольф` принадлежит внутреннему оператору `if`, а не внешнему, т.е. тому оператору `if`, в который он вложен. Исходя из сказанного, на будущее мы договоримся использовать отступы для отражения структуры текста в нашем псевдокоде.

Мы собираемся использовать наш псевдокод для описания действий, которые могут выступать в роли абстрактных вспомогательных инструментов в других приложениях. В компьютерных науках такие программные элементы имеют несколько различных названий, а именно: "подпрограммы", "модули", "процедуры", "функции" и "методы" (значение каждого из них имеет свой смысловой оттенок). Следуя соглашениям, принятым в языке Python, мы будем использовать в нашем псевдокоде термин **функция** и использовать ключевое слово языка Python `def` для обозначения заголовка, по которому на данный блок псевдокода можно будет ссылаться из других частей псевдокода. Говоря точнее, мы будем начинать каждый блок псевдокода со следующей инструкции:

```
def имя():
```

Здесь имя — это конкретное название, присвоенное данному блоку. Ниже этой вводной инструкции будут следовать инструкции, определяющие выполняемые в этом блоке действия. Например, на рис. 5.4 представлен псевдокод функции с именем Приветствие, в которой трижды печатается сообщение "Привет".

```
def Приветствие():
  Счетчик = 3
  while (Счетчик > 0):
    печатать сообщение "Привет"
    Счетчик = Счетчик - 1
```

Рис. 5.4. Процедура Приветствие, записанная на языке Python

Когда где-либо в псевдокоде потребуется выполнить действия, реализованные в некоторой процедуре, эта процедура будет просто вызываться по имени. Например, пусть две процедуры имеют имена ПредоставлениеЗайма и ОтклонениеЗаявки соответственно. Тогда мы сможем включить их выполнение в структуру if-else, написав следующую инструкцию:

```
if (. . .):
  ПредоставлениеЗайма()
else:
  ОтклонениеЗаявки()
```

В результате процедура ПредоставлениеЗайма будет выполняться, если проверяемое условие истинно, а процедура ОтклонениеЗаявки — если это условие ложно. Обратите внимание, что эту структуру мы можем рассматривать и как композицию алгоритмов, поскольку каждая функция сама по себе является алгоритмом. Таким образом, комбинируя алгоритм под названием ПредоставлениеЗайма с алгоритмом под названием ОтклонениеЗаявки и некоторыми дополнительными алгоритмическими блоками, мы создали новый алгоритм, который можно использовать при рассмотрении заявок на предоставление займа. Этот алгоритм, в свою очередь, также может быть представлен в виде функции, которую затем можно будет многократно использовать как часть более крупных алгоритмов.



Основные положения для запоминания

- **Алгоритмы можно комбинировать для создания новых алгоритмов.**

Поскольку функции могут использоваться в различных ситуациях, их следует разрабатывать так, чтобы они были как можно более общими. Функция сортировки списков имен должна быть способна сортировать любой список, а не какой-то один конкретно определенный. Поэтому она должна быть написана таким образом, чтобы подлежащий сортировке список не определялся в самой процедуре. Вместо этого список в процедуре должен быть представлен некоторым обобщенным именем.

В нашем псевдокоде условимся перечислять эти обобщенные имена (которые называют **параметрами**) в круглых скобках в той же строке, в которой определяется имя данной процедуры. В частности, процедура с именем Сортировка, предназначенная для сортировки произвольных списков имен, будет начинаться следующей инструкцией:

```
def Сортировка (Список):
```

Далее в представлении, где потребуется ссылка на подлежащий сортировке список, будет использовано обобщенное имя Список. В свою очередь, когда потребуется обращение к процедуре Сортировка, мы определим, какой именно список имен должен быть передан в процедуру Сортировка под именем Список. Для этого можно использовать, например, синтаксис

```
Сортировка(список сотрудников организации)
```

или

```
Сортировка(список гостей на свадьбе)
```

исходя из от наших потребностей на текущий момент.

Не забывайте, что назначение нашего псевдокода состоит в предоставлении средств, позволяющих записывать схемы алгоритмов лишь в общих чертах, а не в написании законченных формальных программ. Поэтому мы не связаны запретом использования в алгоритмах неформальных фраз, запрашивающих такие действия, детали которых пока не определены достаточно строго. (Как именно будут разрешены эти детали, не столь уж важно для описания алгоритма, поскольку это касается только свойств языка, на котором будет написана формальная программа.)

Именование элементов в программах

В традиционных языках предметы часто имеют имена, состоящие из нескольких слов, например "стоимость создания единицы" или "ориентировочное время прибытия". Опыт показывает, что использование подобных многосложных имен в представлении алгоритма может заметно усложнять его описание. Гораздо

удобнее идентифицировать каждый элемент одним непрерывным блоком текста. За прошедшие годы были выработаны различные методы сжатия нескольких слов в единую лексическую единицу с целью получения описательных имен различных элементов в программах. Один из них заключается в использовании знака подчеркивания для соединения отдельных слов, например `ориентировочное_время_прибытия`. Другой вариант — использование прописных букв в начале каждого слова, что делает более понятным значение всего сжатого многословного имени, например `ОриентировочноеВремяПрибытия`. Последний метод принято называть Pascal casing (регистр Паскаля), поскольку он популяризовался прежде всего теми, кто писал программы на языке Паскаль. Вариантом регистра Паскаля является метод camel casing (верблюжий регистр), который отличается от своего прототипа лишь тем, что первая буква имени всегда является строчной: `ориентировочное ВремяПрибытия`. В этой книге мы склоняемся к использованию регистра Паскаля, но этот выбор во многом — лишь дело вкуса.

5.2. Вопросы и упражнения

1. То, что является примитивом в одном контексте, может, в свою очередь, быть составлено из примитивов другого контекста. Например, инструкция `while` является примитивом в нашем псевдокоде, хотя она является сложной конструкцией из нескольких команд машинного языка. Приведите два примера подобных явлений из области, не связанной с компьютерами.
2. В каком смысле построение процедур является построением примитивов?
3. Алгоритм Евклида позволяет найти наибольший общий делитель двух положительных целых чисел *X* и *Y* с помощью следующего процесса.

   До тех пор, пока значения X и Y отличны от нуля, выполнять деление большей величины на меньшую и присваивать переменным X и Y значения делителя и остатка соответственно. Конечное значение X, если оно существует, является наибольшим общим делителем.

   Запишите этот алгоритм с помощью нашего псевдокода.
4. Опишите набор примитивов, используемых в некотором предмете, отличном от компьютерного программирования.

5.3. Создание алгоритма

Разработка программы состоит из двух различных действий — создания алгоритма ее работы и представления этого алгоритма в виде программы. До настоящего момента мы рассматривали только способы представления алгоритмов и не касались вопроса, как, собственно, эти алгоритмы создаются. Однако создание алгоритма — это, как правило, наиболее сложный этап в процессе разработки программного обеспечения. В конце концов, создать алгоритм решения задачи означает отыскать метод решения этой задачи. Поэтому, чтобы понять, как создаются алгоритмы, необходимо понять процесс решения задач.

Искусство решения задач

Рассмотрение методов решения задач и их подробное изучение не являются аспектами, специфическими только для компьютерных наук. Напротив, это важно практически для любой области науки. Тесная связь между процессом создания алгоритмов и общей проблемой поиска решения задач привела к сотрудничеству специалистов в области компьютерных систем и ученых из других областей науки в поисках лучших методов решения задач. В конечном счете желательно было бы свести проблему решения задач к алгоритмам как таковым, но это оказалось невозможным. (В главе 12 будет показано, что существуют задачи, не имеющие алгоритмических решений.) Таким образом, способность решать задачи в значительной степени является профессиональным навыком, который необходимо развивать, а не точной наукой, которую можно изучить.



Основные положения для запоминания

- **Создание вычислительных артефактов предполагает использование итеративного и часто исследовательского процесса для перевода идей в осязаемую форму.**

Решение задач является скорее искусством, чем наукой. Доказательством этого может служить тот факт, что представленные ниже, весьма расплывчато определенные фазы решения задач, предложенные математиком Дж. Полиа (G. Polya) в 1945 году, остаются теми основными принципами, на которых и сегодня базируется обучение навыкам решения задач.

Фаза 1. Понять существо задачи.

Фаза 2. Разработать план решения задачи.

Фаза 3. Выполнить план.

Фаза 4. Оценить точность решения, а также его потенциал в качестве средства для решения других задач.

Применительно к процессу разработки программ эти фазы выглядят следующим образом.

Фаза 1. Понять существо задачи.

Фаза 2. Предложить идею, какая именно алгоритмическая процедура позволила бы решить задачу.

Фаза 3. Сформулировать алгоритм и представить его в виде программы.

Фаза 4. Оценить точность программы и ее потенциал в качестве средства для решения других задач.

Необходимо подчеркнуть, что предложенные математиком Полиа фазы не являются этапами, которым надо строго следовать при решении задачи. Скорее это те фазы, которые должны быть когда-либо выполнены в процессе ее решения. Здесь ключевое слово — *следовать*, т.е., просто "следуя", вы не сможете решить задачу. Напротив, чтобы решить задачу, необходимо проявлять инициативу и двигаться вперед. Если подходить к решению задачи по принципу "Ну вот, я закончил фазу 1, пора переходить к фазе 2", то вряд ли вам будет сопутствовать успех. Однако если вы с головой окунетесь в решение задачи и в конечном счете решите ее, то, оглянувшись назад, обязательно увидите, что выполнили все четыре фазы, предложенные математиком Полиа.

Кроме того, необходимо заметить, что эти четыре фазы не обязательно выполняются в указанном порядке. Как отмечают многие авторы, те, кто успешно решает задачи, часто начинают формулировать стратегию решения (фаза 2) еще до того, как полностью смогут понять существо задачи (фаза 1). Впоследствии, если выбранные стратегии так и не приведут к успеху (что проявится во время фазы 3 или 4), решающий задачу человек все же приобретет более глубокое понимание сути задачи и, основываясь на этом понимании, сможет вновь вернуться к формулированию других, возможно, более успешных стратегий.

Не забывайте, что здесь мы обсуждаем, как задачи решаются, а не то, как бы нам хотелось, чтобы они решались. В идеале хотелось бы полностью исключить изнурительный по своей сути процесс проб и ошибок, описанный выше. При разработке крупной системы программного обеспечения обнаружение неправильного понимания лишь на фазе 4 может привести к огромным потерям ресурсов. Избежать таких катастроф — основная задача разработчиков программного обеспечения (глава 7), которые традиционно настаивают на том, что полное осмысление задачи должно предшествовать ее решению. Можно возразить, что истинное понимание задачи невозможно, пока не будет найдено ее решение. Тот факт, что решить задачу не удается, подразумевает недостаток ее

понимания. Следовательно, настаивать на том, что нужно вначале полностью осознать задачу, прежде чем предлагать какое-либо ее решение, — это чистый идеализм.

В качестве примера рассмотрим следующую задачу.

> Предположим, что некто *A* хочет определить возраст трех детей некоего *B*. Этот *B* сообщает *A*, что произведение возрастов его детей равно 36. Обдумав эту подсказку, *A* отвечает, что необходима еще подсказка, и *B* сообщает ему сумму возрастов его детей. Затем *A* отвечает, что требуется еще подсказка, и *B* говорит ему, что старший из детей играет на пианино. Услышав эту подсказку, *A* сообщает *B* возраст всех трех его детей. Сколько лет детям?

На первый взгляд может показаться, что последняя подсказка совсем не имеет отношения к задаче, хотя именно она позволила *A* окончательно определить возраст детей. Как это может быть? Давайте двигаться вперед, формулируя план и следуя ему, несмотря на то что у нас еще остается много вопросов по поводу этой задачи. Наш план будет состоять в том, чтобы отслеживать этапы, описанные в условии задачи, учитывая информацию, доступную *A* по мере развития событий.

В первой подсказке сообщалось, что произведение возрастов детей равно 36. Это означает, что искомые значения образуют одну из троек, перечисленных на рис. 5.5, *а*. Следующая подсказка указывала сумму искомых значений. Нам неизвестно, чему именно равна эта сумма, но мы знаем, что этой информации оказалось недостаточно, чтобы *A* смог выбрать правильную тройку. Следовательно, искомая тройка должна быть одной из тех, которые имеют одинаковые суммы в списке на рис. 5.5, *б*. Таких троек на рисунке две: (1, 6, 6) и (2, 2, 9); сумма членов каждой из них равна 13. Это та информация, которая была известна *A* на момент, когда он получил последнюю подсказку. Именно на данном этапе мы осознаем важность последней подсказки. Собственно умение играть на пианино не имеет никакого значения, важен тот факт, что в семье есть *старший* ребенок. Это позволяет отбросить тройку (1, 6, 6) и заключить, что одному ребенку 9 лет, а двум — по 2 года.

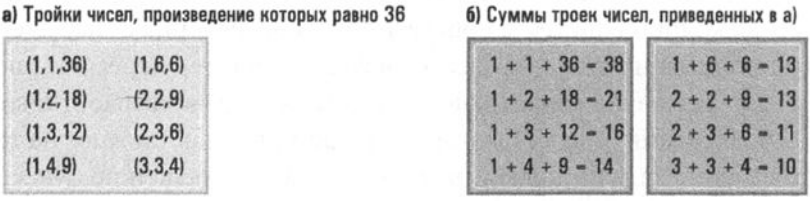


Рис. 5.5. Анализ возможностей

Это именно тот случай, когда, не попытавшись реализовать выбранный план решения (фаза 3), невозможно достичь полного понимания задачи (фаза 1). Если бы мы настаивали на завершении фазы 1, вместо того чтобы двигаться дальше, то, возможно, так и не смогли бы определить возраст детей. Такая неупорядоченность процесса решения задач является основной причиной трудностей, связанных с разработкой систематического подхода к решению задач.

Кроме того, существует и некое мистическое вдохновение, посещающее человека, решающего задачу. Оно проявляется в том, что, работая какое-то время над задачей без видимого успеха, позднее он неожиданно может найти ее решение при выполнении совершенно другого задания. Этот феномен был обнаружен ученым Г. фон Гельмгольцем (H. von Helmholtz) еще в 1896 году, а математик Анри Пуанкаре (Henri Poincare) говорил о нем в своей лекции, прочитанной Психологическому обществу в Париже. Пуанкаре описал свой опыт, связанный с неожиданным осознанием способа решения задачи, которой он безуспешно занимался некоторое время, а затем отложил, обратившись к другим проблемам. Этот феномен отражает процесс, в котором подсознание осуществляет непрерывную работу над решением задачи и в случае успеха выталкивает найденный результат в сознание человека. В наши дни промежуток времени между процессом сознательного решения задачи и внезапным озарением получил название инкубационного периода. Исследования этого явления продолжаются и в настоящее время.

С чего начать

Мы обсудили решение проблем с философской точки зрения, пока не затрагивая вопрос, как следует действовать при попытке решить некоторую задачу. Безусловно, существует множество подходов к решению задач, каждый из которых приводит к успеху в определенных ситуациях. Ниже мы кратко обсудим некоторые из них. Сейчас же просто отметим, что существует нечто общее, присущее всем этим методам. Это можно охарактеризовать как выбор оптимальной отправной точки для дальнейших действий. В качестве примера рассмотрим следующую простую задачу.

> Перед соревнованиями участники *A*, *B*, *C* и *D* сделали следующие прогнозы:
>
> Участник *A* предсказал, что победит участник *B*.
>
> Участник *B* предсказал, что участник *D* будет последним.
>
> Участник *C* предсказал, что участник *A* будет третьим.
>
> Участник *D* предсказал, что сбудется предсказание участника *A*.
>
> Только один из этих прогнозов оказался верным, и это был прогноз победителя. В каком порядке участники *A*, *B*, *C* и *D* закончили соревнования?

Ознакомившись с задачей и проанализировав приведенные в ее условии сведения, нетрудно заметить, что, поскольку прогнозы участников *A* и *D* эквивалентны, а верным оказался только один прогноз, прогнозы участников *A* и *D* неверны. Следовательно, ни участник *A*, ни участник *D* не являются победителями соревнования. Теперь можно считать, что первый шаг сделан и отправная точка найдена. Для получения окончательного решения надо просто продолжить наши рассуждения. Поскольку прогноз участника *A* оказался неверным, участник *B* также не стал победителем. Остается единственный возможный вариант, в котором победителем стал участник *C*. Итак, участник *C* выиграл соревнования, и его прогноз оказался верным. Отсюда можно сделать вывод, что участник *A* занял третье место. Это означает, что участники финишировали либо в порядке *CBAD*, либо в порядке *CDAB*. Но первый вариант нужно отбросить, так как прогноз участника *B* не оправдался. Следовательно, участники финишировали в порядке *CDAB*.

Конечно, сказать, что нужно сделать первый шаг и найти оптимальную отправную точку, — совсем не то же самое, что сказать, как это сделать. Первоначальный прорыв, а затем осознание того, как закрепить полученный успех, чтобы суметь найти окончательное решение задачи, потребуют значительных усилий от того, кто ее решает. Однако существует несколько общих подходов, предложенных математиком Полина и другими исследователями, подсказывающими, как можно осуществить этот первоначальный прорыв. Один из подходов состоит в том, чтобы работать с задачей в обратном порядке. Например, если задача заключается в том, чтобы найти способ получения определенного конечного результата из заданного начального, можно начать поиск с конечного результата и попытаться пройти путь к заданному начальному. Этот подход будет полезен, если потребуется найти алгоритм складывания бумажной птички, упомянутый в предыдущем разделе. Логично начать с попытки развернуть сложенную птичку с целью понять, как она была сделана.

Другой общий подход к решению задачи состоит в поиске связанных с ней проблем, которые или легче решаются, или были решены раньше, а затем в попытке применить способ их решения к данной задаче. Этот метод особенно важен в контексте разработки программ. Часто разработка программы представляет собой не процесс решения конкретного варианта некоторой задачи, а скорее, поиск общего алгоритма, который можно будет использовать для решения всех разновидностей данной задачи. Другими словами, если нам необходимо разработать программу для упорядочения списка имен в алфавитном порядке, наша задача сводится не к сортировке отдельного списка, а к нахождению алгоритма, пригодного для сортировки любого списка имен. Рассмотрим следующий набор инструкций.

```
Поменять местами имена David и Alice.
Поместить имя Carol между именами Alice и David.
Поместить имя Bob между именами Alice и Carol.
```

Этот набор позволяет правильно выполнить сортировку списка, состоящего из имен David, Alice, Carol и Bob, но не является тем общим алгоритмом, который мы ищем. Нам же нужен алгоритм, который в состоянии выполнить сортировку как данного списка, так и любого другого, который может встретиться. Однако найденное решение сортировки конкретного списка не является совершенно бесполезным для разработки общего алгоритма. Мы можем, например, продвинуться в решении, рассматривая такие частные случаи, как попытки найти общий принцип, которые, в свою очередь, послужат основой для создания искомого общего алгоритма. В этом случае искомое решение в конце концов будет найдено с помощью метода решения набора взаимосвязанных частных задач.

Еще один подход к проблеме "с чего начать" заключается в применении метода **поэтапного уточнения**, который предполагает, что задачу не следует пытаться решить сразу же и целиком (во всех ее деталях). Согласно этому методу исследователь должен разбить задачу на ряд подзадач. Идея заключается в том, что при разбиении исходной задачи на подзадачи появляется возможность найти общее решение как последовательность этапов, на каждом из которых решается задача, более простая по сравнению с исходной. Поэтапное уточнение подразумевает также, что каждый из этапов, в свою очередь, можно разбить на меньшие, а те — на еще меньшие, так что в конце концов вся задача сводится к набору легко разрешимых подзадач.

Поэтапное уточнение является **нисходящим методом**, так как его развитие происходит в направлении от общего к частному. В противоположность этому **восходящие методы** предусматривают развитие от частного к общему. Хотя в теории эти подходы противоположны, на практике они просто дополняют друг друга. Например, разбиение задачи, предлагаемое нисходящим методом поэтапного уточнения, обычно осуществляется интуитивно с использованием восходящей модели.

Решениям, полученным с помощью нисходящего метода поэтапного уточнения, свойственна естественная модульная структура, поэтому его, в сущности, можно рассматривать просто как организационный инструмент. Именно в этом кроется основная причина популярности данного метода при разработке алгоритмов, поскольку разработка больших программных систем всегда предполагает значительные организационные усилия и соответствующие мероприятия. Однако, как мы узнаем из главы 7, большие программные системы во все большей степени создаются посредством комбинирования уже готовых, разработанных ранее компонентов, а это подход, который по самой своей сути является восходящим. Таким образом, как нисходящий, так и восходящий подходы к решению задач являются важными инструментами компьютерных наук.

Важность сохранения столь широкой перспективы подчеркивается тем фактом, что использование при поиске решения задач предвзятых мнений и априори предпочитаемых инструментов может в некоторых случаях скрыть, замаскировать их действительную простоту. Яркий тому пример — задача определения возраста детей, обсуждавшаяся выше в этом разделе. Студенты, изучающие алгебру, неизменно пытаются найти ее решение посредством составления системы уравнений, что заводит их в тупик. В результате они попадают в ловушку, весьма характерную для процесса решения задач, — приходят к ошибочному выводу, что предоставленной информации просто недостаточно для решения поставленной задачи.

Еще один аналогичный пример — приведенная ниже задача.

> Когда вы садились в лодку, ваша шляпа упала в воду, но вы этого не заметили. Скорость течения реки — 2,5 км/ч, и ваша шляпа поплыла вниз по течению. Тем временем вы поплыли в лодке вверх против течения со скоростью 4,75 км/час (относительно воды). Спустя 10 минут вы заметили пропажу шляпы, развернули лодку и поплыли вниз по реке догонять шляпу. Через какое время вы ее поймаете?

Большинство студентов, изучающих в высшей школе алгебру, а также любителей карманных калькуляторов начнут решение этой задачи с определения, какое расстояние прошла лодка за 10 минут вверх по течению и какое расстояние за это время шляпа проплыла вниз по течению. Затем они попытаются вычислить, сколько времени понадобится лодке, чтобы пройти вниз по течению до той точки, где находится шляпа. Но ведь пока лодка будет идти к этой точке, шляпа будет уплывать дальше вниз по течению! Таким образом, они, весьма вероятно, попадут в цикл вычислений, где будет находиться шляпа в тот момент, когда лодка достигнет того места, где шляпа была на предыдущем этапе расчета.

На самом же деле задача гораздо проще. Весь фокус состоит в том, чтобы суметь противостоять стремлению писать формулы и делать вычисления. Необходимо просто отложить эти навыки в сторону и взглянуть на задачу в ее истинном свете. Суть всей проблемы — в реке. В данном случае тот факт, что вода в ней движется относительно берегов, не имеет никакого значения. Представьте себе ту же задачу на длинной конвейерной ленте. Сначала решим поставленную задачу, когда лента неподвижна. Если вы, стоя на ленте, положите шляпу у своих ног, а затем будете идти вдоль ленты в течение 10 минут, то вернуться к шляпе вы сможете за те же 10 минут. Теперь включим конвейер. Это будет означать, что сначала вы пойдете против движения ленты. Но поскольку вы, как и шляпа, находитесь на ленте, это не изменит ваших взаимоотношений с лентой и шляпой. Вам по-прежнему потребуется 10 минут, чтобы вернуться к оставленной шляпе.

5.3. Вопросы и упражнения

1. Найдите алгоритм решения следующей задачи и ответьте на дополнительные вопросы.

   а. Для заданного положительного числа n найдите такую комбинацию целых положительных чисел, произведение которых максимально среди всех возможных комбинаций целых положительных чисел, сумма которых равна n. Например, если n равно 4, то искомый список есть (2, 2), так как 2×2 больше, чем $1 \times 1 \times 1 \times 1$, $2 \times 1 \times 1$ и 3×1. Если n равно 5, искомая комбинация будет равна (2, 3).

   б. Какова искомая комбинация для $n = 2001$?

   в. Объясните, как вам удалось продвинуться в решении этой задачи.

2. **а.** Пусть дана квадратная шахматная доска размером $2^n \times 2^n$ клеток, где n — целое положительное число, и коробка с L-образными фишками, каждая из которых может закрыть на доске в точности три клетки. Если вырезать из доски один какой-либо квадрат, сможем ли мы покрыть оставшуюся часть доски фишками таким образом, чтобы они не перекрывались и не выступали за край доски?

   б. Объясните, как предложенное в *а* решение этой задачи может быть использовано, чтобы показать, что $2^{2n} - 1$ делится на 3 для любых целых положительных n.

   в. Как ответы на два предыдущих вопроса связаны с фазами решения задач, предложенными математиком Полиа?

3. Декодируйте следующее сообщение (на английском языке) и объясните, как вам удалось сделать первый шаг в поисках решения:

   Pdeo eo pda yknnayp wjosan.

4. Воспользуетесь ли вы нисходящим методом поэтапного уточнения при попытке собрать пазл неизвестной вам картинки вместо того, чтобы высыпать все элементы на стол и попытаться собрать из них эту картинку? Изменится ли ваш ответ, если вы предварительно взглянете на крышку коробки головоломки, чтобы увидеть, как должна выглядеть собранная картинка?

В результате можно прийти к заключению, что создание алгоритмов является сложным искусством, которое необходимо постоянно совершенствовать, а не изучать как предмет, состоящий из хорошо определенных методологий. Учить решать задачи, следуя лишь четко определенным методологиям, — фактически означает попытку подавить творческие способности учащихся, которые, напротив, нужно всемерно развивать.

5.4. Итерационные структуры

Теперь нашей задачей является изучение некоторых повторяющихся структур, используемых при описании алгоритмических процессов. В этом разделе мы обсудим **итерационные структуры**, в которых выполнение набора инструкций повторяется в циклическом режиме. В следующем разделе рассматривается метод рекурсии. Читатель также сможет познакомиться с некоторыми популярными алгоритмами — последовательного и двоичного поиска, а также с алгоритмом сортировки методом вставки. Начнем с рассмотрения алгоритма последовательного поиска.

Алгоритм последовательного поиска

Рассмотрим задачу поиска в списке некоторого заданного значения. Необходимо разработать алгоритм, позволяющий установить, есть ли заданное значение в списке. Если это значение в списке присутствует, поиск будет считаться успешным, в противном случае будем считать его завершившимся неудачей. Также примем, что исходно список отсортирован согласно некоторому правилу, позволяющему упорядочить его элементы. Например, если это список имен, будем считать, что имена в нем расположены в алфавитном порядке. Если же это список числовых значений, будем полагать, что его элементы расположены в порядке их возрастания.

Для начала попробуем представить, как бы мы действовали при поиске определенного имени в списке гостей, содержащем около 20 фамилий. В данном случае можно было бы просмотреть весь список от начала и до конца, сравнивая каждый его элемент с искомым именем. Если требуемое имя будет найдено, поиск завершится успешно. Однако, если будет достигнут конец списка или обнаружено имя, расположенное по алфавиту после искомого, поиск завершится неудачей. (Не забывайте, что список упорядочен в алфавитном порядке, поэтому если будет найдено имя, расположенное по алфавиту после требуемого, то это означает, что нужного нам имени в списке нет.) Таким образом, наша задача — выполнять последовательный просмотр элементов списка

в направлении сверху вниз до тех пор, пока не будут проверены все имена или пока очередное проанализированное имя не окажется расположенным в алфавитном порядке ниже, чем то, которое нам требуется найти.

С помощью нашего псевдокода этот процесс можно представить следующим образом:

```
выбрать первый элемент списка как ПроверяемоеЗначение
while (ТребуемоеЗначение > ПроверяемоеЗначение and есть еще элементы):
  выбрать следующий элемент списка как ПроверяемоеЗначение
```

По окончании выполнения структуры `while` будет выполнено одно из следующих двух условий: либо будет найдено искомое значение, либо выяснится, что в списке его нет. В любом случае успешность поиска можно установить, сравнив искомое значение с проверяемым. Если они эквивалентны, поиск успешен. Следовательно, для завершения записи алгоритма необходимо добавить инструкцию

```
if (ТребуемоеЗначение == ПроверяемоеЗначение):
  объявить поиск успешным
else:
  объявить поиск неудачным
```

в конец приведенной выше программы.

И наконец, в нашей программе первая инструкция, в которой в качестве проверяемого значения явно указан первый элемент списка, сформулирована в предположении, что проверяемый список содержит как минимум один элемент. Конечно же, можно полагать, что это условие должно выполняться всегда, однако для полной уверенности в правильности программы следует поместить всю составленную выше программу в предложение `else` следующей инструкции `if`:

```
if (список пуст):
  объявить поиск неудачным
else:
  . . .
```

В результате получается функция, текст которой приведен на рис. 5.6. Заметим, что для поиска некоторых значений в списке другие функции вполне могут использовать ее с помощью следующей инструкции.

```
Поиск() список пассажиров для нахождения значения "Даррел Бейкер"
```

Эта инструкция позволяет установить, является ли Даррел Бейкер пассажиром некоторого рейса. Вот еще один пример:

```
Search() список ингредиентов для нахождения значения "мускатный орех"
```

Эта инструкция позволит установить, входит ли мускатный орех в перечень ингредиентов некоторого блюда.

```
def Поиск(Список, ТребуемоеЗначение):
  if (список пуст):
    объявить поиск неудачным
  else:
    выбрать первый элемент списка как ПроверяемоеЗначение
    while (ТребуемоеЗначение > ПроверяемоеЗначение and
        в списке есть еще элементы ):
      выбрать следующий элемент списка как ПроверяемоеЗначение
    if (ТребуемоеЗначение = Проверяемое значение):
      объявить поиск успешным
    else:
      объявить поиск неудачным
```

Рис. 5.6. Алгоритм последовательного поиска, сформулированный с помощью псевдокода

Итак, можно сказать, что представленный на рис. 5.6 алгоритм последовательно рассматривает все элементы списка. По этой причине данный алгоритм называется алгоритмом **последовательного поиска** или, иногда, **линейного поиска**. В силу своей простоты он часто применяется к коротким спискам либо когда это необходимо по каким-то иным соображениям. Однако в случае длинных списков этот метод оказывается менее эффективным, чем другие (в чем мы скоро убедимся).

Управление циклами

Неоднократное использование инструкции или последовательности инструкций представляет собой важную алгоритмическую концепцию. Одним из методов организации такого повторения является итеративная структура, известная как **цикл**; здесь последовательность инструкций, называемая **телом** цикла, многократно выполняется под контролем некоторого управляющего процесса. Типичный пример цикла можно найти в алгоритме последовательного поиска, представленном на рис. 5.6. Здесь инструкция `while` используется в целях управления повторным выполнением единственной инструкции `выбрать следующий элемент списка как ПроверяемоеЗначение`. Общий синтаксис инструкции `while` имеет такой вид:

```
while (условие):
  тело цикла
```

Эта инструкция представляет собой типичный образец циклической структуры, т.е. при ее выполнении циклически совершаются следующие действия:

```
проверить условие
выполнить тело цикла
проверить условие
выполнить тело цикла
.
.
.
проверить условие
```

И так до тех пор, пока заданное условие будет выполняться.

Как правило, использование циклических структур придает алгоритму большую гибкость по сравнению с явным многократным написанием тела цикла. Рассмотрим такой пример. Пусть инструкцию `Добавить каплю серной кислоты` требуется выполнить три раза. В этом случае можно просто записать:

```
добавить каплю серной кислоты
добавить каплю серной кислоты
добавить каплю серной кислоты
```

Однако невозможно написать аналогичную последовательность, эквивалентную следующему циклу:

```
while (уровень pH > 4):
  добавить каплю серной кислоты
```

Суть в том, что мы не знаем заранее, сколько капель серной кислоты понадобится в каждом конкретном случае.

А теперь давайте подробно рассмотрим, как осуществляется управление циклом. Может показаться, что эта часть структуры цикла менее важна, поскольку именно в теле цикла непосредственно выполняются требуемые действия (например, добавляются капли кислоты). Поэтому управляющие действия можно рассматривать просто как надстройку, появившуюся только из-за того, что мы решили повторять выполнение тела цикла. Однако опыт показывает, что именно управление циклом чаще всего служит источником ошибок в циклических структурах и, следовательно, требует особого внимания.

Управление циклом состоит из трех операций: инициализации, проверки и модификации (рис. 5.7), причем все они обязательны для успешного управления циклом. Назначение операции проверки состоит в обеспечении своевременного окончания циклического процесса за счет отслеживания возникновения условия, указывающего, что цикл пора заканчивать. Это условие называют **условием окончания** цикла. Именно для выполнения операции проверки некоторое условие обязательно указывается при записи каждой инструкции `while` нашего псевдокода. Однако условие, задаваемое в инструкции `while`, — это то

условие, при котором тело цикла должно выполняться, а условие окончания — это отрицание условия, заданного в инструкции while. Поэтому в инструкции

```
while (уровень pH > 4):
  добавить каплю серной кислоты
```

условие окончания цикла выглядит следующим образом: "уровень pH *не* превышает 4", а в инструкции while на рис. 5.6 условие окончания цикла можно сформулировать в виде

```
(ТребуемоеЗначение ≤ ПроверяемоеЗначение) or (больше нет элементов
для проверки)
```

Инициализация:	**Установить начальное состояние, которое будет модифицироваться до тех пор, пока не будет выполнено условие окончания**
Проверка:	**Сравнить текущее состояние с условием окончания и прекратить повторение тела цикла, если состояние соответствует условию окончания**
Модификация	**Изменить состояние таким образом, чтобы приблизиться к ситуации, в которой будет выполнено условие окончания**

Рис. 5.7. Операции процедуры управления циклом

Остальные две операции управления циклом гарантируют, что условие окончания обязательно возникнет. Операция инициализации устанавливает начальное условие, а операция модификации изменяет его в направлении достижения условия окончания. Например, на рис. 5.6 операция инициализации выполняется инструкцией, предшествующей инструкции while, где первый элемент списка устанавливается в качестве текущего проверяемого. Операция модификации в этом случае реализуется в теле цикла, когда интересующая нас позиция (проверяемый элемент) перемещается к концу списка. Таким образом, выполнение операции инициализации и многократное выполнение операции модификации приводят к тому, что условие окончания обязательно будет достигнуто. (Либо обнаружится проверяемый элемент, больший либо равный искомому, либо будет достигнут конец списка.)

Следует подчеркнуть, что операции инициализации и модификации обязательно должны приводить к заданному условию окончания. Это является важнейшим требованием организации надлежащего управления циклом, поэтому при разработке циклической структуры следует как минимум дважды убедиться в том, что оно выполняется. Если пренебречь подобной проверкой, то это может привести к ошибкам даже в простейших случаях. Типичным примером могут служить следующие инструкции:

```
Число = 1
while (Число != 6):
  Число = Число + 2
```

Обратите внимание на использование в этом примере оператора языка Python "!=", который читается как "не равно". В данном случае условием окончания является выражение `Число = 6`. Однако переменная `Число` инициализируется значением 1, а затем увеличивается на 2 на каждом этапе модификации. Таким образом, при выполнении цикла переменной `Число` последовательно будут присваиваться значения 1, 3, 5, 7, 9, ... и ее значение никогда не будет равно 6. В результате выполнение данного цикла никогда не закончится.

Порядок, в котором выполняются операции процедуры управления циклом, может незначительно различаться. Фактически существует два варианта широко распространенных циклических структур, которые различаются только порядком выполнения операций управления циклом. Первая представлена инструкцией нашего псевдокода:

```
while (условие):
  действие
```

Семантика этой циклической структуры представлена на рис. 5.8 в виде **блок-схемы**. На подобных схемах для представления отдельных этапов выполнения используются различные геометрические фигуры, а стрелки указывают порядок выполнения этих этапов. Разные фигуры отражают различные типы деятельности, выполняемой на соответствующем этапе. Ромб указывает на принятие решения, а прямоугольник представляет произвольную инструкцию или последовательность инструкций. Обратите внимание, что в данной циклической структуре проверка условия окончания производится до того, как выполняется тело цикла.

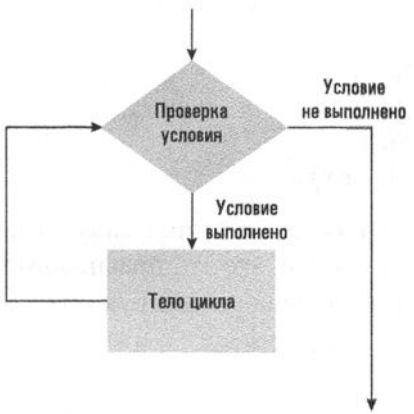


Рис. 5.8. Структура цикла типа `while`

В противоположность этому в структуре, представленной на рис. 5.9, указывается, что тело цикла должно выполняться до проверки условия окончания. В результате тело цикла всегда выполняется хотя бы один раз, в то время как в

структуре типа `while` тело цикла может не выполняться ни разу (если условие окончания будет выполнено при первой же его проверке).



Рис. 5.9. Структура цикла типа `repeat`

В языке Python не существует встроенной структуры для циклов этого типа, хотя совсем несложно построить ее эквивалент, воспользовавшись существующей в этом языке структурой `while` с операторами `if` и `break` в конце ее тела цикла. Для нашего псевдокода мы позаимствуем ключевые слова, существующие в нескольких других языках программирования, для создания следующей семантической формы, представляющей структуру, отображаемую на рис. 5.9.

```
repeat:
  действие
  until (условие)
```

Рассмотрим конкретный пример:

```
repeat:
  взять монету из кармана
  until (в кармане нет монет)
```

Обратите внимание: в этом алгоритме подразумевается, что изначально в кармане есть хотя бы одна монета. Однако это предположение не является обязательным при использовании следующей инструкции:

```
while (в кармане есть монета):
  взять монету из кармана
```

Следуя принятой в нашем псевдокоде терминологии, в дальнейшем мы будем ссылаться на эти структуры как на структуру цикла `while` или структуру цикла `repeat`. В более общем контексте вам может встретиться ссылка на структуру цикла `while` как на цикл **с предварительной проверкой** (или **предпроверкой**) условия, поскольку проверка условия окончания выполняется

в нем до выполнения тела цикла, а ссылка на структуру цикла `repeat` — как на цикл **с последующей проверкой** (или **постпроверкой**) условия, поскольку проверка условия окончания выполняется в нем после выполнения тела цикла.

Хотя в одних алгоритмах при организации управления циклами требуется тщательный анализ действий, выполняемых на этапах инициализации, проверки и модификации, в других достаточно просто следовать некоторым общепринятым простейшим приемам. В частности, при работе со списками данных типичным подходом будет начинать обработку с первого элемента списка и последовательно обрабатывать каждый его элемент, пока список не закончится. Возвращаясь к приведенному выше примеру последовательного поиска мы обнаружим в нем тот же шаблон:

```
выбрать первый элемент списка
while (есть еще элементы для анализа):
  ...
  выбрать следующий элемент списка
```

Поскольку такая структура применяется в алгоритмах очень часто, мы будем использовать для нее синтаксическую форму

```
for Элемент in Список:
  ...
```

представляющую цикл, в котором последовательно обрабатываются все элементы указанного списка. Обратите внимание, что этот примитив псевдокода фактически на один уровень абстракции выше по отношению к структуре `while`, поскольку при ее использовании мы можем достичь того же самого результата, воспользовавшись явно описанными процедурами инициализации, модификации и проверки, однако в данном варианте описание необходимого цикла осуществляется более лаконично, без излишней детализации.

Каждый раз в теле цикла этой структуры `for` в качестве значения переменной `Элемент` будет использоваться следующий элемент списка, указанного под именем `Список`. Условие окончания этого цикла неявно определяется как достижение конца обрабатываемого списка. Например, для вычисления общей суммы элементов списка можно воспользоваться следующей записью.

```
Сумма = 0
for Число in Список:
  Сумма = Сумма + Число
```

В языках, отличных от языка Python, этот тип цикла часто называют циклом **for-each** (для каждого...) и рассматривают как особый случай цикла с предварительной проверкой условия. Структура `for` наилучшим образом подходит к ситуации, когда в алгоритме выполняется одна и та же последовательность действий для каждого элемента списка и нет необходимости отдельно контролировать общее количество повторений цикла.

Алгоритм сортировки методом вставки

В качестве дополнительного примера использования итеративных структур рассмотрим задачу сортировки списка имен в алфавитном порядке. Но прежде чем приступить к обсуждению, следует установить некоторые ограничения, которые необходимо будет учитывать. Попросту говоря, наша задача — отсортировать список "внутри него самого". Другими словами, мы хотим отсортировать список, просто переставляя его элементы, а не перемещая весь список в другое место. Это аналогично сортировке списка, элементы которого записаны на отдельных карточках, разложенных на переполненном рабочем столе. На столе достаточно места для всех карточек, однако запрещается отодвигать другие находящиеся на столе материалы, чтобы освободить дополнительное пространство. Подобное ограничение типично для компьютерных приложений, но не потому, что рабочее пространство в машине обязательно переполнено, как наш рабочий стол, а потому, что мы стремимся использовать доступный объем памяти самым эффективным образом.

Попробуем сделать первый шаг в поисках решения данной задачи. Рассмотрим, как можно было бы отсортировать карточки с именами, расположенные на рабочем столе. Пусть исходный список имен выглядит следующим образом:

Fred
Alex
Diana
Byron
Carol

Один из подходов к сортировке этого списка заключается в следующем. Обратите внимание, что подсписок, состоящий из единственного верхнего имени, Fred, уже отсортирован, а подсписок из двух верхних имен, Fred и Alex, — еще нет. Поэтому поднимем карточку с именем Alex, поместим карточку с именем Fred вниз, туда, где раньше была карточка с именем Alex, а затем переместим карточку с именем Alex в самое верхнее положение, как показано в первой строке на рис. 5.10. Теперь список будет иметь такой вид:

Alex
Fred
Diana
Byron
Carol

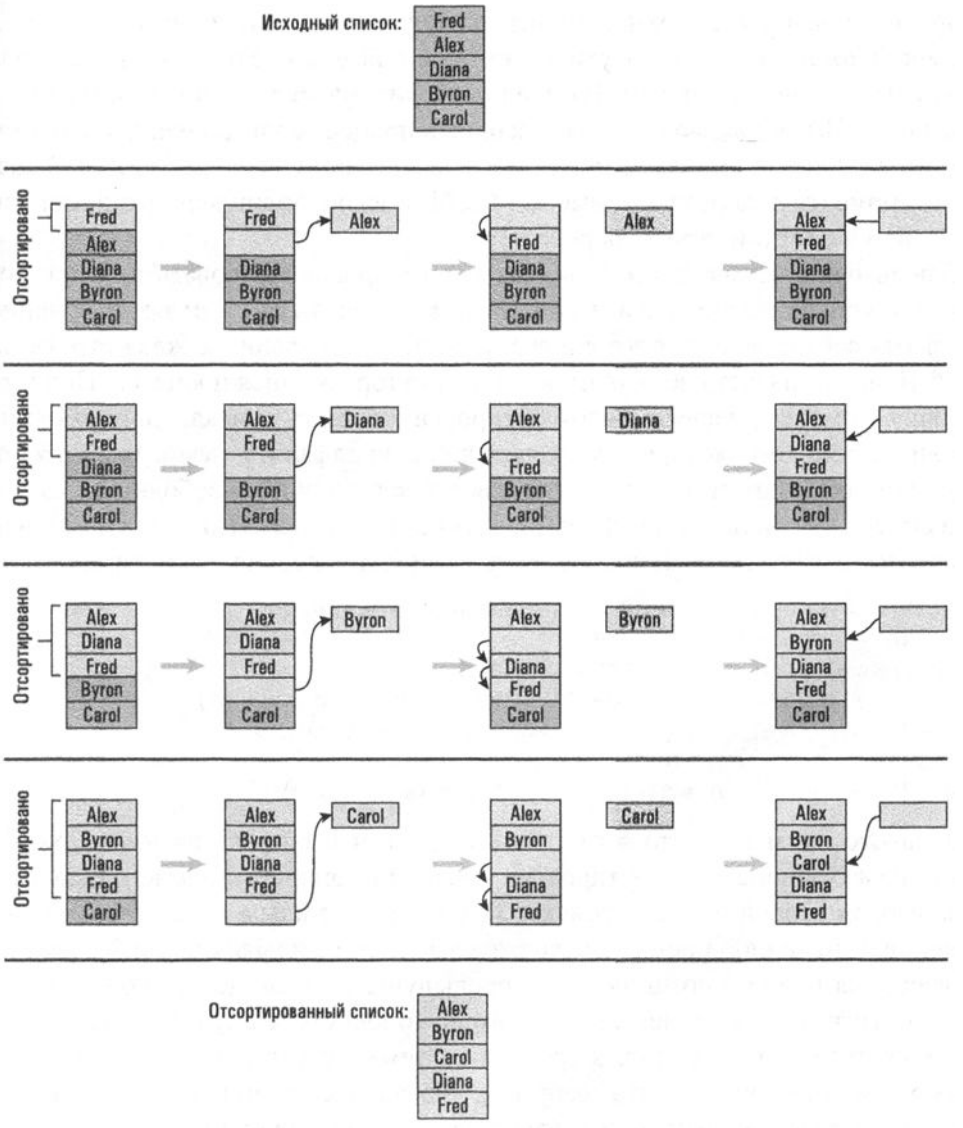


Рис. 5.10. Сортировка списка имен Fred, Alex, Diana, Byron и Carol в алфавитном порядке

В этом варианте два верхних имени образуют отсортированный подсписок, а три верхних — нет. Поднимем третью карточку с именем Diana, опустим карточку с именем Fred туда, где только что была карточка с именем Diana, а затем поместим карточку с именем Diana в ту позицию, которую раньше занимала карточка с именем Fred, как показано во второй строке на рис. 5.10. Теперь три верхних элемента списка образуют отсортированный подсписок. Продолжая действовать таким способом, мы можем получить список, в котором будут

отсортированы четыре верхних элемента. Для этого нужно поднять четвертую карточку с именем Byron, опустить карточки с именами Fred и Diana, а затем поместить карточку с именем Byron в освободившуюся позицию (третья строка на рис. 5.10). И наконец, чтобы завершить процесс сортировки, необходимо поднять карточку с именем Carol, опустить карточки с именами Fred и Diana, а затем поместить карточку с именем Carol в освободившуюся позицию, как показано в четвертой строке на рис. 5.10.

Теперь наша задача состоит в том, чтобы проанализировать процесс сортировки конкретного списка и попытаться обобщить его в целях получения алгоритма сортировки любого списка. С этой точки зрения каждая строка на рис. 5.10 представляет собой один и тот же повторяющийся процесс: "Поднять карточку с именем, первую в неотсортированной части списка, сдвинуть вниз карточки с именами, которые находятся ниже по алфавиту, чем имя на взятой нами карточке, а затем поместить эту взятую карточку на освободившееся место в списке". Если назвать выбранное имя опорным элементом, то с помощью нашего псевдокода данный процесс можно описать следующим образом.

```
переместить опорный элемент во временное хранилище,
  оставив в списке пустое место
while (над пустым местом есть имя and
    оно по алфавиту размещается ниже опорного элемента):
  переместить имя, находящееся над пустым местом, вниз,
    оставив в его прежней позиции пустое место
поместить опорный элемент на пустое место в списке
```

Обратите внимание, что этот процесс должен выполняться многократно. Чтобы начать процедуру сортировки, в качестве опорного элемента должен быть выбран второй элемент списка. Затем перед каждым последующим выполнением описанной процедуры должен выбираться новый опорный элемент, находящийся на одну позицию ниже предыдущего, и так до тех пор, пока не будет достигнут конец списка, т.е. положение опорного элемента должно перемещаться от второго элемента к третьему, затем — к четвертому и т.д. Следуя этому, мы можем организовать требуемое управление путем повторения процедуры с помощью следующей последовательности инструкций.

```
N = 2
while (значение N не превышает длину списка):
  выбрать N-й элемент списка в качестве опорного элемента
    .
    .
    .
  N = N + 1
```

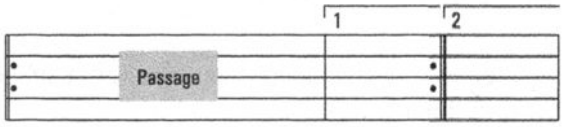
Здесь N — счетчик, параметр длины списка определяет количество элементов в списке, а точки указывают место, где должен располагаться составленный нами выше фрагмент программы.

Итеративные структуры в музыке

В музыке итеративные структуры использовали и программировали за много столетий до того, как это стали делать специалисты в области компьютерных наук. Действительно, структура песни (состоящая из нескольких куплетов, за каждым из которых следует припев) может быть представлена с помощью следующей инструкции `while`:

```
while (остались куплеты):
  исполнить следующий куплет
  исполнить припев
```

Более того, запись



представляет собой способ описания композитором такой структуры:

```
N = 1
while (N < 3):
  играть пассаж
  играть концовку N
  N = N + 1
```

Полный текст программы сортировки на языке псевдокода приведен на рис. 5.11. Не вдаваясь в подробности, можно сказать, что эта программа сортирует список, многократно повторяя следующие действия: "Элемент извлекается из списка, а затем вставляется на надлежащее ему место". Именно по причине многократного повторения вставки элементов в список данный алгоритм получил название **сортировки методом вставки**.

```
def Сортировка(Список):
  N = 2
  while (значение N не превышает длину списка):
    выбрать N-й элемент списка в качестве опорного элемента
    переместить опорный элемент во временное хранилище,
      оставив в списке пустое место
    while (над пустым местом есть имя and
        оно по алфавиту размещается ниже опорного элемента):
      переместить имя, находящееся над пустым местом, вниз,
        оставив в его прежней позиции пустое место
    поместить опорный элемент на пустое место в списке
    N = N + 1
```

Рис. 5.11. Алгоритм сортировки методом вставки, написанный на псевдокоде

Обратите внимание, что представленная на рис. 5.11 структура содержит цикл, помещенный внутрь другого цикла. Внешний цикл представлен первой инструкцией `while`, а внутренний цикл — второй инструкцией `while`. Каждое выполнение тела внешнего цикла приводит к тому, что внутренний цикл инициализируется и выполняется до тех пор, пока не будет выполнено условие его окончания. Таким образом, однократное выполнение тела внешнего цикла будет сопровождаться многократным выполнением тела внутреннего цикла.

При инициализации управления внешним циклом начальное значение счетчика `N` устанавливается с помощью инструкции

```
N = 2
```

Операция модификации этого цикла включает увеличение значения счетчика `N` в конце тела цикла с помощью инструкции

```
N = N + 1
```

Условие окончания внешнего цикла выполняется, когда значение счетчика `N` превышает длину сортируемого списка.

Управление внутренним циклом инициализируется, когда опорный элемент извлекается из списка и в нем образуется пустое место. Операция модификации включает перемещение расположенных выше элементов на пустое место вниз, в результате чего свободное место перемещается вверх по списку. Условие окончания выполняется, когда пустое место или находится непосредственно под именем, которое по алфавиту размещается выше опорного значения, или же достигает верхней позиции списка.

5.4. Вопросы и упражнения

1. Преобразуйте показанную на рис. 5.6 процедуру последовательного поиска так, чтобы она могла работать с неотсортированными списками.
2. Перепишите приведенную ниже программу на псевдокоде так, чтобы в ней использовались повторяющиеся инструкции.

```
Z = 0
X = 1
while (X < 6):
  Z = Z + X
  X = X + 1
```

3. Некоторые из популярных в настоящее время языков программирования используют синтаксис

```
while (. . .) do (. . .)
```

для представления цикла с предварительной проверкой (предпроверкой) условия и синтаксис

```
do (. . .) while (. . .)
```

для представления циклов с последующей проверкой (постпроверкой) условия. Хотя подобный подход выглядит довольно элегантным, какие проблемы могут быть вызваны столь большим сходством двух вариантов цикла?

4. Предположим, что процедура сортировки методом вставки, представленная на рис. 5.11, была применена к списку Gene, Cheryl, Alice и Brenda. Опишите, как будет выглядеть список каждый раз по окончании выполнения тела внешней структуры `while`.
5. Почему не следует заменять фразу "по алфавиту размещается ниже" в инструкции `while` на рис. 5.11 фразой "по алфавиту размещается ниже или эквивалентно"?
6. Вариантом алгоритма сортировки методом вставки является **выборочная сортировка**. В этом алгоритме сначала выбирается наименьший элемент списка и помещается на первое место. Затем выбирается наименьший элемент из оставшихся элементов списка и помещается на второе место в списке. Многократно выбирая наименьший элемент из оставшейся части списка и перемещая его вперед, мы увеличиваем отсортированную часть списка, находящуюся в начале, тогда как его конечная часть, состоящая из неотсортированных элементов, сжимается. С помощью нашего псевдокода напишите процедуру сортировки списка с использованием алгоритма выборочной сортировки, аналогичную процедуре, представленной на рис. 5.11.
7. Другой известный алгоритм сортировки — сортировка **методом пузырька**. Он основан на процессе повторяющегося сравнения двух стоящих рядом имен и перестановки их местами, если они находились относительно друг друга в порядке, отличном от требуемого. Предположим, что сортируемый список состоит из n элементов. Сортировка методом пузырька начнется сравнением (и, возможно, перестановкой) элементов, стоящих на местах n и $n - 1$. Затем сравниваются элементы, стоящие на местах $n - 1$ и $n - 2$, и так далее в направлении к началу списка, пока не будет выполнено сравнение (и, возможно, перестановка) первого и второго элементов списка. В результате прохождения по всему списку его наименьший элемент будет вынесен на первое место. Аналогичным образом после второго прохождения списка следующий по величине элемент будет вынесен на второе место и т.д. Таким образом, пройдя список $n - 1$ раз, мы отсортируем его целиком. (Если визуально представить себе работу данного алгоритма, то создается впечатление, что наименьшие из оставшихся неупорядоченных элементов списка последовательно всплывают к его вершине как пузырьки, — отсюда и название алгоритма.) Используя наш псевдокод, напишите процедуру сортировки списка методом пузырька по аналогии с процедурой, представленной на рис. 5.11.



Основные положения для запоминания

- **Для решения одной и той же задачи могут быть разработаны разные алгоритмы.**

5.5. Рекурсивные структуры

Рекурсивные структуры представляют собой альтернативу парадигме циклических структур в отношении реализации повторяющихся действий. В то время как цикл предполагает повторение набора инструкций таким образом, что весь этот набор полностью выполняется, а затем его выполнение повторяется, рекурсивная структура предполагает повторение набора инструкций как подзадачу самой себя. В качестве иллюстрации рассмотрим процесс совершения интерактивной покупки в браузере, допускающем использование нескольких вкладок. Знакомясь с описанием некоторого товара на одной веб-странице, пользователь может обнаружить ссылки на еще один или более интересующих его товаров, достаточно привлекательных, чтобы открыть в новых вкладках браузера те веб-страницы, на которые эти ссылки указывают. Далее он откладывает незавершенную транзакцию покупки первого товара на то время, пока знакомится с содержимым одной или более вновь открытых страниц. Закончив просмотр этих страниц, пользователь закрывает соответствующие вкладки и возвращается к первой странице, чтобы завершить выполнение транзакции. В результате во время просмотра одной веб-страницы ему удалось параллельно ознакомиться еще с несколькими. Однако этот процесс проходил не по схеме линейной циклической структуры, когда страницы открываются, прочитываются и закрываются одна за другой. В действительности ознакомление с очередной новой страницей проходило тогда, когда работа с предыдущей еще не достигла своего завершения. Иначе говоря, возврат к продолжению выполнения внешней задачи осуществлялся только после завершения выполнения одной или нескольких вложенных задач.

Алгоритм двоичного поиска

В качестве удобного средства ознакомления с парадигмой рекурсии давайте вновь рассмотрим задачу поиска заданного элемента в отсортированном списке. Но на этот раз подойдем к этому несколько иначе — попытаемся использовать процедуру, которой мы обычно следуем при поиске нужного слова в словаре. Никто в таких случаях не просматривает весь словарь последовательно, элемент за элементом или даже страница за страницей. Мы просто открываем его примерно в том месте, где, как мы думаем, может находиться нужное слово.

Если повезет, оно окажется именно там, в противном случае поиск придется продолжить. Однако в этой точке мы уже существенно сузим область поиска.

Безусловно, при поиске в словаре у нас есть определенное преимущество: мы заранее знаем, где примерно может находиться искомое слово. Например, если искать слово "сомнамбула", то оно, скорее всего, будет находиться ближе к последним страницам словаря. Однако в общем случае при поиске в списках у нас не будет подобного преимущества, поэтому давайте договоримся всегда начинать поиск с элемента, расположенного в "середине" списка. Слово *середина* здесь указано в кавычках в том смысле, что в списке может быть четное количество элементов, а значит, элемента, расположенного точно в середине списка, в этом случае просто не будет. Договоримся в подобных случаях считать средним тот элемент, который окажется первым во второй половине списка.

Если средний элемент списка является искомым, мы можем объявить поиск успешным. В противном случае мы можем хотя бы ограничить процесс поиска первой или второй половиной списка в зависимости от того, является ли искомое значение меньше или больше чем анализируемый средний элемент (не забывайте — список является отсортированным).

Чтобы продолжить поиск в оставшейся части списка, мы вполне можем воспользоваться методом последовательного поиска, однако вместо этого давайте попробуем применить к оставшейся части списка тот же самый подход, который ранее применили ко всему списку. Иначе говоря, выберем средний элемент в оставшейся части списка и проанализируем его значение. Как и ранее, если это то, что мы ищем, работу можно считать выполненной. В противном случае вновь можно ограничить область поиска еще меньшей частью списка.

Описанный подход к процедуре поиска представлен на рис. 5.12, на котором рассматривается процедура поиска элемента John в списке имен, приведенном в левой части рисунка. Сначала анализируется средний элемент списка Harry. Поскольку искомый элемент должен быть расположен в списке ниже этого значения, продолжим поиск в нижней половине исходного списка. Средним элементом этой части списка является Larry. Так как искомый элемент должен находиться выше значения Larry, следует продолжить поиск, анализируя верхнюю половину текущей части списка. Выбрав средний элемент второго подсписка, мы обнаруживаем искомое значение John, а значит, можем объявить поиск успешно завершенным. Если обобщить сказанное, то наша стратегия состоит в последовательном делении анализируемого списка на меньшие сегменты до тех пор, пока искомый элемент не будет найден либо пока очередной сегмент не окажется пустым.

Следует особо подчеркнуть последнюю фразу. Если искомый элемент отсутствует в исходном списке, то обсуждавшийся выше метод поиска приведет к последовательному делению исходного списка на все более и более мелкие

сегменты до тех пор, пока очередной анализируемый сегмент не окажется пустым. Именно эта ситуация указывает на то, что поиск завершился неудачей, т.е. искомый элемент в списке отсутствует.

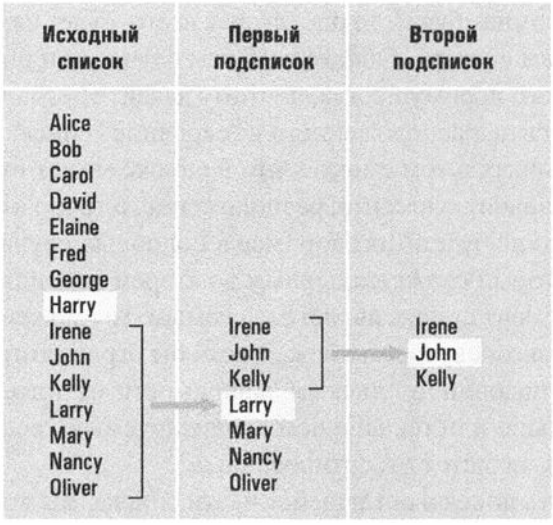


Рис. 5.12. Применение выбранной стратегии к поиску в списке имен элемента John

На рис. 5.13 представлен первый, предварительный, вариант обсуждавшегося выше алгоритма поиска, записанный с помощью нашего псевдокода. В нем предлагается начать поиск с анализа того, не является ли список пустым. Если это так, можно сразу выдать сообщение о том, что поиск завершился неудачей. В противном случае следует проанализировать "средний" элемент списка. Если этот элемент не является искомым, следует продолжить поиск либо в верхней, либо в нижней половине списка. Оба эти варианта требуют повторного выполнения поиска. Было бы очень удобно выполнить и тот, и другой поиск посредством вызова некоего вспомогательного абстрактного инструмента. Поэтому для решения данной задачи был выбран вариант вызова некой функции с именем `Поиск`, которая и должна будет осуществить повторный поиск в обоих случаях. Следовательно, для завершения нашей программы мы должны предоставить этот абстрактный инструмент, т.е. написать псевдокод указанной функции.

Однако эта функция должна выполнять те же самые действия, которые мы описали выше средствами псевдокода. Она точно так же сначала должна проверить, не является ли переданный ей список пустым, и если это не так, то продолжить работу и проанализировать средний элемент этого списка. Следовательно, мы можем определить требуемую функцию, просто присвоив уже

написанному нами ранее блоку псевдокода имя Поиск, а затем вставив обращение к этой функции в тех местах, где требуется выполнение повторного поиска. Полученный результат представлен на рис. 5.14.

```
if (список пуст):
  сообщение о том, что поиск завершился неудачей
else:
  ПроверяемоеЗначение = "средний" элемент списка
  if (ТребуемоеЗначение == ПроверяемоеЗначение):
    сообщение о том, что поиск завершился успешно
  if (ТребуемоеЗначение < ПроверяемоеЗначение):
    Поиск() требуемого значения в части списка, предшествующей
      проверяемому значению, и сообщение о результатах этого поиска
  if (Требуемое значение > ПроверяемоеЗначение):
    Поиск() требуемого значения в части списка, следующей за
      проверяемым значением, и сообщение о результатах этого поиска
```

Рис. 5.13. Первый, предварительный, вариант алгоритма двоичного поиска

```
def Поиск (Список, ТребуемоеЗначение):
  if (список пуст):
    сообщение о том, что поиск завершился неудачей
  else:
    ПроверяемоеЗначение = "средний" элемент списка
    if (ТребуемоеЗначение == ПроверяемоеЗначение):
      сообщение о том, что поиск завершился успешно
    if (ТребуемоеЗначение < ПроверяемоеЗначение):
      Подсписок = часть списка,предшествующая проверяемому
        значению
      Поиск(Подсписок, ТребуемоеЗначение)
    if (ТребуемоеЗначение > ПроверяемоеЗначение):
      Подсписок = часть списка, следующая за проверяемым
        значением
      Поиск(Подсписок, ТребуемоеЗначение)
```

Рис. 5.14. Алгоритм двоичного поиска, написанный на псевдокоде

Обратите внимание, что эта функция содержит ссылку на саму себя. Когда при выполнении этой функции мы дойдем до инструкции

```
Поиск(. . .)
```

потребуется вновь применить ту же самую функцию, но уже к меньшей части того списка, который обрабатывался при исходном выполнении функции. Если этот поиск завершится успешно, мы можем объявить, что и поиск в исходном выполнении завершился успешно. Если же вторичный поиск завершится неудачей, то мы можем объявить, что и исходный поиск завершился неудачей.

Поиск и сортировка

Алгоритмы последовательного и двоичного поиска — это всего лишь два представителя большого семейства алгоритмов, осуществляющих поисковый процесс. Аналогично сортировка методом вставки — это лишь один из многих существующих алгоритмов сортировки. Другими классическими алгоритмами являются сортировка слиянием (обсуждается в главе 12), выборочная сортировка (ее описание можно найти в подразделе "Вопросы и упражнения", п. 6, раздела 5.4), сортировка методом пузырька (см. подраздел "Вопросы и упражнения", п. 7, раздела 5.4), быстрая сортировка (применяющая к процессу сортировки принцип "разделяй и властвуй") и древовидная сортировка (использующая искусную методику для нахождения элементов, которые следует переместить вверх по списку). Описание этих алгоритмов вы сможете найти в книгах, указанных в списке дополнительной литературы в конце главы.

Чтобы увидеть, как представленная на рис. 5.14 процедура выполняет свою задачу, попробуем с ее помощью определить, содержится ли значение `Bill` в списке имен `Alice`, `Bill`, `Carol`, `David`, `Evelyn`, `Fred`, `George`. Поиск начинается с выбора в качестве проверяемого элемента имени `David` (среднего элемента списка). Так как искомое значение (`Bill`) по алфавиту предшествует проверяемому, следует применить процедуру `Поиск` к списку элементов, предшествующих имени `David`, т.е. к списку `Alice`, `Bill`, `Carol`. Для этого нам потребуется создать вторую копию процедуры `Поиск`, предназначенную для решения данной промежуточной задачи.

Теперь мы имеем две выполняющиеся копии нашей процедуры поиска, как показано на рис. 5.15. Дальнейшее выполнение исходной копии процедуры временно приостановлено на следующей инструкции:

```
Поиск(Подсписок, ТребуемоеЗначение)
```

Вторая копия процедуры используется для поиска имени `Bill` в списке `Alice`, `Bill`, `Carol`. Завершив вторую процедуру двоичного поиска, мы аннулируем ее копию и сообщаем полученные в ней результаты исходной копии, после чего выполнение исходной копии будет продолжено с указанного места. Таким образом, вторая копия процедуры функционирует как подчиненная исходной, выполняя задачу, запрошенную исходной копией, а затем исчезая.

Вторичная процедура поиска выбирает имя `Bill` в качестве проверяемого значения, так как это средний элемент в списке `Alice`, `Bill`, `Carol`. Поскольку он совпадает с искомым значением, поиск объявляется успешным и вторичная процедура завершает свою работу.

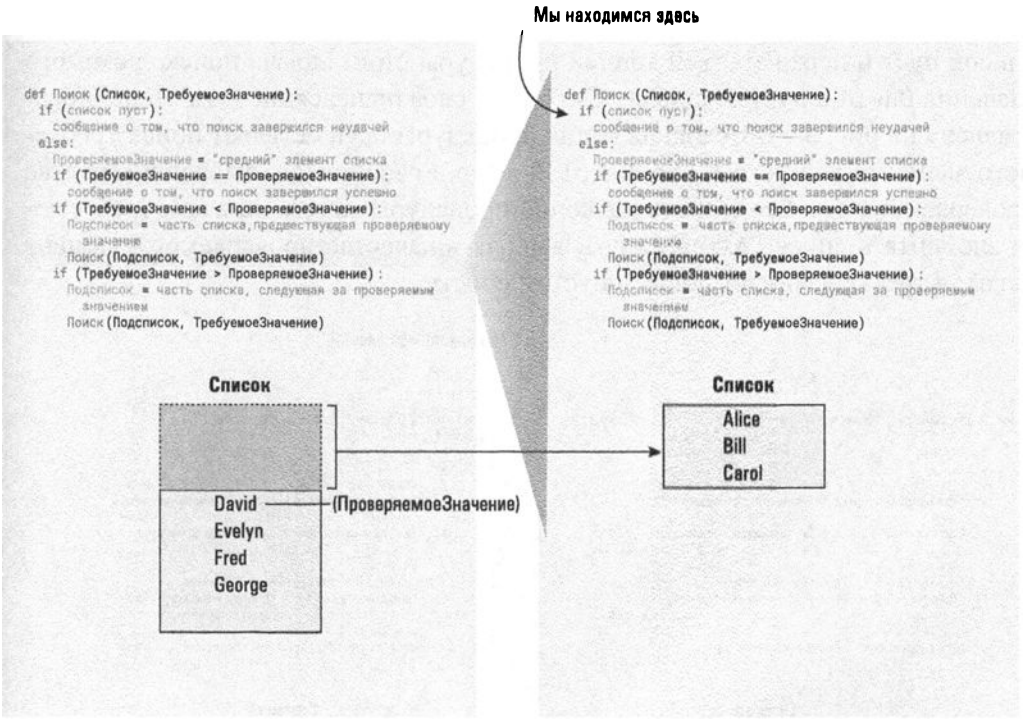


Рис. 5.15. Рекурсивный поиск

Теперь вторичный поиск, запрошенный исходной копией процедуры, завершен, и выполнение этой исходной копии может быть продолжено. В очередной инструкции указано, что, поскольку вторичный поиск был успешным, следует объявить успешным и исходный поиск. В результате выполнения всего процесса было совершенно справедливо установлено, что имя `Bill` присутствует в списке имен `Alice`, `Bill`, `Carol`, `David`, `Evelyn`, `Fred`, `George`.

Теперь давайте посмотрим, что произойдет, если перед представленной на рис. 5.14 процедурой поставить задачу определить наличие в списке `Alice`, `Carol`, `Evelyn`, `Fred`, `George` элемента `David`. На этот раз исходная копия процедуры выбирает в качестве проверяемого значения имя `Evelyn` и определяет, что искомое значение должно находиться в предшествующей части списка. Поэтому она вызывает еще одну копию процедуры для поиска в списке тех элементов, которые стоят перед именем `Evelyn`, т.е. в двухэлементном списке, состоящем из имен `Alice` и `Carol`. Ситуация на этой стадии выполнения алгоритма представлена на рис. 5.16.

Вторая копия процедуры `Поиск` выберет в качестве проверяемого элемента имя `Carol` и определит, что искомое значение должно находиться после него. Процедура вызовет третью копию процедуры для поиска требуемого элемента

в списке имен, следующих за именем Carol в списке Alice, Carol. Однако этот список пуст и перед третьей копией процедуры стоит задача поиска искомого значения David в пустом списке. Ситуация, сложившаяся на этом этапе, представлена на рис. 5.17. Исходная копия процедуры осуществляет поиск требуемого элемента в списке Alice, Carol, Evelyn, Fred, George, выбрав в качестве проверяемого имя Evelyn; вторая копия процедуры занята поиском требуемого элемента в списке Alice, Carol, выбрав в качестве проверяемого элемент Carol; а третья начинает поиск в пустом списке.

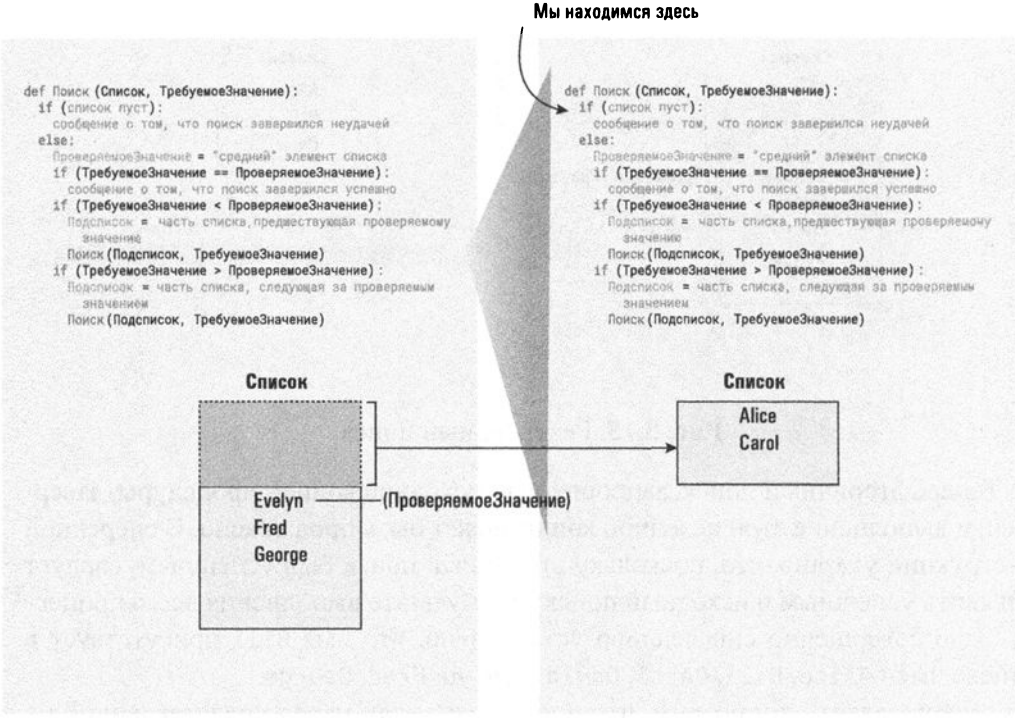


Рис. 5.16. Еще один пример рекурсивного поиска, первая ситуация

Безусловно, третья копия процедуры тут же объявляет свой поиск неудачным и завершается. После этого вторая копия может продолжить свою работу. Она обнаруживает, что запрошенный поиск оказался неудачным, поэтому также объявляет свой поиск неудачным и завершается. Исходная копия процедуры, ожидавшая поступления сообщения от второй копии, теперь может продолжить свою работу. Так как запрошенный поиск оказался неудачным, она тоже объявляет свой поиск неудачным, после чего завершается. Таким образом, наша программа пришла к правильному заключению, что имя David не содержится в списке имен Alice, Carol, Evelyn, Fred, George.

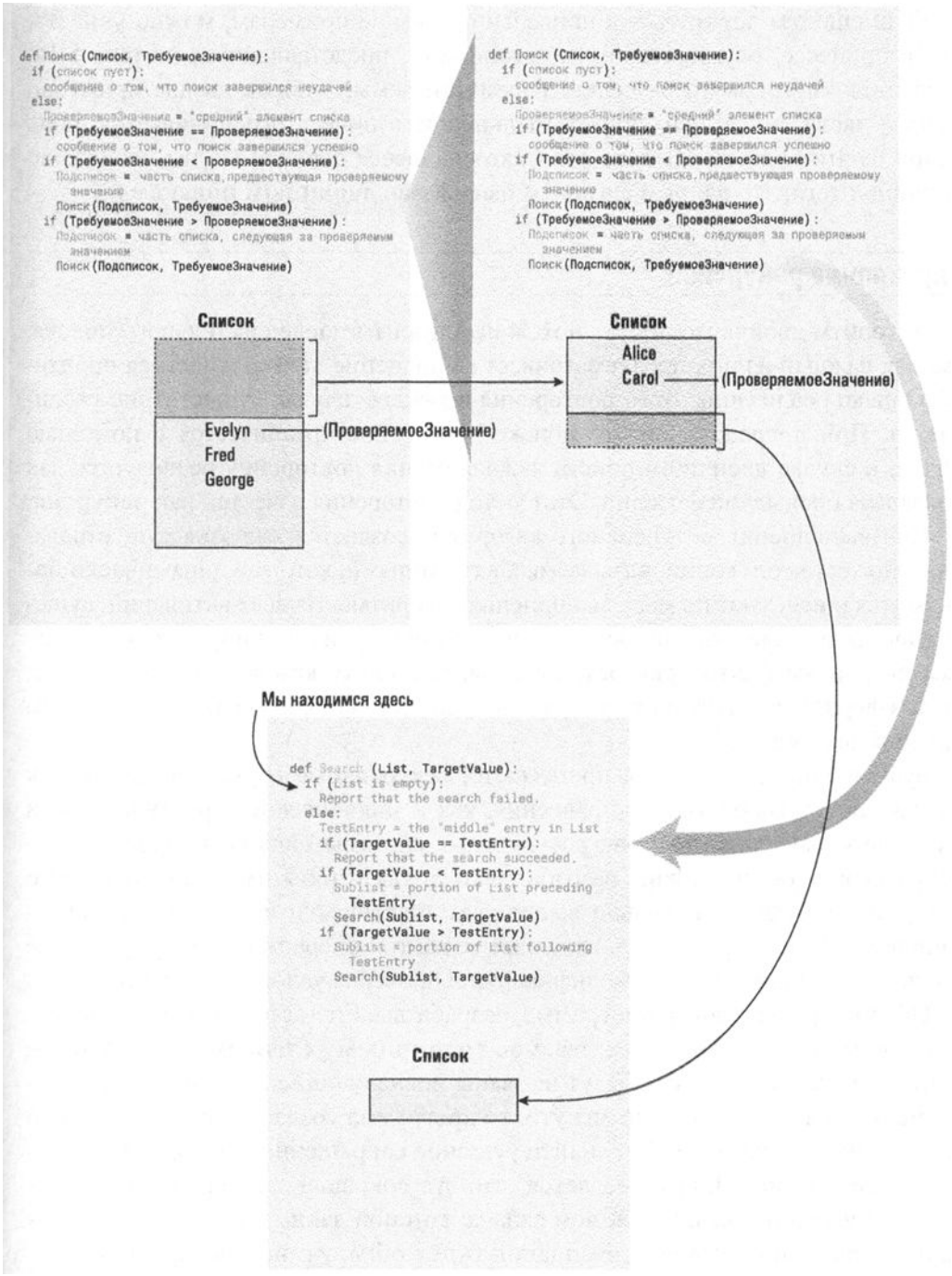


Рис. 5.17. Вторая ситуация: в рекурсивном поиске получен пустой список

Если еще раз обратиться к приведенным выше примерам, можно увидеть, что в процессе, осуществляемом алгоритмом, представленным на рис. 5.14, требуется многократно разделять рассматриваемый список на две примерно равные части, после чего область дальнейшего поиска ограничивается лишь одной из этих частей. Именно это повторяющееся деление на два послужило причиной того, что данный алгоритм был назван **двоичным поиском**.

Управление рекурсией

Алгоритм двоичного поиска похож на алгоритм последовательного поиска, так как каждый из них предусматривает выполнение повторяющегося процесса. Однако реализация этого повторения в каждом случае существенно различается. При последовательном поиске повторение организуется с помощью цикла, в случае двоичного поиска каждая стадия повторения реализуется как подзадача предыдущей стадии. Этот метод повторения известен как **рекурсия**.

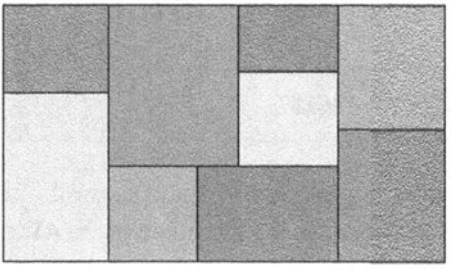
При выполнении рекурсивного алгоритма создается иллюзия существования множества его копий, называемых активациями, которые динамически появляются и исчезают по мере выполнения алгоритма. Из всех активаций, существующих в заданный момент времени, активно функционирует только одна. Все остальные фактически остановлены, поскольку каждая из них ожидает, пока завершится следующая запущенная ею активация, и только после этого продолжает свою работу.

Будучи повторяющимися процессами, рекурсивные структуры почти так же зависят от корректного управления, как и циклические структуры. Как и в случае управления циклами, рекурсивные системы зависят от проверки условия окончания и должны разрабатываться так, чтобы иметь гарантии, что это условие будет обязательно выполнено. Фактически правильно организованное управление рекурсией включает те же три операции, что и управление циклом, — инициализация, модификация и проверка условия окончания.

Обычно рекурсивная программа разрабатывается так, чтобы проверять условие окончания (часто называемое **граничным условием** или **условием вырождения**) до того, как будут вызваны последующие активации. Если условие окончания еще не достигнуто, то программа создает следующую свою активацию, задача которой — найти решение сокращенной версии задачи текущей активации. Подразумевается, что эта сокращенная версия находится ближе к условию окончания, чем задача, которой занимается текущая активация. Как только условие окончания будет обнаружено, выбирается путь, вызывающий завершение текущей активации без создания дополнительных активаций.

Рекурсивные структуры в изобразительном искусстве

С помощью приведенной ниже рекурсивной функции на прямоугольном холсте можно создавать картины в стиле нидерландского живописца Пита Мондриана (Piet Mondrian, 1872–1944). На его картинах прямоугольный холст разделен на последовательно уменьшающиеся прямоугольники. Следуя данной процедуре, попытайтесь самостоятельно создать картины, аналогичные изображенной на рисунке. Начните с прямоугольника, представляющего весь холст, на котором вы работаете. (Если у вас возникнут сомнения, является ли алгоритм, представленный этой функций, таким, который полностью отвечает определению, приведенному в разделе 5.1, ваши подозрения будут вполне обоснованными. В действительности это пример недетерминированного алгоритма, поскольку здесь есть места, в которых исполнитель или машина, выполняющая эту функцию, вынуждена будет принимать "творческие" решения. Возможно, именно по этой причине результаты работы Мондриана рассматриваются как искусство, тогда как наши результаты таковыми не признаются.)



```
 def Mondrian (Прямоугольник):
if (размер прямоугольника слишком велик для вашего
      художественного восприятия):
   разделите прямоугольник на два меньших прямоугольника
   примените функцию Mondrian к первому из двух меньших
     прямоугольников и закрасьте его некоторым цветом
   примените функцию Mondrian ко второму из двух меньших
     прямоугольников и закрасьте его другим цветом
```

Теперь посмотрим, как операции инициализации и модификации механизма управления повторением реализованы в рекурсивной программе двоичного поиска, представленной на рис. 5.14. В этом случае создание дополнительных активаций прекращается, когда обнаруживается искомое значение или задача сужается до поиска в пустом списке. Процесс инициализируется неявно, посредством задания исходного списка и искомого значения. Начиная с этой конфигурации, программа модифицирует свою задачу, что приводит к поиску во все уменьшающемся списке. Поскольку длина исходного списка конечна, а каждый этап модификации уменьшает длину рассматриваемого списка, можно

гарантировать, что либо искомое значение будет найдено, либо задача сузится до поиска в пустом списке. Следовательно, можно сделать заключение, что процесс повторения гарантированно прекратится.

И наконец, поскольку как циклические, так и рекурсивные управляющие структуры представляют собой способ организации повторения выполнения набора инструкций, можно попробовать установить, одинаковы ли их возможности. То есть, если некоторый алгоритм разработан с использованием циклической структуры, можно ли для решения этой же задачи разработать другой алгоритм, применяющий только рекурсивные методы, и наоборот. Такие вопросы важны с точки зрения компьютерных наук, так как ответы на них позволяют понять, какие функции необходимо реализовать в языке программирования, чтобы получить как можно более мощную систему разработки программ. Мы вернемся к этой проблеме в главе 12, где будут рассматриваться некоторые теоретические аспекты компьютерных наук и их математические основы. Опираясь на сделанные в этой главе выводы, в приложении Д будет доказана эквивалентность итеративных и рекурсивных структур.

5.5. Вопросы и упражнения

1. Какие имена будут проверены программой двоичного поиска (рис. 5.14) при поиске имени `Joe` в списке имен `Alice`, `Bob`, `Carol`, `David`, `Evelyn`, `Fred`, `George`, `Henry`, `Irene`, `Joe`, `Karl`, `Larry`, `Mary`, `Nancy` и `Oliver`?
2. Какое максимальное количество элементов может потребоваться проверить при выполнении двоичного поиска в списке из 200 элементов? А в списке из 100 000 элементов?
3. Какая последовательность чисел будет напечатана при выполнении следующей рекурсивной функции, если начать ее выполнение с присвоения переменной `N` значения `1`?

```
def Упражнение (N):
  print(N)
  if (N < 3):
    Упражнение(N + 1)
  print(N)
```

4. Что является условием окончания в рекурсивной функции, представленной в предыдущем вопросе этого раздела?

5.6. Эффективность и правильность

В этом разделе мы представляем две темы, которые составляют важные области исследований в компьютерных науках. Первая из них — эффективность, а вторая — правильность алгоритмов.

Эффективность алгоритма

Хотя современные машины способны выполнять миллионы и даже миллиарды операций в секунду, эффективность по-прежнему остается важнейшим аспектом разработки алгоритмов. Зачастую выбор между эффективным и неэффективным решениями задачи может на самом деле означать выбор между реализуемым и нереализуемым способами ее решения.

Рассмотрим задачу, с которой сталкивается секретарь университета при поиске и заполнении личных дел студентов. Хотя в университете на протяжении любого семестра фактически числится около 10 000 студентов, секретарю в действительности приходится иметь дело более чем с 30 000 личных дел, поскольку за несколько предыдущих лет многие из студентов зарегистрировались для изучения хотя бы одной из преподаваемых в университете дисциплин, но не смогли закончить цикл обучения. Теперь предположим, что все личные дела хранятся в компьютере секретаря в виде списка, упорядоченного по идентификационным номерам каждого из студентов. Чтобы найти личное дело некоторого студента, секретарь должен выполнить поиск по его идентификационному номеру в общем списке.

Мы уже познакомились с двумя алгоритмами поиска в подобных списках — последовательным и двоичным поиском. Сейчас нам нужно дать ответ на вопрос, почувствует ли секретарь разницу между этими двумя алгоритмами? Начнем с рассмотрения последовательного поиска.

При заданном идентификационном номере студента алгоритм последовательного поиска начинает работу с начала списка и последовательно сравнивает каждый выбираемый элемент с искомым числом. Не зная, что представляет собой искомое число, мы не можем определить, насколько далеко потребуется просматривать список. Все же можно утверждать, что для множества выполненных операций поиска их средняя глубина будет равна приблизительно половине длины списка, хотя в одних случаях поиск потребует меньшего числа операций, а в других — большего. Следовательно, можно сделать вывод, что при многократном выполнении последовательного поиска на каждый случай в среднем приходится приблизительно 15 000 просмотренных личных дел. Если выборка каждого личного дела из памяти и сравнение его номера с искомым выполняются за десять миллисекунд (десять тысячных долей секунды), то

среднее время поиска будет составлять 150 секунд или две с половиной минуты. Если секретарю придется так долго ожидать появления на экране монитора личного дела интересующего его студента, несомненно, что этот вариант совершенно неприемлем. Даже если время выборки и проверки каждой записи сократить до одной миллисекунды, на поиск личного дела студента все равно потребуется в среднем около 15 секунд — все еще слишком много для среднего времени ожидания ответа, которое можно считать приемлемым.

В противоположность этому алгоритм двоичного поиска начинает работу со сравнения искомого значения со средним элементом списка. Если это не искомый элемент, то область поиска сразу же сужается до половины исходного списка, т.е. после проверки среднего элемента списка из 30 000 личных дел алгоритм двоичного поиска в большинстве случаев выберет для дальнейшего рассмотрения только 15 000 дел. После второго этапа область поиска в большинстве случаев сократится до 7500 дел, после третьего — до 3750 и т.д. В результате искомое значение будет найдено при выборе максимум 15 элементов списка, состоящего из 30 000 дел. Таким образом, если каждое выбранное значение обрабатывается за 10 миллисекунд, процесс поиска нужного личного дела потребует не более 0,15 секунды, а это означает, что с точки зрения секретаря личное дело любого студента будет появляться на экране практически мгновенно. Можно сделать обоснованное заключение, что выбор между алгоритмом последовательного поиска и алгоритмом двоичного поиска в данном случае имеет большое значение.



Основные положения для запоминания

- **Нахождение эффективного алгоритма решения задачи позволяет успешно решать более сложные ее варианты.**

Этот пример иллюстрирует важность той области компьютерных наук, которую называют анализом алгоритмов. Эта область связана с изучением необходимых алгоритмам ресурсов, таких как время или используемый объем памяти. Основным практическим применением результатов подобных исследований является оценка относительных достоинств альтернативных алгоритмов. Анализ алгоритма включает изучение ситуаций, в которых он демонстрирует свои наилучшие свойства, ситуаций, когда его эффективность минимальна, а также оценку его средней производительности. В нашем случае мы выполнили анализ средней производительности алгоритмов последовательного и двоичного поиска, чтобы оценить, какое время им потребуется для выполнения поиска в списке из 30 000 элементов. В общем случае такой анализ осуществляется

в более широком контексте. Это означает, что при рассмотрении алгоритмов, выполняющих поиск в списке, мы не ограничиваемся списком фиксированной длины, но пытаемся вывести формулу эффективности алгоритма для списков произвольной длины. Не составит большого труда обобщить наши предыдущие рассуждения на случай списка произвольной длины. В частности, при применении к списку из n элементов алгоритму последовательного поиска в среднем потребуется проверить $n/2$ элементов, тогда как алгоритму двоичного поиска в самом худшем случае потребуется проверить только $\log_2 n$ элементов. (В данном случае выражение $\log_2 n$ представляет логарифм числа n по основанию 2, который показывает, сколько раз число n можно разделить на два. Если явно не указывается обратное, в компьютерных науках всегда, когда речь идет об логарифмах, подразумевается логарифм по основанию 2.)



Основные положения для запоминания

- **Определение эффективности алгоритмов осуществляется посредством формального или математического обоснования алгоритма.**
- **Различные правильные алгоритмы решения одной и той же проблемы могут иметь разную эффективность.**

Давайте попробуем проанализировать аналогичным образом алгоритм сортировки методом вставки (см. рис. 5.11). Вспомним, что этот алгоритм предусматривает выбор одного из элементов списка, называемого опорным, с последующим сравнением этого элемента с предшествующими ему до тех пор, пока для него не будет найдена правильная позиция, в которую этот опорный элемент и помещается. Поскольку основным действием в реализации данного алгоритма является сравнение двух элементов, наш подход будет состоять в подсчете количества таких сравнений, которые потребуется выполнить при сортировке списка длиной n элементов.

Алгоритм начинается с выбора второго элемента списка в качестве опорного. По мере его выполнения в качестве опорных выбираются следующие элементы, пока не будет достигнут конец списка. В самом лучшем случае каждый опорный элемент уже находится на положенном ему месте. Следовательно, чтобы это было обнаружено, его потребуется сравнить только с одним именем. Поэтому в наилучшем случае применение алгоритма сортировки методом вставки к списку из n элементов потребует выполнения $n - 1$ сравнений. (Второй элемент сравнивается с одним элементом (первым), третий элемент — с одним элементом (вторым) и т.д.)

И наоборот, наихудший сценарий имеет место в том случае, когда каждый опорный элемент требуется сравнивать со всеми стоящими впереди элементами, прежде чем удастся найти правильное место его расположения. Очевидно, что в этом случае исходный список упорядочен в обратном порядке. Первый опорный элемент (второй элемент списка) сравнивается с одним элементом, второй опорный элемент (третий элемент списка) — с двумя элементами и т.д. (рис. 5.18). Следовательно, общее количество сравнений при сортировке списка из n элементов составит $1 + 2 + 3 + ... + (n - 1)$, что эквивалентно $(1/2)(n^2 - n)$. В частности, для списка из 10 элементов алгоритму сортировки методом вставки в наихудшем случае потребуется выполнить 45 сравнений.



Рис. 5.18. Работа алгоритма сортировки методом вставки в наихудшем случае

В среднем при сортировке методом вставки можно ожидать, что каждый опорный элемент потребуется сравнить с половиной предшествующих ему элементов. В этом случае общее количество выполненных сравнений будет вдвое меньше, чем в наихудшем случае, т.е. $(1/4)(n^2 - n)$ сравнений для списка длиной n. Например, если использовать сортировку методом вставки для упорядочения множества списков из 10 элементов, то среднее число производимых в каждом случае сравнений будет равно 22,5.

Важность полученного выше результата состоит в том, что количество сравнений, выполненных алгоритмом сортировки методом вставки, позволяет оценить время, которое потребуется для выполнения сортировки. Эта оценка была использована для построения графика, представленного на рис. 5.19. Он показывает, как будет возрастать время, необходимое для выполнения сортировки методом вставки, при увеличении длины сортируемого списка. Данный график построен по оценкам работы алгоритма в наихудшем случае, когда, исходя из результатов наших исследований, для списка длиной n элементов потребуется выполнить не менее $(1/2)(n^2 - n)$ сравнений. На графике отмечено несколько конкретных значений длины списка и указано время, необходимое в каждом случае. Обратите внимание: при увеличении длины списка на одно и то же количество элементов время, необходимое для сортировки списка, все больше

и больше возрастает. Таким образом, с увеличением длины списка эффективность данного алгоритма уменьшается.

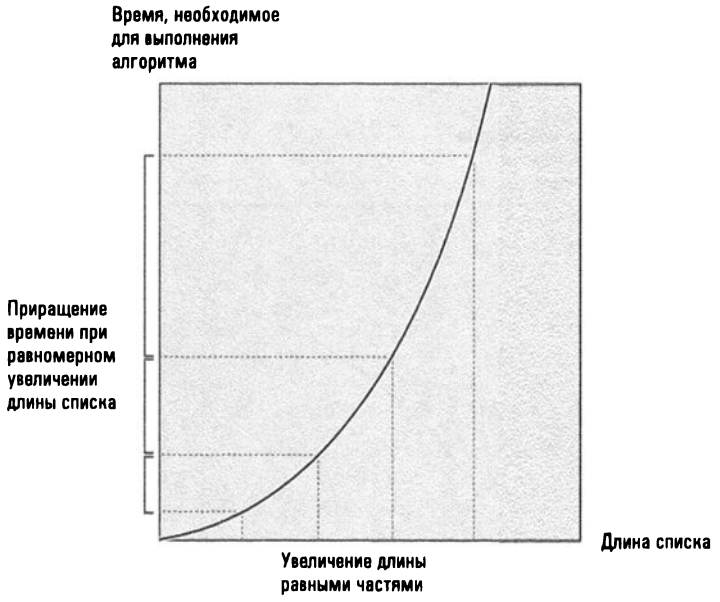


Рис. 5.19. Работа алгоритма сортировки методом вставки в наихудшем случае

А теперь давайте выполним аналогичный анализ для алгоритма двоичного поиска. Как было установлено выше, при использовании этого алгоритма для поиска в списке из n элементов потребуется проанализировать не более $\log_2 n$ элементов. Это позволяет оценить время, необходимое для выполнения алгоритма при различной длине сортируемого списка. На рис. 5.20 представлен график, построенный по результатам данного анализа. На этом графике также отмечены конкретные значения длины списка, возрастающие на одну и ту же величину, и указано соответствующее время выполнения алгоритма. Обратите внимание, что темпы роста времени выполнения алгоритма снижаются по мере увеличения длины списка, т.е. эффективность алгоритма двоичного поиска возрастает с увеличением длины списка.

Основным различием между графиками, представленными на рис. 5.19 и 5.20, безусловно, является их общая форма. Именно общая форма графика, а не его индивидуальные особенности, демонстрирует, насколько хорошо данный алгоритм будет справляться со все возрастающими объемами данных. Заметим, что общая форма графика определяется типом отображаемого математического выражения, а не его конкретными особенностями: все линейные выражения изображаются прямой линией, все квадратичные выражения — параболической

кривой, а все логарифмические выражения порождают логарифмическую кривую, подобную представленной на рис. 5.20. Общую форму кривой принято определять простейшим выражением, порождающим кривую данной формы. В частности, параболическая форма обычно определяется выражением n^2, а логарифмическая — выражением $\log_2 n$.

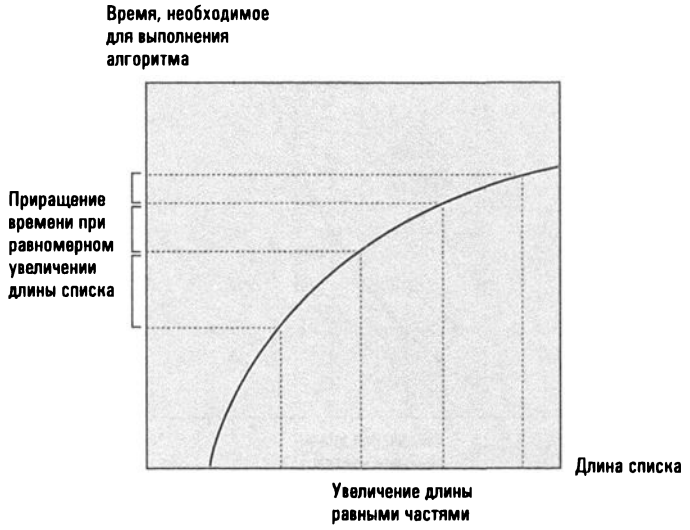


Рис. 5.20. График продолжительности работы алгоритма двоичного поиска для наихудшего случая

Поскольку форма графика, представляющего зависимость времени выполнения алгоритма от объема входных данных, отражает общие характеристики эффективности алгоритма, общим подходом является классифицировать алгоритмы согласно форме их графиков, как правило, построенных для самого неблагоприятного случая. Способ обозначения, используемый для определения этих классов, иногда называют **тета-классами**. Все алгоритмы, графики которых имеют параболическую форму (например, сортировка методом вставки), относятся к классу $\Theta(n^2)$ (читается как "тета эн-квадрат"), а алгоритмы, графики которых имеют логарифмическую форму (например, двоичный поиск), — к классу $\Theta(\log_2 n)$. Зная класс, к которому относится конкретный алгоритм, можно с уверенностью предсказать показатели его производительности и сравнить их с показателями других алгоритмов, позволяющих решить ту же самую задачу. Два алгоритма класса $\Theta(n^2)$ будут демонстрировать сходные изменения в количестве требуемого им времени при возрастании объема входных данных. Более того, можно с уверенностью утверждать, что затраты времени любого алгоритма класса $\Theta(\log_2 n)$ никогда не будут возрастать так же быстро, как в случае алгоритмов класса $\Theta(n^2)$.

Верификация программ

Вспомним, что четвертая фаза процедуры решения задачи согласно схеме, предложенной математиком Полиа (см. раздел 5.3), заключается в оценке точности найденного решения и определении его потенциала как инструмента для решения других задач. Важность первой части этой фазы мы проиллюстрируем следующим примером.

> Путешественник, у которого есть золотая цепочка из семи звеньев, намерен остановиться в уединенном отеле не более чем на семь ночей. Плата за каждую проведенную в отеле ночь составляет одно звено его цепочки. Какое наименьшее число звеньев необходимо разрезать, чтобы путешественник мог платить владельцу отеля одно звено каждое утро, не внося плату заранее?



Основные положения для запоминания

- **Правильность алгоритма определяется путем его формального или математического анализа, а не путем тестирования некоторой реализации этого алгоритма.**

Первым делом уясним, что нет необходимости разрезать все звенья. Если мы разрежем только второе звено, то и первое, и второе будут отделены от остальных пяти звеньев. Следуя этому, можно прийти к решению разрезать только второе, четвертое и шестое звенья цепочки. В результате все звенья окажутся свободными, причем только три из них будут разрезанными (рис. 5.21). Более того, любое меньшее число разрезов оставит два звена соединенными, поэтому мы заключаем, что правильный ответ для этой задачи — три звена.

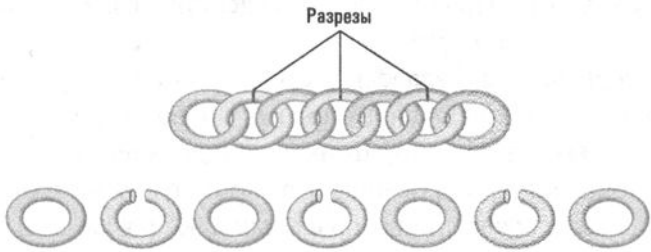


Рис. 5.21. Разъединение всех звеньев цепочки с помощью всего лишь трех разрезов

Однако, рассмотрев задачу более внимательно, можно заметить, что если разрезать только третье звено, то получится три фрагмента цепочки, состоящих

из одного, двух и четырех звеньев (рис. 5.22). С этими фрагментами мы можем поступить следующим образом.

Первое утро. Отдать владельцу отеля одно звено.

Второе утро. Забрать у владельца отеля одно звено и отдать ему фрагмент цепочки из двух звеньев.

Третье утро. Отдать владельцу отеля одно звено.

Четвертое утро. Забрать у владельца отеля три отданные ему ранее звена
и отдать ему фрагмент цепочки из четырех звеньев.

Пятое утро. Отдать владельцу отеля одно звено.

Шестое утро. Забрать у владельца отеля одно звено и отдать ему фрагмент цепочки из двух звеньев.

Седьмое утро. Отдать владельцу отеля одно звено.

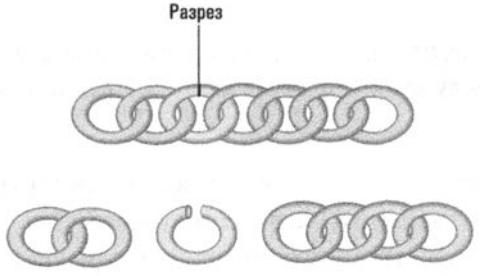


Рис. 5.22. Решение задачи с помощью всего лишь одного разреза

Следовательно, тот ответ, который мы считали правильным, на самом деле неверен. Как же убедиться, что новое решение действительно правильно? В качестве доказательства можно привести следующее рассуждение. Поскольку в первое утро необходимо отдать владельцу отеля одно звено, придется разрезать по крайней мере одно звено цепочки. Но так как в новом варианте решения требуется разрезать только одно звено, это решение должно быть оптимальным.

Применительно к программированию этот пример демонстрирует различия между программой, которая выглядит правильной, и программой, которая действительно является правильной. Это не всегда одно и то же. Специалистам в области обработки данных известно множество ужасных историй о том, как программное обеспечение, которое считалось безусловно правильным, все же отказывало в критический момент, поскольку возникшая ситуация оказывалась для него совершенно непредвиденной. Следовательно, верификация

программного обеспечения — это важное и необходимое дело, а поиск эффективных методов верификации является активным направлением исследований в области компьютерных наук.

Одно из самых современных направлений в этой области заключается в использовании методов формальной логики для доказательства корректности программ. Это означает, что цель проводимых исследований состоит в применении формальной логики для доказательства того факта, что реализованный данной программой алгоритм делает именно то, для чего он предназначен. Основополагающий тезис заключается в том, что при сведении процесса верификации к формальной процедуре мы застрахованы от некорректных умозаключений, вытекающих из интуитивной аргументации, как это было в случае с задачей о золотой цепочке. Давайте рассмотрим данный подход к верификации программ более подробно.

Подобно тому, как формальное математическое доказательство основывается на аксиомах (геометрические доказательства часто базируются на аксиомах Евклидовой геометрии, тогда как доказательства других утверждений — на аксиомах теории множеств), формальное доказательство правильности программы основывается на спецификациях, в соответствии с которыми эта программа разрабатывалась. Чтобы доказать, что программа правильно сортирует списки имен, мы можем начать с предположения о том, что на вход программы подается список имен. Если программа создана для вычисления среднего значения одного или более положительных чисел, мы можем предположить, что исходными данными для программы является одно или несколько положительных чисел. Короче говоря, доказательство корректности начинается с предположения о том, что в начале работы программы удовлетворены некоторые условия, называемые предварительными условиями или **предусловиями**.

Следующий этап доказательства корректности заключается в рассмотрении того, как следствия из этих предусловий распространяются по программе. С этой целью исследователи изучили различные программные структуры, пытаясь установить, как выполнение данной структуры влияет на утверждение, о котором до выполнения этой структуры было известно, что оно истинно. Рассмотрим простой пример: пусть определенное утверждение о значении переменной `Y` перед выполнением приведенной ниже инструкции было истинно.

```
X = Y
```

Тогда после выполнения этой инструкции то же заключение можно сделать о значении переменной `X`. Например, если перед выполнением инструкции значение переменной `Y` отличалось от `0`, то можно сделать заключение, что после выполнения этой инструкции значение переменной `X` также будет отличаться от `0`.

Верификация требуется не только программному обеспечению

Обсуждаемые в тексте проблемы верификации касаются не только программного обеспечения. Столь же важно получить гарантии, что выполняющая программу аппаратура также не содержит ошибок. Это подразумевает верификацию как разрабатываемых схем, так и конструкции всей машины. И в этом случае полученные результаты в значительной степени зависят от тестирования, задача которого, как и в случае с программным обеспечением, — выявить скрытые ошибки. Показателен пример машины Mark I, созданной в Гарвардском университете в 1940 году, монтажные ошибки в которой оставались необнаруженными в течение многих лет. В 1990-х годах были обнаружены ошибки при выполнении операций с плавающей точкой, имевшие место в первых микропроцессорах типа Pentium, — при делении определенных чисел частное оказывалось неверным. В обоих случаях существующие ошибки были выявлены до возникновения каких-либо серьезных последствий.

Современные математические методы, подобные *проверке моделей*, предлагают многообещающий подход к проверке аппаратного обеспечения и определенных типов программного обеспечения, но имеют собственные ограничения по мощности и выразительности.

Несколько более сложный случай являет собой структура `if-else`, например, следующего вида:

```
if (условие):
  инструкция A
else:
  инструкция B
```

Если в этом примере известно, что некоторое утверждение было истинно перед выполнением данной структуры, то непосредственно перед выполнением инструкции `A` мы знаем, что истинны как это утверждение, так и проверяемое условие. В то же время, если должна выполняться инструкция `B`, мы знаем, что должны быть истинны исходное утверждение и отрицание проверяемого условия.

Если следовать правилам, приведенным выше, доказательство корректности программы осуществляется путем определения положений, называемых **утверждениями**, которые устанавливаются в различных точках программы. В результате получается набор утверждений, каждое из которых является следствием предусловий программы и последовательности инструкций, приводящей к той точке программы, в которой установлено данное утверждение. Если утверждение, установленное подобным образом в конце программы, соответствует спецификациям того, что требуется получить на ее выходе, то можно сделать заключение о правильности программы.

В качестве примера рассмотрим типичную циклическую структуру `while`, представленную на рис. 5.23. Предположим, что как следствие предусловий, заданных в точке A, мы можем установить, что определенное утверждение истинно при каждой проверке условия окончания цикла (точка B) на протяжении всего процесса повторения. (Такое утверждение внутри цикла называется **инвариантом цикла**.) Как только повторение завершается, выполнение переходит к точке C, в которой мы можем заключить, что истинны как инвариант цикла, так и условие его окончания. (Инвариант цикла остается истинным, поскольку проверка условия окончания не изменяет никаких величин в программе, а условие окончания истинно, поскольку в противном случае цикл просто не завершился бы.) Если комбинация этих положений означает то, что мы хотим видеть на выходе, наше доказательство корректности можно завершить, просто показав, что компоненты инициализации и модификации цикла в конечном счете приводят к условию окончания.

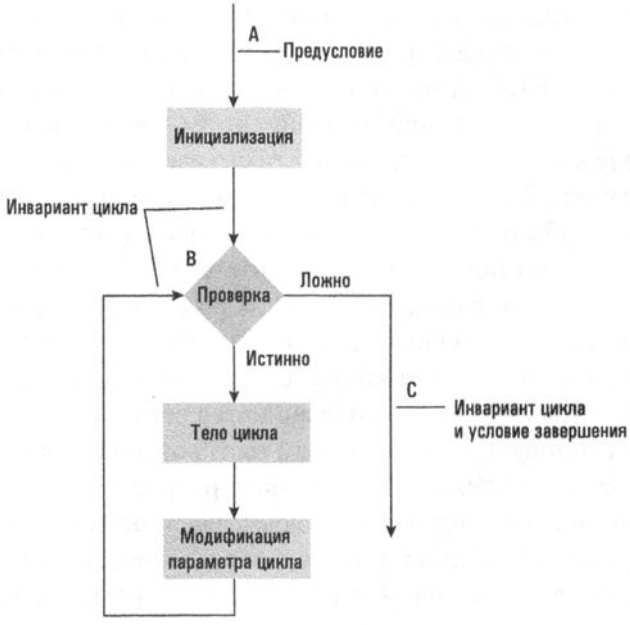


Рис. 5.23. Утверждения, относящиеся к типичной структуре `while`

Этот метод анализа можно применить к приведенному выше алгоритму сортировки методом вставки (см. рис. 5.11). Внешний цикл в этой программе основан на следующем инварианте цикла.

> Каждый раз при выполнении проверки условия окончания цикла элементы списка от первой позиции до позиции $N - 1$ образуют отсортированный список.

Условие окончания этого цикла формулируется следующим образом.

Значение *N* больше, чем длина списка.

Таким образом, как только цикл завершится, мы будем знать, что оба эти условия должны быть выполнены, а это означает, что весь список будет отсортирован.

Достижение прогресса в разработке методов проверки программ по-прежнему связано с существенными трудностями. Тем не менее определенные успехи в этом направлении уже достигнуты. Одно из наиболее значительных достижений можно найти в языке программирования SPARK, который очень близок к более популярному языку Ada. (Язык Ada входит в число тех основных языков программирования, которые будут обсуждаться в следующей главе.) Помимо того что язык SPARK позволяет представлять программы в высокоуровневой форме, подобной нашему псевдокоду, он также предлагает программистам средства включения в программы утверждений, таких как предусловия, постусловия и инварианты циклов. В результате программа, написанная на языке SPARK, содержит не только реализацию требуемого алгоритма, но также информацию, необходимую для применения формальных методов ее верификации. На настоящий момент язык SPARK был успешно использован во многих проектах создания прикладного программного обеспечения, в том числе критически важного с точки зрения надежности и безошибочности его работы. К последнему можно отнести программное обеспечение для Агентства национальной безопасности США, программное обеспечение внутренней управляющей системы самолета C130J Геркулес компании Локхид, а также критической с точки зрения безопасности движения системы управления железнодорожным транспортом.

Несмотря на достигнутые успехи, такие как разработка языка SPARK, методы формальной верификации программ пока не получили широкого распространения, и поэтому на сегодняшний день большинство создаваемого программного обеспечения "верифицируется" посредством тестирования — процесса, дающего в лучшем случае не слишком убедительные результаты. В конечном счете верификация с помощью тестирования не доказывает ничего иного, кроме того, что программа правильно работает в тех условиях, в которых ее проверяли. Любые дополнительные заключения — это всего лишь предположения. Ошибки, содержащиеся в программе, чаще всего являются следствием оплошности и недосмотра, что может произойти и при тестировании. В результате ошибки в программе, такие как в задаче с золотой цепочкой, могут остаться и часто остаются незамеченными, хотя были потрачены значительные усилия, чтобы этого избежать. Драматический пример последствий подобной ошибки имел место в США и Канаде в 2003 году. Ошибка в программном обеспечении, контролирующем работу электростанции, привела к цепному отключению около 100 других электростанций, в результате чего без электричества осталось более 50 миллионов человек в пяти штатах США и канадской провинции Онтарио.

5.6. Вопросы и упражнения

1. Предположим, было установлено, что при использовании алгоритма сортировки методом вставки машине требуется в среднем одна секунда для сортировки списка из 100 элементов. Оцените, сколько времени ей понадобится для сортировки списка из 1000 элементов? А что можно сказать о списке в 10 000 элементов?
2. Приведите примеры алгоритмов, относящихся к каждому из следующих классов: $\Theta(\log_2 n)$, $\Theta(n)$ и $\Theta(n^2)$.
3. Перечислите следующие классы в порядке убывания их эффективности: $\Theta(n^2)$, $\Theta(\log_2 n)$, $\Theta(n)$ и $\Theta(n^3)$.
4. Проанализируйте следующую задачу и предлагаемый ответ. Является ли этот ответ правильным? Поясните свои выводы.

   Задача. Предположим, что в коробке находятся три карточки. У одной из них обе стороны черного цвета, у второй обе стороны красного цвета, а у третьей одна сторона черного цвета, а другая — красного. Из коробки вынимают одну карточку, и вам разрешается посмотреть на одну из ее сторон. Какова вероятность того, что вторая сторона этой карточки того же цвета, что и та, которую вам показали?

   Предлагаемый ответ. Одна вторая. Предположим, что показанная вам сторона карточки была красного цвета. (Рассуждения будут аналогичны и в том случае, если она будет черного цвета, так как задача симметрична.) Из трех карточек только у двух есть красная сторона. Следовательно, карточка, которую вы увидели, — одна из этих двух. У одной из этой пары карточек вторая сторона красная, а у другой — черная. Следовательно, вторая сторона у выбранной из коробки карточки с равной вероятностью может быть как черной, так и красной.
5. Приведенный ниже программный сегмент используется, чтобы вычислить частное (не принимая во внимание остаток) двух целых положительных чисел (делимого и делителя) путем подсчета, сколько раз делитель можно вычесть из делимого, пока оставшаяся часть станет меньше делителя. Например, при делении по этому методу числа 7 на 3 получится 2, так как число 3 можно вычесть из 7 дважды. Правильно ли составлена эта программа? Обоснуйте свои выводы.

```
Счетчик = 0
Остаток = Делимое
repeat:
```

```
    Остаток = Остаток - Делитель
    Счетчик = Счетчик + 1
    until (Остаток < Делитель)
  Частное = Счетчик.
```

6. Приведенный ниже программный сегмент предназначен для определения произведения двух неотрицательных целых чисел X и Y посредством вычисления суммы X копий числа Y. Другими словами, выражение 3 × 4 вычисляется как сумма трех четверок. Правильно ли составлена эта программа? Обоснуйте свой ответ.

```
Произведение = Y
Счетчик = 1
while (Счетчик < X):
  Произведение = Произведение + Y
  Счетчик = Счетчик + 1
```

7. Приняв как предусловие, что значение N — целое положительное число, установите инвариант цикла, который приводит к заключению, что по окончании приведенной ниже программы переменной Сумма будет присвоено значение, равное $0 + 1 + \ldots + N$.

```
Сумма = 0
K = 0
while (K < N):
  K = K + 1
  Сумма = Сумма + K
```

Приведите аргументы в пользу того, что эта программа действительно завершится.

8. Предположим, что программа и выполняющая ее аппаратура были формально проверены на корректность. Гарантирует ли это правильность их работы?

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. Приведите примеры последовательности этапов, удовлетворяющей неформальному определению алгоритма, приведенному во введении к разделу 5.1, но не удовлетворяющей определению, приведенному на рис. 5.1.
2. Объясните различие между неоднозначностью в предложенном алгоритме и неоднозначностью в представлении алгоритма.
3. Поясните, как использование примитивов помогает устранить неоднозначность в представлении алгоритмов.
4. Выберите научный предмет, в котором вы хорошо разбираетесь, и разработайте псевдокод, предназначенный для записи инструкций по выполнению некоторых действий в рамках этого предмета. В частности, опишите примитивы, которые вам было бы удобно использовать, а также синтаксис, который можно было бы применять для их представления. (Если обдумывать положения научных предметов вам затруднительно, попробуйте обратиться к спорту, искусству или ремеслам.)
5. Представляет ли следующая программа алгоритм в строгом смысле этого слова? Поясните свой ответ.

```
Счетчик = 0
while (Счетчик != 5):
   Счетчик = Счетчик + 2
```

6. По какой причине приведенная ниже последовательность из трех этапов не образует алгоритм?

   Этап 1. Провести отрезок прямой линии, соединяющий точки с координатами (2, 5) и (6, 1).

   Этап 2. Провести отрезок прямой линии, соединяющий точки с координатами (1, 3) и (3, 6).

   Этап 3. Провести окружность с радиусом 2 и центром в точке пересечения проведенных отрезков.

7. Перепишите следующий сегмент программы, используя структуру `repeat` вместо `while`. Убедитесь, что новая версия программы печатает те же значения, что и исходная.

```
Счетчик = 2
while (Счетчик < 7):
   print(Счетчик)
   Счетчик = Счетчик + 1
```

8. Перепишите следующий сегмент программы, используя структуру `while` вместо `repeat`. Убедитесь, что новая версия программы печатает те же значения, что и исходная.

```
Счетчик = 1
repeat:
   print(Счетчик)
   Счетчик = Счетчик + 1
   until (Счетчик == 5)
```

9. Что следует предпринять, чтобы преобразовать цикл с постусловием, представленный в форме

```
repeat:(. . .)
  until (. . .)
```

в эквивалентный ему цикл с постусловием, представленный в форме

```
do:
  (. . .)
  while (. . .)
```

10. Разработайте алгоритм, который получает на входе некую конфигурацию из цифр 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 и переставляет полученные цифры таким образом, чтобы новая конфигурация представляла собой значение, следующее по величине за исходным, из числа тех, которые могут быть составлены из этих цифр (или сообщает, что такой перестановки не существует, если никакие перестановки не приводят к большему значению). Например, для исходной конфигурации 5 647 382 901 таким числом будет 5 647 382 910.
11. Разработайте алгоритм определения всех множителей положительного целого числа. Например, в случае целого числа 12 ваш алгоритм должен будет вывести следующие значения: 1, 2, 3, 4, 6 и 12.
12. Разработайте алгоритм определения дня недели для любой даты начиная с 1 января 1700 года. Например, 17 августа 2001 года — пятница.
13. В чем отличие формального языка программирования от псевдокода?
14. В чем различие между синтаксисом и семантикой?
15. Ниже приведен зашифрованный пример сложения двух десятичных чисел, причем каждая буква в этой записи представляет определенную десятичную цифру. Какую именно цифру представляет каждая буква в этой записи? С чего бы вы начали решение этой задачи?

```
   XYZ
+  YWY
  ZYZW
```

16. Ниже приведен зашифрованный пример умножения двух десятичных чисел, причем каждая буква в этой записи представляет определенную десятичную цифру. Какую именно цифру представляет каждая буква в этой записи? С чего бы вы начали решение этой задачи?

```
  XY
× YX
  XY
 YZ
 WVY
```

17. Ниже приведен зашифрованный пример сложения двух двоичных чисел, причем каждая буква в этой записи представляет определенную двоичную цифру. Какая буква в этой записи представляет 1, а какая представляет 0? Разработайте алгоритм для решения задач, подобных этой.

```
  YXX
+ XYX
 XYYY
```

18. Четыре шахтера, у которых есть лишь один фонарь, должны пройти через шахту. Одновременно по шахте могут двигаться не больше двух человек, и каждый шахтер, двигаясь в шахте, должен идти обязательно при свете. Шахтеры, имена которых — Эндрю, Блэйк, Джонсон и Келли, могут пройти шахту за одну, две, четыре и восемь минут соответственно. Когда два шахтера идут вместе, они движутся со скоростью более медленного из них. Каким образом шахтеры могут пройти через шахту за 15 минут? После того как вы решите задачу, объясните, с чего вы начали решение.

19. Допустим, у нас есть большой и маленький стаканчики для вина. Сначала наполним вином маленький стаканчик и перельем его в большой стакан. Затем наполним маленький стакан водой, перельем некоторое количество воды в большой стакан и смешаем его с вином. Теперь будем переливать смесь обратно в маленький стакан, пока он не наполнится. Чего теперь больше: воды в большом стакане или вина в маленьком? После того как вы решите задачу, объясните ход ваших рассуждений.

20. Две пчелы, Ромео и Джульетта, живут в разных ульях, но они встретились и полюбили друг друга. Однажды безветренным весенним утром они одновременно вылетели из своих ульев, чтобы слетать друг к другу в гости. В 50-ти метрах от ближайшего улья они встретились, но не заметили друг друга и полетели дальше. Прибыв к месту своего назначения, они потратили одинаковое время, чтобы выяснить, что того, к кому они прилетели, нет дома, и повернуть назад. На обратном пути они встретились в точке, находящейся на расстоянии 20 метров от ближайшего улья. На этот раз они увидели друг друга и устроили пикник, прежде чем возвратиться

домой. На каком расстоянии друг от друга расположены их ульи? Решив задачу, объясните, с чего вы начали свои рассуждения.

21. Разработайте алгоритм, который получает на вход две строки символов и проверяет, является ли первая строка частью второй.

22. Следующий алгоритм разработан для того, чтобы напечатать несколько первых чисел Фибоначчи. Определите, что является телом цикла. Где выполняется операция инициализации управления циклом? Где выполняется операция модификации? Какая инструкция реализует операцию проверки? Какой список чисел получится в результате работы алгоритма?

```
ПредыдущееЗначение = 0
ТекущееЗначение = 1
while (ТекущееЗначение < 100):
   print(ТекущееЗначение)
   Буфер = ПредыдущееЗначение
   ПредыдущееЗначение = ТекущееЗначение
   ТекущееЗначение = ПредыдущееЗначение + Буфер
```

23. Какую последовательность чисел напечатает следующий алгоритм, если на входе задать значения 0 и 1?

```
def ВыводНаПечать (ПредыдущееЗначение, ТекущееЗначение):
   if (ТекущееЗначение < 100):
      print(ТекущееЗначение)
      Буфер = ТекущееЗначение + ПоследнееЗначение
      ВыводНаПечать(ТекущееЗначение, Буфер)
```

24. Преобразуйте функцию `ВыводНаПечать` из предыдущего задания так, чтобы она печатала числа в обратном порядке.

25. Какие буквы будут проверяться, если применить двоичный поиск (см. рис. 5.14) для поиска значения J в списке A, B, C, D, E, F, G, H, I, J, K, L, M, N, O? А в случае поиска значения Z?

26. Сколько раз в среднем потребуется сравнивать между собой два элемента при поиске значения в списке из 6000 элементов с помощью метода последовательного поиска? А что можно сказать о методе двоичного поиска?

27. Сформулируйте условие окончания для каждого из приведенных ниже примеров итеративной инструкции.

а.
```
while (Счетчик < 5):
    . . .
```

б.
```
repeat:
    . . .
    until (Счетчик == 1)
```

в.
```
while ((Счетчик < 5) and (Итог < 56)):
    . . .
```

28. Определите тело цикла в следующей структуре и подсчитайте, сколько раз оно будет выполнено. Что произойдет, если проверяемое условие заменить выражением "(Счетчик != 6)"?

```
Счетчик = 1
while (Счетчик != 7):
   print(счетчик)
   Счетчик = Счетчик + 3
```

29. Какие проблемы могут возникнуть при реализации на компьютере следующей программы? (*Подсказка*. Вспомните об ошибках округления при выполнении арифметических операций с плавающей точкой.)

```
Счетчик = ОднаДесятая
repeat:
   print(Счетчик)
   Счетчик = Счетчик + ОднаДесятая
   until (Счетчик == 1)
```

30. Разработайте рекурсивную версию алгоритма Евклида (вопрос 3 к разделу 5.2).

31. Предположим, что на вход функций `Проверка1` и `Проверка2`, приведенных ниже, передано значение 1. Чем будут различаться напечатанные этими функциями результаты?

```
def Проверка1 (Счетчик):
   if (Счетчик != 5):
      print(Счетчик)
      Проверка1(Счетчик + 1)

def Проверка2 (Счетчик):
   if (Счетчик != 5):
      Проверка2(Счетчик + 1)
      print(Счетчик)
```

32. Определите основные составляющие механизма управления в предыдущем задании. В частности, какое условие вызывает окончание процесса? Где происходит модификация состояния процесса, приближающая его к условию завершения? Где инициализируется состояние управляющего процесса?

33. Сформулируйте условия окончания для следующей рекурсивной функции.

```
def XXX (N):
   if (N == 5):
      XXX(N + 1)
```

34. Пусть функция ВыводНаПечать (приведенная ниже) была вызвана с параметром N, равным 3. Укажите последовательность чисел, которые эта функция выведет на печать.

```
def ВыводНаПечать (N):
   if (N > 0):
      print(N)
      ВыводНаПечать(N - 2)
   print(N + 1)
```

35. Пусть функция ВыводНаПечать (приведенная ниже) была вызвана с параметром N, равным 2. Укажите последовательность чисел, которые эта функция выведет на печать.

```
def ВыводНаПечать (N):
   if (N > 0):
      print(N)
      ВыводНаПечать(N - 2)
   else:
      print(N)
      if (N > -1):
         ВыводНаПечать(N + 1)
```

36. Разработайте алгоритм генерации последовательности целых положительных чисел (в порядке возрастания), которые имеют только два простых множителя — 2 и 3, т.е. программа должна генерировать последовательность чисел 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27... Представляет ли эта программа алгоритм в строгом смысле этого слова?

37. Ответьте на следующие вопросы применительно к списку имен Alice, Byron, Carol, Duane, Elaine, Floyd, Gene, Henry, Iris.

а. Какой алгоритм поиска (последовательный или двоичный) позволит быстрее найти имя Gene?

б. Какой алгоритм поиска (последовательный или двоичный) позволит быстрее найти имя Alice?

в. Какой алгоритм поиска (последовательный или двоичный) позволит быстрее обнаружить отсутствие в списке имени Bruce?

г. Какой алгоритм поиска (последовательный или двоичный) позволит быстрее обнаружить отсутствие в списке имени Sue?

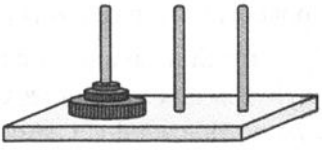
д. Сколько элементов будет рассмотрено при поиске имени Elaine с использованием метода последовательного поиска? Сколько элементов будет рассмотрено при поиске этого имени с использованием метода двоичного поиска?

38. По определению факториал числа 0 равен 1. Факториал целого положительного числа — это произведение данного целого положительного числа и факториала предшествующего ему целого положительного числа. Для обозначения факториала целого положительного числа n используется нотация $n!$. Таким образом, факториал числа 3 (обозначается как 3!) — это $3 \times (2!) = 3 \times (2 \times (1!)) = 3 \times (2 \times (1 \times (0!))) = 3 \times (2 \times (1 \times (1))) = 6$. Разработайте рекурсивный алгоритм вычисления факториала заданного числа.

39. **а.** Предположим, что вам необходимо отсортировать список из пяти имен и что вы уже разработали раньше алгоритм для сортировки списка из четырех имен. Разработайте алгоритм сортировки списка из пяти имен, использующий ранее разработанный алгоритм.

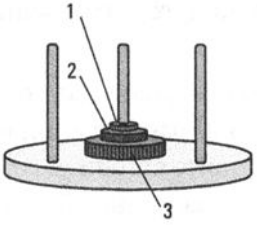
б. Разработайте рекурсивный алгоритм сортировки списка произвольной длины, основанный на методе, используемом для решения предыдущего задания (*а*).

40. Головоломка "Ханойская башня" состоит из трех вертикальных стержней, на один из которых надето несколько колец последовательно уменьшающихся (в направлении снизу вверх) диаметров. Задача состоит в том, чтобы переместить набор колец на другой стержень. Разрешается перемещать только одно кольцо за один ход, причем нельзя надевать большее кольцо поверх меньшего. Заметим, что головоломка с одним кольцом решается предельно просто. Если колец несколько, то, переместив на другой стержень все кольца, кроме наибольшего, наибольшее кольцо можно было бы перенести на третий стержень. После этого задача была бы сведена к перемещению всех остальных колец на наибольшее. Исходя из этого разработайте рекурсивный алгоритм для решения головоломки "Ханойская башня" с произвольным числом колец.



41. Еще один подход к решению головоломки "Ханойская башня" (задание 40) состоит в следующем. Представьте себе, что стержни расставлены по кругу в положениях, соответствующих отметкам 4, 8 и 12 часов на циферблате. Кольца, находящиеся исходно на одном из стержней, нумеруются числами 1, 2, 3 и так далее начиная с верхнего. Кольца с нечетными номерами, находящиеся вверху набора, разрешается перемещать только на стержень, следующий по часовой стрелке, кольца с четными

номерами можно перемещать только против часовой стрелки (при условии, что такое перемещение не приведет к помещению большего кольца над меньшим). Учитывая вышеизложенные требования, вы всегда должны выбирать кольцо с наибольшим номером из числа тех, которые доступны для перемещения. Используя такой подход, разработайте нерекурсивный алгоритм решения головоломки "Ханойская башня".



42. Разработайте циклический и рекурсивный алгоритмы для распечатки дневной заработной платы рабочего, который в каждый последующий день получает вдвое больше, чем в предыдущий (первый платеж равен одному пенни), за 30 дней работы. С какими проблемами, касающимися хранения данных, вам придется столкнуться при реализации вашего решения на реальной машине?

43. Разработайте алгоритм для нахождения значения квадратного корня из положительного числа с помощью следующего метода. В качестве первого приближения выбирается само это число, а последующие приближения получаются из предыдущих путем вычисления среднего арифметического для предыдущего приближения и числа, полученного при делении исходного числа на предыдущее приближение. Проанализируйте возможности управления этим повторяющимся процессом. В частности, какое условие должно использоваться для его окончания?

44. Разработайте алгоритм, который печатает все возможные варианты перестановки символов в строке из пяти различных символов.

45. Разработайте алгоритм, который в заданном списке имен находит самое длинное имя. (Воспользуйтесь для этой цели циклом `for`.) Определите, как поведет себя алгоритм, предложенный вами в качестве решения, если "самых длинных" имен в списке будет несколько. В частности, как поведет себя ваш алгоритм, если все имена в списке будут одной длины?

46. Разработайте алгоритм, который в заданном списке из пяти или более чисел находит пять наименьших и пять наибольших чисел, не сортируя полностью весь список.

47. Расположите имена Brenda, Doris, Raymond, Steve, Timothy и William в таком порядке, который при использовании алгоритма сортировки методом вставки потребует выполнения наименьшего числа сравнений (рис. 5.11).

48. Какое максимальное количество элементов может быть проверено при применении алгоритма двоичного поиска (рис. 5.14) к списку, содержащему 4000 имен? Как оно соотносится с аналогичным значением для метода последовательного поиска (рис. 5.6)?

49. Используя нотацию тета-классов, классифицируйте традиционные школьные алгоритмы для сложения и умножения в столбик. Другими словами, определите, сколько отдельных операций сложения необходимо выполнить при суммировании двух чисел значностью n цифр и сколько отдельных операций умножения потребуется для их перемножения.

50. Иногда небольшое изменение условия задачи может вызвать существенные изменения в способе ее решения. Например, найдите простой алгоритм решения следующей задачи и определите его тета-класс.

Разделите группу людей на две непересекающиеся подгруппы (произвольных размеров), такие, чтобы разность между суммами возрастов членов этих подгрупп была максимальной.

Теперь измените условие задачи так, чтобы требуемая разность была минимальной, и вновь классифицируйте ваше решение.

51. Из следующего списка выделите несколько чисел, сумма которых равна 3165. Насколько эффективен ваш метод решения задачи?

26, 39, 104, 195, 403, 504, 793, 995, 1156, 1677

52. Завершится ли цикл в следующей программе? Поясните свой ответ. Объясните, что могло бы случиться, если бы эта программа в действительности выполнялась машиной (см. раздел 1.7).

```
X = 1
Y = 1 / 2
while (X !=0):
  X = X - Y
  Y = Y / 2
```

53. Следующий фрагмент программы разработан для вычисления произведения двух неотрицательных целых чисел X и Y путем вычисления суммы X копий числа Y. Иначе говоря, выражение 3×4 вычисляется посредством нахождения суммы трех четверок. Правильно ли составлен данный фрагмент? Поясните свой ответ.

```
Произведение = 0
Счетчик = 0
repeat:
  Произведение = Произведение + Y
  Счетчик = Счетчик + 1
  until (Счетчик == X)
```

54. Следующий фрагмент программы составлен для определения, какое из двух целых чисел, *X* и *Y*, является бо́льшим. Является ли этот фрагмент правильным? Поясните свой ответ.

```
Разность = X - Y
if (разность больше нуля):
    print('X больше Y')
else:
    print('Y больше X')
```

55. Следующий фрагмент программы должен находить наибольший элемент в непустом списке целых чисел. Правильно ли он составлен? Поясните свой ответ.

```
ПроверяемоеЗначение = первый элемент списка
ТекущееЗначение = первый элемент списка
while (текущий элемент не является последним элементом):
  if (ТекущееЗначение > ПроверяемоеЗначение):
    ПроверяемоеЗначение = ТекущееЗначение
  ТекущееЗначение = следующий элемент в списке
```

56. а. Определите предусловия для алгоритма последовательного поиска, представленного на рис. 5.6. Установите инвариант цикла для структуры `while` в этой программе, который, будучи объединен с условием окончания цикла, предполагает, что по окончании этого цикла алгоритм правильно сообщит об успехе или неудаче.

б. Приведите аргументы в пользу того, что цикл `while` на рис. 5.6 действительно завершается.

57. Опираясь на предусловие для приведенной ниже программы, утверждающее, что параметрам *X* и *Y* присвоены неотрицательные целые значения, определите инвариант цикла ее структуры `while` , который, будучи объединен с указанным условием окончания, предполагает, что значение переменной *Z* по завершении цикла будет *X* – *Y*.

```
Z = X
J = 0
while (J < Y):
  Z = Z - 1
  J = J + 1
```

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также разобраться, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. Поскольку сегодня невозможно осуществить всеобъемлющую верификацию сложных программ, при каких обстоятельствах (если таковые вообще существуют) создатель программы должен нести ответственность за имеющиеся в ней ошибки?
2. Предположим, у вас есть идея, которую вы воплощаете в некоем продукте, пригодном для использования множеством людей. Более того, чтобы ваша идея обрела форму, в которой она становится полезной большинству людей, потребовалось потратить год работы и 50 тыс. долларов. Однако большинство людей могут использовать продукт в его окончательном виде без того, чтобы приобретать у вас что-нибудь. Есть ли у вас право потребовать компенсацию? Является ли этичным использование пиратских копий программ?
3. Предположим, что пакет программ стоит настолько дорого, что вам он совершенно не по карману. Этично ли скопировать его для личного пользования? (В конце концов, вы же не лишаете поставщика программы заработка, поскольку вы бы ее все равно не купили.)
4. Право собственности на реки, леса, океаны и тому подобное всегда было предметом ожесточенных дебатов. В каком смысле правомочно говорить о передаче какому-то лицу или организации права собственности на алгоритм?
5. Одни полагают, что новые алгоритмы открываются, тогда как другие считают, что они создаются. А как думаете вы? Может ли это привести к различным заключениям в отношении владения алгоритмами и прав собственности на алгоритмы?
6. Этично ли разработать алгоритм для незаконного действия? Имеет ли значение, был ли этот алгоритм когда-либо действительно применен на практике? Следует ли предоставить данному лицу право собственности на этот алгоритм? Если да, то какими правами должно быть наделено

данное лицо? Следует ли предоставление права собственности на алгоритм поставить в зависимость от назначения данного алгоритма? Является ли этичным рекламировать и распространять методы взлома систем защиты? Имеет ли значение, что именно подвергается взлому?

7. Писателю платят за право экранизации его произведения, хотя очень часто при этом в сюжет вносятся те или иные изменения. До какой степени допустимо изменять сюжет произведения, чтобы новая версия не превратилась в другое произведение? Какие изменения должны быть внесены в алгоритм, чтобы он превратился в другой алгоритм?
8. В настоящее время на рынке имеется образовательное программное обеспечение для детей в возрасте 18 месяцев и даже младше. Сторонники этих продуктов заявляют, что данное программное обеспечение воспроизводит изображения и звуки, которые многие дети иначе никогда не смогли бы увидеть или услышать. Противники же утверждают, что подобные программы являют собой очень слабую замену личному общению ребенка с его родителями. Каково ваше мнение по этому вопросу? Можете ли вы выбрать одну из указанных точек зрения, не имея никаких дополнительных сведений об этих программных продуктах? Если да, то как поступите?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Aho A.V., Hopcroft J.E., Ullman J.D. *The Design and Analysis of Computer Algorithms*. — Boston, MA: Addison-Wesley, 1974. (Имеется русский перевод этой книги: Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. *Разработка и анализ компьютерных алгоритмов*: — М.: Издательство "Диалектика", 2020.)
2. Baase S. *Computer Algorithms: Introduction to Design and Analysis*, 3rd ed. — Boston, MA: Addison-Wesley, 2000.
3. Barnes J. *High Integrity Software: The SPARK Approach to Safety and Security*. — Boston, MA: Addison-Wesley, 2003.
4. Gries D. *The Science of Programming*. — New York: Springer-Verlag, 1998.
5. Harbin R. *Origami — the Art of Paper Folding*. — London: Hodder Paperbacks, 1973.
6. Johnsonbaugh R., Schaefer M. *Algorithms*. — Upper Saddle River, NJ: PrenticeHall, 2004.
7. Kleinberg J. *Algorithm Design*, 2nd ed. — Boston, MA: Addision-Wesley, 2014.

8. Knuth D.E. *The Art of Computer Programming*, Vol. 3, 2nd ed. — Boston, MA: Addison-Wesley, 1998. (Имеется русский перевод этой книги: Кнут Д.Э. *Искусство программирования. Т.3. Сортировка и поиск*, 2-е изд.: — М.: Издательский дом "Вильямс", 2000.)
9. Levitin A.V. *Introduction to the Design and Analysis of Algorithms*, 3rd ed. — Boston, MA: Addison-Wesley, 2011.
10. Polya G. *How to Solve It.* — Princeton, NJ: Princeton University Press, 1973.
11. Roberts E.S. *Thinking Recursively.* — New York: Wiley, 1986.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Ахо А., Хопкрофт Дж., Ульман Дж. *Структуры данных и алгоритмы.* — М.: Издательский дом "Вильямс", 2000.
2. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. *Алгоритмы: построение и анализ*, 3-е изд. — М.: Издательский дом "Вильямс", 2013.
3. Кормен Т.Х. Алгоритмы: вводный курс. — М.: Издательский дом "Вильямс", 2014.
4. Мюллер Дж.П., Массарон Л. *Алгоритмы для чайников.* — СпБ. : ООО "Альфа-книга", 2018.

Глава 6

В этой главе речь пойдет о языках программирования. Наша цель состоит вовсе не в том, чтобы сосредоточиться на определенном языке и подробно рассмотреть все его особенности, хотя мы будем продолжать использовать примеры, представленные на языке Python, везде, где это потребуется. Вместо этого в данной главе вам предлагается познакомиться прежде всего с самой концепцией языков программирования и их общими характеристиками, а также разобраться в том, чем в действительности определяется столь большое разнообразие как самих существующих языков программирования, так и связанных с ними методологий.

Языки программирования

6.1. ИСТОРИЧЕСКИЙ ОБЗОР
- Ранние поколения
- Машинная независимость: что далее?
- Парадигмы программирования

6.2. КОНЦЕПЦИИ ТРАДИЦИОННОГО ПРОГРАММИРОВАНИЯ
- Переменные и типы данных
- Структура данных
- Константы и литералы
- Операторы присваивания
- Управляющие операторы
- Комментарии

6.3. ПРОЦЕДУРНЫЕ ЭЛЕМЕНТЫ ПРОГРАММ
- Функции
- Параметры
- Функции, возвращающие значение

6.4. РЕАЛИЗАЦИЯ ЯЗЫКА
- Процесс трансляции
- Пакеты для разработки программ

6.5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ
- Классы и объекты
- Конструкторы
- Дополнительные характеристики

*6.6. ПРОГРАММИРОВАНИЕ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ

*6.7. ДЕКЛАРАТИВНОЕ ПРОГРАММИРОВАНИЕ
- Логический вывод
- Язык Prolog

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

Разработка сложных систем программного обеспечения, например операционных систем, сетевого программного обеспечения или же широчайшего диапазона прикладных пакетов, существующих на сегодняшний день, была бы практически невозможна, если бы люди вынуждены были выражать требуемые алгоритмы на машинном языке. Было бы, мягко говоря, очень неудобно работать с огромным количеством запутанных деталей, связанных с использованием таких языков, а любая попытка построить сколько-нибудь сложную систему превращалась бы как минимум в тяжелое испытание.

В результате было разработано множество языков программирования, позволяющих выражать алгоритмы в форме, доступной для людей и в то же время достаточно удобной для преобразования в инструкции машинного кода. В этой главе мы подробно познакомимся с областью компьютерных наук, охватывающей разработку и использование языков программирования.

6.1. Исторический обзор

Давайте начнем изучение этой темы с того, что проследим общую историю развития языков программирования.

Ранние поколения

Как уже говорилось в главе 2, программы для современных компьютеров представляют собой последовательность инструкций, закодированных в виде цифровых последовательностей. Такую систему кодирования называют машинным языком. К сожалению, написание программ на машинном языке — задача крайне утомительная, что часто приводило к ошибкам, которые требовалось выявлять и исправлять (процесс, называемый **отладкой**), прежде чем работу можно было считать законченной.

В 1940-х годах первым шагом на пути к упрощению задачи программирования был отказ от использования цифр для записи команд и операндов непосредственно в той форме, в которой они используются в машине. С этой целью были разработаны системы записи программ, в которых применялась мнемоническая запись машинных команд вместо их шестнадцатеричного представления. Например, инструкцию

```
Переслать содержимое регистра 5 в регистр 6
```

с использованием машинного языка, представленного в главе 2, следовало бы записать как

```
0x4056
```

С использованием же мнемонической системы записи эта же инструкция будет выглядеть как

```
MOV R5, R6
```

Вот более развернутый пример, представляющий собой запись на машинном языке программы, приведенной в конце раздела 2.2. Эта программа суммирует содержимое ячеек с адресами 0x6C и 0x6D, а затем помещает результат в ячейку с адресом 0x6E (см. рис. 2.7 в главе 2). На машинном языке соответствующая последовательность команд имеет вид

```
0x156C
0x166D
0x5056
0x306E
0xC000
```

Воспользовавшись мнемонической записью, ту же самую программу можно переписать следующим образом.

```
LD R5, Price
LD R6, ShippingCharge
ADDI R0, R5, R6
ST R0, TotalCost
HLT
```

Здесь мнемонические обозначения LD, ADDI, ST и HLT использованы для представления кодов команд *загрузить*, *сложить*, *сохранить* и *стоп* соответственно. Кроме того, описательные имена Price (*цена*), ShippingCharge (*стоимость доставки*) и TotalCost (*общая стоимость*) используются здесь для ссылок на ячейки памяти с адресами 0x6C, 0x6D и 0x6E соответственно. Такие описательные имена обычно называют программными переменными или **идентификаторами**. Обратите внимание: мнемоническое представление, оставаясь достаточно лаконичным, в то же время позволяет гораздо лучше представить суть выполняемых в программе действий, чем при ее записи в сугубо числовом представлении.

Как только мнемонические системы получили достаточное распространение, были разработаны программы, названные **ассемблерами** и предназначенные для перевода записанных в мнемоническом виде программ на машинный язык. Таким образом, вместо того, чтобы разрабатывать программы непосредственно на машинном языке, программисты получили возможность писать программы в мнемонической форме, а затем переводить их на машинный язык с помощью программ-ассемблеров.

Мнемоническая система записи программ стала рассматриваться как язык программирования, получивший название **язык ассемблера**. Разработка

языков ассемблера стала гигантским шагом вперед в поисках более совершенных технологий программирования. Фактически появление языков ассемблера было настолько революционным явлением, что их стали называть языками программирования второго поколения, тогда как к первому поколению были отнесены сами машинные языки.

Хотя языки второго поколения имели много преимуществ по сравнению с машинными языками, они все же не могли обеспечить завершенную среду программирования. Помимо всего прочего, применяемые в языке ассемблера языковые конструкции, по существу, совпадают с конструкциями соответствующих машинных языков. Разница заключается лишь в синтаксическом способе их выражения. По этой причине программы, написанные на языке ассемблера, являются принципиально машинно-зависимыми, т.е. команды в этих программах выражаются в терминах определенных машинных атрибутов. Программу на языке ассемблера достаточно сложно перенести и выполнить на машине другого типа, поскольку для этого ее нужно переписать с учетом новой конфигурации регистров и набора команд.

Еще одним недостатком языков ассемблера является тот факт, что, хотя программист и не обязан больше кодировать программу с помощью нулей и единиц, он все еще вынужден мыслить в терминах пошагового выполнения команд машинного языка. Это аналогично проектированию дома из досок, гвоздей, кирпичей, балок и других материалов. Конечно, реальная конструкция дома состоит именно из этих элементарных единиц, но проектировать его все же легче, имея дело с более крупными составляющими: комнатами, окнами, дверьми и прочими подобными конструктивными компонентами.

Короче говоря, элементарные примитивы, из которых в конечном счете должен быть сконструирован продукт, вовсе не обязательно должны использоваться и при разработке проекта этого продукта. При проектировании удобнее пользоваться примитивами более высокого уровня, каждый из которых представляет концепцию, связанную с некоторой функцией конечного продукта достаточно высокого уровня. По окончании проектирования эти примитивы могут быть выражены с помощью концепций более низкого уровня, относящихся к деталям их реализации.

Следуя такому подходу, специалисты по компьютерам стали разрабатывать языки программирования, которые больше подходили для целей разработки программного обеспечения, чем низкоуровневые языки ассемблера. В результате появились языки программирования третьего поколения, которые отличались от предыдущих поколений тем, что их языковые конструкции имели более высокий уровень (а значит, выражали инструкции с большим приращением) и были **машинно-независимыми** (в том смысле, что они не опирались на характерные особенности определенной машины). Наиболее известными

примерами ранних языков третьего поколения являются FORTRAN (FORmula TRANslator — переводчик формул), который был предназначен для научных и инженерных расчетов, и COBOL (COmmon Business-Oriented Language — язык общего назначения деловой ориентации), разработанный специалистами военно-морского флота США для реализации бизнес-приложений.

В общем случае язык программирования третьего поколения представляет собой определенный набор языковых конструкций достаточно высокого уровня, предназначенный для разработки программного обеспечения. (По сути, точно так же был разработан и наш псевдокод, описанный в главе 5.) Каждая из языковых конструкций была разработана так, чтобы ее можно было реализовать в виде последовательности низкоуровневых примитивов, существующих в машинных языках. Рассмотрим следующий оператор:

```
assign TotalCost the value Price + ShippingCharge
```

Он представляет собой выражение высокого уровня, в котором совершенно отсутствуют указания, как именно определенная машина должна выполнять поставленную задачу. Однако этот оператор вполне можно реализовать в виде последовательности машинных команд, которые мы обсуждали выше. Следовательно, приведенная ниже структура псевдокода потенциально также является языковой конструкцией высокого уровня.

```
идентификатор = выражение
```

После того как необходимый набор примитивов высокого уровня будет определен, пишется программа, называемая **транслятором** (*translator* — переводчик). Она предназначена для перевода программ, записанных с использованием примитивов языка высокого уровня, на машинный язык. Подобный транслятор похож на программу-ассемблер второго поколения, за исключением того, что ему часто приходится объединять (или компилировать, от англ. *compile*) несколько машинных инструкций в короткие последовательности команд, предназначенные для имитации выполнения отдельных примитивов высокого уровня. Именно поэтому подобные программы-переводчики часто называют **компиляторами**.

Распространенной альтернативой трансляторам являются **интерпретаторы**, предложенные как еще одно средство выполнения программ, написанных на языках программирования третьего поколения. Эти программы подобны трансляторам, однако они выполняют команды программы непосредственно после их перевода, а не записывают, подобно трансляторам, переведенный код в виде выполняемого модуля, предназначенного для последующего использования. Это означает, что вместо создания копии программы на машинном языке, которую необходимо будет выполнить позже, интерпретатор просто немедленно выполняет все переведенные им инструкции.

В качестве интересного замечания следует отметить, что процесс признания и распространения языков третьего поколения оказался не настолько простым, как это можно было бы себе представить. Мысль о возможности написания программ в форме, близкой к естественным языкам, оказалась настолько революционной, что многие на руководящих должностях поначалу активно противостояли этому. Грейс Хоппер, которую считают разработчиком первого компилятора, часто рассказывала историю о том, что однажды ей пришлось демонстрировать работу транслятора с языка третьего поколения, в котором служебные слова были взяты из немецкого языка вместо английского. Суть здесь в том, что этот язык программирования был построен на основе небольшого набора примитивов, которые могли быть выражены на многих естественных языках, причем переход от одного языка к другому требовал внесения в программу-транслятор лишь небольшого количества очень простых модификаций. Однако к своему удивлению она вдруг обнаружила, что многие из присутствующих на демонстрации были шокированы тем, что она якобы учит компьютер "понимать" немецкий всего через несколько лет после окончания второй мировой войны. Сегодня мы уже знаем, что задача понимания естественного языка в действительности на много порядков сложнее, чем распознание нескольких четко определенных примитивов. В действительности **естественные языки** (такие, как английский, немецкий, латинский или русский) сильно отличаются от **формальных языков** (таких, как языки программирования) в том, что последним присуща четко определенная грамматика (см. раздел 6.4), тогда как первые на протяжении долгого времени развивались без какого-либо формального грамматического анализа.

Кроссплатформенное программное обеспечение

Типичная прикладная программа при решении многих задач вынуждена полагаться на операционную систему. Например, она может обратиться к диспетчеру окна для организации взаимодействия с пользователем или к диспетчеру файлов для получения данных с устройств массовой памяти. К сожалению, различные операционные системы выполняют такие запросы по-разному. Поэтому если программа предназначена для рассылки и выполнения в сети, объединяющей машины разного типа, которые имеют различные операционные системы, то она должна быть независимой как от операционных систем, так и от типа используемых машин. Для обозначения такого уровня независимости используется термин "кроссплатформенное программное обеспечение". Иными словами, кроссплатформенное программное обеспечение — это программы, которые не зависят ни от операционной системы, ни от аппаратного обеспечения, а значит, могут выполняться на разных компьютерах, объединенных в сеть.

Машинная независимость: что далее?

С появлением языков программирования третьего поколения цель обеспечения машинной независимости программ была в основном достигнута. Поскольку операторы в языках третьего поколения не привязаны к особенностям какой-то конкретной машины, они легко могут быть скомпилированы на любом компьютере. Теоретически программа, написанная на языке третьего поколения, может быть выполнена на любой машине при условии использования соответствующего компилятора.

В действительности не все так просто. При разработке самого компилятора приходится учитывать определенные ограничения, накладываемые той машиной, для которой он предназначен, что иногда отражается в виде наложения определенных условий на язык программирования, который транслируется на машинный язык. Например, различные способы выполнения операций ввода-вывода на машинах разных типов исторически приводил к тому, что на разных машинах один и тот же язык программирования приобретал некоторые особенности, образуя, так сказать, его диалекты. Вследствие этого программистам часто приходится выполнять как минимум легкую модификацию программы при переносе ее с одной машины на другую.

Усугубляет проблему переносимости программ и тот факт, что часто даже отсутствует общая точка зрения на то, что именно считать стандартом данного языка программирования. В связи с этим Американский национальный институт стандартов (ANSI) и Международная организация по стандартизации (ISO) приняли и опубликовали стандарты для многих популярных языков программирования. В других случаях применяются неформальные стандарты, которые являются следствием популярности того или иного диалекта языка, а также желания многих разработчиков компиляторов создавать продукты, совместимые с другими, подобными им. Тем не менее, даже в случае высокостандартизованных языков, разработчики компиляторов часто реализуют в своих продуктах такие функции (обычно называемые *расширениями* языка), которые не являются частью его стандартной версии. Если прикладной программист воспользуется преимуществами этих функций, созданная им программа может оказаться несовместимой со средой разработки, обеспечиваемой компиляторами от других разработчиков.

В общей истории языков программирования тот факт, что языки третьего поколения не достигли истинной машинной независимости, на самом деле не имеет большого значения по двум причинам. Во-первых, они все же являются достаточно машинно-независимыми, для того чтобы можно было относительно легко переносить программное обеспечение с одной машины на другую. Во-вторых, машинная независимость, как оказалось, — это лишь промежуточная ступень на пути к достижению более важных целей.

В каждую следующую волну новых языков программирования более высокого уровня встраивались все более и более мощные абстракции, что повышало уровень, на котором программисты могли строить концепции разрабатываемого ими программного обеспечения, и позволяло создавать все более и более сложные программные системы. Несколько строк кода языка программирования высокого уровня могут использовать миллионы строк библиотечного кода, вызывать встроенные функции операционной системы или инициировать сложные взаимодействия с сетями и базами данных. В результате языки программирования более высокого уровня делают компьютерные программы более простыми с точки зрения их написания авторами-программистами и одновременно более легкими для прочтения и понимания другими людьми.

Эти достижения, в свою очередь, привели к возникновению среди ученых в области компьютерных наук заманчивой мечты о создании среды программирования, которая позволила бы людям общаться с машиной в терминах абстрактных понятий, а не заставляла их переводить эти понятия в машинно-совместимую форму. Более того, ученым понадобились машины, способные самостоятельно выявлять и строить алгоритмы, а не просто выполнять действия, описанные с помощью набора инструкций. В результате спектр языков программирования заметно расширился, что в конечном счете привело к усложнению их прежней классификации в терминах простого разделения на поколения.



Основные положения для запоминания

- **Языки программирования высокого уровня предоставляют в распоряжение программистов абстракции более высокого уровня, что существенно упрощает задачи написания и чтения программ человеком.**

Парадигмы программирования

Классификация языков программирования по поколениям требует распределения их по линейной шкале (рис. 6.1) в соответствии с той степенью свободы от компьютерной тарабарщины, которую данный язык предоставляет программисту, что позволяет ему мыслить понятиями, связанными непосредственно с решаемой задачей. В действительности развитие языков программирования происходило несколько иначе. Оно протекало по разным направлениям, связанным с появлением и развитием альтернативных подходов к процессу программирования (называемых **парадигмами программирования**). В результате историческую схему развития языков программирования наиболее точно можно изобразить составной диаграммой, показанной на рис. 6.2, на которой

отдельные линии, символизирующие различные парадигмы программирования, появляются и развиваются независимо друг от друга. В частности, на рисунке показаны четыре независимых направления, соответствующие функциональной, объектно-ориентированной, императивной и декларативной парадигмам программирования, а также представлены различные относящиеся к ним языки. Месторасположение названия языка на линии соответствует времени его появления относительно других языков. Однако это вовсе не означает, что каждый последующий язык обязательно является наследником предыдущего.



Рис. 6.1. Схематическое представление поколений языков программирования

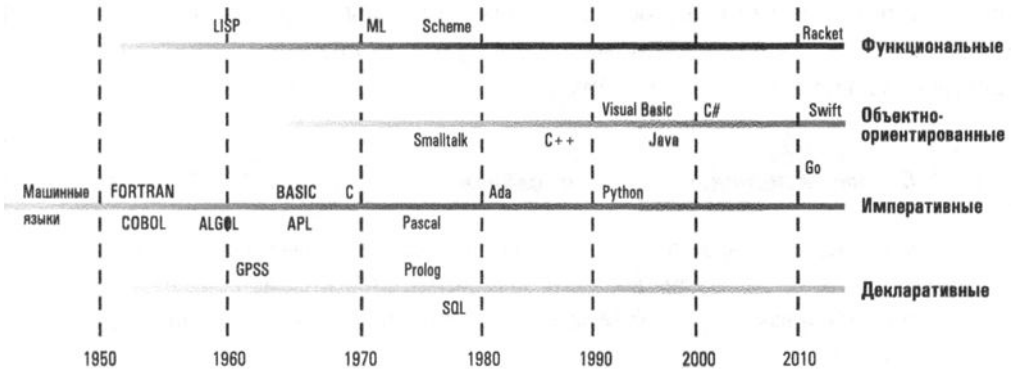


Рис. 6.2. Эволюция парадигм программирования

Следует отметить, что хотя парадигмы, представленные на рис. 6.2, называются парадигмами *программирования*, появление этих альтернативных подходов имело последствия, далеко выходящие за рамки собственно процесса программирования. В действительности они представляют принципиально разные *подходы* к построению решений поставленных задач и, следовательно, оказывают глубокое влияние на весь процесс разработки программного обеспечения. В этом смысле термин *парадигма программирования* является неправильным. Более реалистичным термином будет *парадигма разработки программного обеспечения*.

В рамках каждой парадигмы может существовать много разных языков, предлагающих разные уровни абстракции, от языков очень низкого уровня до очень высокого. Как мы уже видели, естественные языки и псевдокод позволяют адекватно выражать алгоритмы на вполне приемлемом уровне, тогда как языки программирования с текстовой или визуальной средой разработки предназначены для явного описания алгоритмов с точностью, достаточной для их автоматического выполнения процессором компьютера. Бурное развитие языков программирования высокого уровня обусловлено несколькими факторами. В обширной области поисков способов решения различных задач, стоящих перед людьми, было разработано множество языков программирования специального назначения, предназначенных для обеспечения лучшей поддержки определенных типов алгоритмов. Хотя почти все языки программирования позволяют выразить практически любой алгоритм, тем не менее язык, предназначенный для поддержки проектирования графических пользовательских интерфейсов, обязательно будет предлагать иные алгоритмические сокращения в сравнении с языком, разработанным для проведения статистического анализа, или языком, предназначенным для поддержки встроенных сенсорных систем с обратной связью. Выбор языка программирования с правильной парадигмой, наиболее точно соответствующей конкретным особенностям поставленной задачи — или особому стилю решения задач команды разработчиков, — может существенно повлиять на ясность, а также удобство использования и сопровождения создаваемого программного пакета.



Основные положения для запоминания

- **Различные языки программирования предлагают различные уровни абстракции.**
- **К языкам, позволяющим выражать алгоритмы, относятся естественные языки, псевдокод и языки программирования с текстовой или визуальной средой разработки.**
- **Разные языки лучше подходят для выражения различных алгоритмов.**
- **Некоторые языки программирования разработаны для конкретных областей применения и позволяют наилучшим образом выражать алгоритмы, характерные именно для этой области.**
- **Язык, выбранный для выражения алгоритма, может повлиять на такие его характеристики, как ясность или читабельность, но ни в коем случае не определяет саму возможность существования алгоритмического решения.**
- **Почти все языки программирования являются эквивалентными в том смысле, что позволяют выразить любой алгоритм.**
- **Ясность и читабельность — это важные условия, которые обязательно должны быть приняты во внимание при выражении алгоритма на том или ином языке.**

Императивная, или, как ее иначе называют, **процедурная**, парадигма представляет традиционный подход к процессу программирования. Это та парадигма, на которой основан язык Python и наш псевдокод, представленный в главе 5 (так же как и машинный язык, обсуждавшийся в главе 2). Как следует из названия, императивная парадигма определяет процесс программирования как запись последовательности команд, которая при выполнении осуществит обработку данных, необходимую для получения желаемого результата. Иначе говоря, императивная парадигма предлагает такой подход процессу программирования, при котором сначала ищется алгоритм решения поставленной задачи, а затем найденный алгоритм представляется в виде соответствующей последовательности команд.

В противоположность императивной парадигме **декларативная парадигма** предлагает программисту описать *саму* поставленную задачу, вместо того чтобы предлагать алгоритм, по которому ее следует решать. Если говорить точнее, то система декларативного программирования использует предустановленный алгоритм решения задач общего назначения для нахождения решения конкретных поставленных задач. В такой среде задачей программиста становится точная формулировка заданной проблемы, а не поиск и описание алгоритма ее решения.

Основной трудностью в разработке декларативных языков программирования является выбор встроенного базового алгоритма решения задач. По этой причине ранние декларативные языки были узкоспециализированными по самой своей природе и четко ориентированными на определенные приложения. Например, декларативный подход уже многие годы применяется для моделирования систем (экономических, физических, политических и т.п.) в целях проверки выдвинутых гипотез или получения прогнозов. В этом случае базовый алгоритм, в сущности, является процессом моделирования течения времени посредством многократно повторяющегося вычисления значений параметров (роста внутреннего продукта, торгового дефицита и т.д.) исходя из вычисленных ранее значений. В конечном счете разработка декларативного языка для выполнения такого моделирования сводится, прежде всего, к реализации алгоритма, выполняющего указанную повторяющуюся процедуру. В результате единственной задачей программиста является лишь описание моделируемой ситуации. Так, синоптик не должен разрабатывать алгоритм для прогнозирования погоды, он просто описывает текущее ее состояние, что позволяет основному алгоритму моделирования просчитать прогноз погоды на ближайшее время.

Мощный толчок развитию декларативной парадигмы дало осознание того факта, что применение методов математической формальной логики позволяет создавать простые алгоритмы решения задач, подходящие для использования в системах декларативного программирования общего назначения. Результатом пристального внимания ученых к декларативной парадигме явилось появление дисциплины **логического программирования**, которое будет обсуждаться в разделе 6.7.

В еще одной парадигме программирования, в **функциональной парадигме**, любая программа рассматривается как совокупность неких "черных ящиков", каждый из которых получает определенные исходные данные (на входе) и вырабатывает соответствующий результат (на выходе). Математики называют такие "ящики" *функциями*, поэтому этот подход и называется функциональной парадигмой. Таким образом, согласно этой парадигме программа строится посредством соединения меньших по размеру уже существующих программных элементов (предопределенных функций) таким образом, что выходные данные каждого элемента становятся входными данными для другого, а сами эти элементы образуют структуру, позволяющую получить требуемые результаты из исходных данных. Короче говоря, в соответствии с функциональной парадигмой процесс программирования заключается в конструировании требуемых функций в виде вложенных друг в друга совокупностей более простых функций.

В качестве примера на рис. 6.3 показано, как можно построить функцию подведения баланса по счету чековой книжки из двух более простых функций. Первая из них, с названием `FindSum`, получает на вход несколько числовых значений и выдает их сумму в качестве выходного значения. Вторая функция, с названием `FindDif`, принимает на вход два числовых значения и вычисляет их разность, передавая этот результат на выход. Структуру, показанную на рис. 6.3, можно представить на языке LISP (популярном функциональном языке программирования), в котором эта конструкция может быть записана в виде следующего выражения:

```
(FindDiff (FindSum ПрежнийБаланс Поступления) (FindSum Расходы))
```

Использование в этом выражении вложенных структур (на что указывают скобки) отражает тот факт, что исходные данные для функции `FindDiff` являются результатами двух вызовов функции `FindSum`. При первом вызове функция `FindSum` возвращает сумму значений переменной `ПрежнийБаланс` и всех элементов массива `Поступления`, а при втором вызове функция `FindSum` вычисляет сумму всех элементов массива `Расходы`. И наконец, функция `FindDiff` используется для вычисления нового баланса по счету чековой книжки.

Чтобы лучше понять различие между функциональной и императивной парадигмами, сравните представленную выше программу для балансирования чековой книжки (функциональная парадигма) с приведенной ниже программой на псевдокоде, составленной согласно положениям императивной парадигмы.

```
ВсегоПоступлений = сумма всех членов массива Поступления
ПромежуточныйБаланс = ПрежнийБаланс + ВсегоПоступлений
ВсегоРасходов = сумма всех членов массива Расходы
НовыйБаланс = ПромежуточныйБаланс - ВсегоРасходов
```

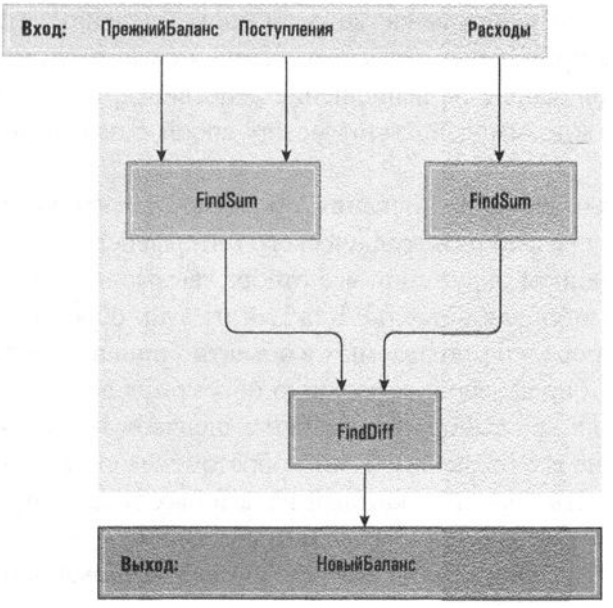


Рис. 6.3. Функция определения баланса по счету чековой книжки, построенная из более простых функций

Обратите внимание, что программа, написанная в соответствии с императивной парадигмой, состоит из нескольких операторов, каждый из которых требует выполнения вычисления и сохранения результата для последующего его использования. Напротив, программа, написанная в соответствии с функциональной парадигмой, состоит всего из одного оператора, в котором результат каждого вычисления немедленно передается в следующее. В некотором смысле императивная программа аналогична группе заводов, каждый из которых преобразует свое сырье в конечный продукт, который хранится на складе. С этих складов продукция затем отправляется на другие заводы — по мере необходимости. Если обратиться к функциональной программе, то она аналогична некоей совокупности предприятий, работа которых скоординирована таким образом, что каждое производит только те продукты, которые заказаны другими заводами, а затем немедленно отправляет эти продукты по назначению без промежуточного хранения. Эта эффективность является одним из преимуществ, провозглашенных сторонниками функциональной парадигмы.

Совершенно иная парадигма программирования (и наиболее востребованная в современной индустрии разработки программного обеспечения) — это **объектно-ориентированная парадигма**, которая предполагает применение методов **объектно-ориентированного программирования** (ООП). В рамках этого подхода программная система рассматривается как набор элементов,

называемых **объектами**, каждый из которых способен выполнять действия, которые могут быть применены непосредственно к самому этому объекту либо представлять собой запрос на выполнение действий другими объектами. Все вместе эти объекты взаимодействуют между собой с целью решения поставленной задачи.

В качестве примера использования объектно-ориентированного подхода рассмотрим задачу разработки графического интерфейса пользователя. В объектно-ориентированном окружении все отображаемые на экране графические элементы реализуются как объекты. Каждый из этих объектов включает собственный набор процедур (называемых в объектно-ориентированной терминологии **методами**), определяющих реакцию объекта на возникновение различных событий, таких как выбор этого объекта щелчком на нем кнопкой мыши или перетаскивание его по экрану. Таким образом, вся система в целом выглядит как совокупность объектов, каждый из которых знает, как реагировать на определенное событие, так или иначе с ним связанное.

Чтобы оценить различия между объектно-ориентированной и императивной парадигмами, рассмотрим конкретный пример — список имен. В традиционной императивной парадигме этот список рассматривается просто как совокупность некоторых данных. Любая программа, получающая на вход этот список, должна содержать алгоритм выполнения над ним требуемых действий. Однако в объектно-ориентированной парадигме список рассматривается как объект, содержащий некоторую совокупность данных (имен) вместе с набором процедур для их обработки. (Этот набор может включать процедуры для вставки в список нового элемента, удаления элемента из списка, сортировки списка или определения, является ли список пустым.) Поэтому программная единица, получающая доступ к списку для его обработки, не обязана содержать алгоритм для выполнения указанных действий. При необходимости она просто выполняет соответствующие процедуры, предоставляемые самим объектом. В этом смысле объектно-ориентированная программа вместо сортировки списка (как при императивной парадигме) скорее просит список отсортировать самого себя.

Хотя объектно-ориентированная парадигма более подробно будет обсуждаться в разделе 6.5, ее огромное значение в области разработки современного программного обеспечения требует, чтобы в это введение было включено обсуждение концепции *класса*. Прежде всего напомним, что объект может состоять из данных (таких, как список имен) и набора методов, предназначенных для выполнения определенных действий (таких, как вставка новых имен в список). Все свойства и функции объекта должны быть описаны соответствующими

операторами в текстовом представлении программы. В объектно-ориентированной терминологии подобное описание свойств объекта называется **классом**. После того как класс создан, его можно использовать в любой момент, как только программе потребуется объект с такими характеристиками. В результате при выполнении программы в ней могут быть созданы сразу несколько объектов с использованием одного и того же описания класса. Как и идентичные внешне близнецы, эти объекты будут разными сущностями, но будут иметь одинаковые характеристики, поскольку они созданы по одному и тому же шаблону (с использованием одного класса). Согласно объектно-ориентированной терминологии объект, созданный на базе определенного класса, называется **экземпляром** этого класса.

Именно по той причине, что объекты являются четко определенными единицами, описания которых изолированы в повторно используемых классах, объектно-ориентированная парадигма приобрела столь большую популярность. Действительно, сторонники объектно-ориентированного программирования утверждают, что объектно-ориентированная парадигма обеспечивает естественную среду для конструирования программного обеспечения из "строительных блоков". Они предсказывают появление все новых и новых программных библиотек, содержащих определения различных объектов, с помощью которых создаваемое программное обеспечение можно будет собирать точно так же, как обычные промышленные изделия собирают из готовых компонентов. Создание и расширение таких библиотек — это непрерывный процесс, о чем вы узнаете в главе 7.

В заключение следует заметить, что методы в рамках объектов, в сущности, представляют собой небольшие программные компоненты, построенные в соответствии с требованиями императивной парадигмы. А это означает, что большинство объектно-ориентированных языков программирования включают в себя множество функциональных элементов, свойственных императивным языкам. Например, популярный объектно-ориентированный язык C++ был создан посредством добавления объектно-ориентированных функциональных возможностей к императивному языку программирования C. Более того, поскольку языки Java и C# являются производными от языка C++, они также унаследовали его императивное ядро. В разделах 6.2 и 6.3 вы познакомитесь со многими из этих императивных свойств и при этом речь фактически будет идти о концепциях, пронизывающих подавляющее большинство современного объектно-ориентированного программного обеспечения. Позднее, в разделе 6.5, будут рассмотрены функциональные возможности и характеристики, являющиеся уникальными для объектно-ориентированной парадигмы.

6.1. Вопросы и упражнения

1. В каком смысле программа на языке третьего поколения является машинно-независимой? В каком смысле она остается машинно-зависимой?
2. Какая разница между ассемблером и компилятором?
3. Императивную парадигму программирования можно кратко охарактеризовать, просто сказав, что она делает акцент на описании процесса, который ведет к решению поставленной задачи. Дайте аналогичное краткое описание декларативной, функциональной и объектно-ориентированной парадигм программирования.
4. В каком смысле языки программирования третьего поколения являются языками более высокого уровня, чем языки предыдущих поколений?

6.2. Концепции традиционного программирования

Языки программирования высокого уровня включают абстракции разных уровней, начиная от простейших (констант, литералов и переменных) через умеренно сложные (операторы, выражения и управляющие структуры) до понятных лишь посвященным (процедурные единицы, модули и библиотеки).

В этом разделе мы рассмотрим некоторые основные концепции, положенные в основу императивных и объектно-ориентированных языков программирования. Для этого рассмотрим примеры программ на языках Ada, C, C++, C#, FORTRAN и Java. Наша цель не в том, чтобы углубиться в дебри какого-либо конкретного языка, а в том, чтобы просто продемонстрировать, как общие языковые свойства и характеристики появляются во множестве совершенно разных реальных языков программирования. Именно из этих соображений и была выбрана представленная выше подборка языков программирования, — в целом они представляют весь существующий ландшафт.

Язык C — это императивный язык программирования третьего поколения. Язык C++ — это объектно-ориентированный язык, который был разработан как расширение языка C. Языки Java и C# — это объектно-ориентированные языки, производные от C++. (Язык Java был разработан компанией Sun Microsystems, которая позднее была приобретена компанией Oracle, тогда как язык C# является продуктом корпорации Microsoft.) Языки FORTRAN и Ada

изначально были разработаны как императивные языки третьего поколения, но их последние версии были расширены для охвата большинства концепций объектно-ориентированной парадигмы. В приложении Г представлены краткие описания каждого из этих языков.

Хотя в предстоящее обсуждение среди прочих включены и объектно-ориентированные языки, такие как C++, Java и C#, в этом разделе всегда используется такой подход, как если бы мы писали программу в соответствии с императивной парадигмой. Это возможно по той причине, что многие элементы в объектно-ориентированных программах (такие, как функции, определяющие, как объект должен реагировать на внешние события) в сущности представляют собой небольшие императивные программы. Позднее, в разделе 6.5, наше обсуждение будет сосредоточено уже только на тех аспектах, которые действительно являются уникальными для объектно-ориентированной парадигмы.

В общем случае императивная программа состоит из набора операторов, которые можно разделить на три категории: операторы объявления, выполняемые операторы и комментарии. **Операторы объявления** вводят пользовательскую терминологию, которая в дальнейшем будет использоваться в программе, например имена для обозначения отдельных элементов данных. **Выполняемые операторы** описывают отдельные этапы используемого алгоритма, а **комментарии** просто повышают читабельность программы, давая пояснения ее специфическим особенностям в более удобной для пользователя форме. Чаще всего императивная программа (или императивная программная единица в рамках объектно-ориентированной программы) может восприниматься как имеющая структуру, представленную на рис. 6.4. Она начинается с набора операторов объявления, описывающих данные, которые будут обрабатываться программой. За этим предварительным материалом следуют выполняемые операторы, образующие в своей совокупности тот алгоритм, который необходимо выполнить. В настоящее время многие языки программирования позволяют смешивать операторы объявления и выполнимые операторы, расставляя их в свободном порядке, но концептуальное различие между ними при этом сохраняется. Операторы комментариев размещаются в программе там, где это необходимо в целях предоставления разъяснений в отношении работы программы.



Основные положения для запоминания

- **Программное обеспечение разрабатывается с использованием многих уровней абстракции, таких как константы, выражения, операторы, процедуры и библиотеки.**

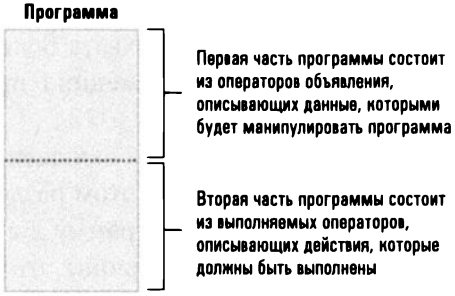


Рис. 6.4. Общая структура типичной императивной программы или программной единицы

Мы продолжим изучение концепций программирования, следуя приведенному выше примеру и рассматривая категории операторов в том порядке, в котором их можно встретить в программе, т.е. начиная с концепций, связанных с операторами объявлений.

Переменные и типы данных

Как вы уже знаете из раздела 1.8, языки программирования высокого уровня позволяют ссылаться на определенные участки в основной памяти с помощью описательных имен, используемых вместо адресов, представленных в виде числовых значений. Такие идентификаторы обычно называют **переменными**, подчеркивая тот факт, что изменяя значение, размещенное в данном участке памяти, мы изменяем значение, связанное с идентификатором, присвоенным этому участку по ходу выполнения программы. В отличие от языка Python, ко всем примерам, приводимым в этой главе, мы предъявим требование, по которому переменные обязательно должны быть объявлены с помощью соответствующего оператора объявления *до того*, как они будут использованы где-либо в программе. К этим операторам объявления также предъявляется дополнительное требование, согласно которому в них обязательно должно присутствовать явное указание типа данных, сохраняемых в том участке памяти, который связан с этой переменной.



Основные положения для запоминания

- **Инструкция в программе может включать переменные, которые инициализируются и обновляются, считываются и записываются.**

Языки сценариев

Одно из подмножеств императивных языков программирования представляет собой группу языков, называемых **языками сценариев** или скриптовыми языками. Эти языки чаще всего используются для выполнения служебных операций и решения административных задач, а не для создания сложных программ. Программный или управляющий код для решения подобных задач обычно называют **сценарием** (или скриптом, от англ. *script*), что поясняет происхождение термина "язык сценариев". Например, администратор компьютерной системы может написать сценарий, предназначенный для выполнения последовательности операций резервного копирования и сохранения, которая должна выполняться каждый вечер, или же пользователь ПК может подготовить сценарий, обеспечивающий выполнение последовательности программ, необходимых для считывания изображений из цифровой фотокамеры, индексирования файлов с этими изображениями по дате и сохранения их копий на устройстве внешней памяти, предназначенном для ведения архива. Свое происхождение языки сценариев ведут от языков управления заданиями, появившихся в 1960-х годах и предназначавшихся для управления операционной системой с целью организации пакетной обработки в многопользовательских системах (см. раздел. 3.1). Даже в наши дни многие рассматривают языки сценариев лишь как средство для управления выполнением других программ, что не совсем соответствует действительности, если принять во внимание все возможности современных языков сценариев. Примерами таких современных языков сценариев являются Perl и PHP, которые широко используются для создания управляющих веб-приложений, функционирующих на стороне сервера (см. раздел 4.3), а также язык VBScript, представляющий собой диалект языка Visual Basic, разработанного компанией Microsoft и широко используемый в среде ОС Windows.

Указываемый при объявлении переменной **тип данных** определяет как интерпретацию конкретных данных, так и операции, которые можно с ними выполнять. Например, тип данных **`integer`** используется для обозначения числовых данных, являющихся целыми числами, которые в памяти чаще всего представляются с помощью двоичной нотации с дополнением. С данными типа `integer` можно выполнять обычные арифметические операции и операции сравнения их значений с целью определения, является ли одно значение больше другого. Тип **`real`** предназначен для представления числовых данных, которые могут содержать нецелые величины и сохраняются в памяти чаще всего как двоичные числа в нотации с плавающей точкой. Операции, которые можно выполнять с данными типа `real`, аналогичны операциям, выполняемым с данными типа `integer`. Однако заметим, что манипуляции, которые следует выполнить, чтобы сложить два элемента данных типа `real`, отличаются от

манипуляций, необходимых для выполнения аналогичных действий с переменными типа `integer`.

Предположим, что нам требуется использовать в программе переменную `ОграничениеВеса` для ссылок на область основной памяти, содержащую числовое значение в двоичном дополнительном коде. В языках C, C++, Java и C# наши намерения можно описать с помощью следующего оператора объявления:

```
int ОграничениеВеса;
```

Этот оператор должен находиться в начале программы. Фактически этот оператор означает "Имя `ОграничениеВеса` будет позднее использовано в программе для ссылок на область основной памяти, содержащую значение, представленное в двоичном дополнительном коде". Несколько переменных одного и того же типа обычно могут быть объявлены в одном и том же операторе объявления. Вот пример такого оператора:

```
int Высота, Ширина;
```

Здесь объявляется присвоение имен `Высота` и `Ширина` двум переменным типа `integer`. Более того, большинство языков программирования позволяют сразу же назначить объявляемым переменным требуемое исходное значение. Выглядит это обычно так:

```
int ОграничениеВеса = 100;
```

Как видите, здесь переменная `ОграничениеВеса` не только объявляется как имеющая тип `integer`, — одновременно ей присваивается начальное значение, равное `100`. В противоположность этому подходу языки с динамической типизацией, такие как Python, позволяют выполнять первое объявление переменных без указания их типа. Такие переменные проверяются на корректность их типа позднее, когда над ними выполняются операции. В языках с выведением типа, таких как Swift, вообще нет необходимости указывать тип переменной при ее объявлении и инициализации, за исключением тех случаев, когда тип исходного значения является потенциально неоднозначным.

К остальным распространенным типам данных относятся `character` и `Boolean`. Тип **`character`** применяется к данным, состоящим исключительно из символов, чаще всего сохраняемых в кодировках ASCII или Unicode. Операции, которые могут быть выполнены над данными такого типа, включают сравнение с целью определения, расположен ли один символ перед другим в алфавитном порядке, проверку, является ли одна строка символов частью другой строки, и конкатенацию — добавление одной строки символов в конец другой строки с образованием новой длинной строки. Приведенный ниже оператор объявления может быть использован в языках C, C++, C# и Java:

```
char Буква, Цифра;
```

В этом операторе объявляются две переменные с именами Буква и Цифра и каждая из них имеет тип character.

Тип **Boolean** применяется к таким данным, которые могут принимать только одно из двух допустимых значений: true (*истина*) и false (*ложь*). Операции, которые можно выполнить над данными типа Boolean, включают проверку, является ли текущее значение переменной true или false. Например, если переменная ЛимитПревышен была объявлена с типом Boolean, то она может быть корректно использована в операторе следующего вида:

```
if (ЛимитПревышен) then (...) else (...)
```

Те типы данных, которые включены в язык программирования в качестве примитивов, таких как int для типа integer и char для типа character, называют **примитивными типами данных** (или *встроенными* либо *базовыми*). Как мы уже знаем, типы данных integer, float, character и Boolean являются общепринятыми примитивами. Другие типы данных, которым пока не соответствуют какие-либо общепринятые элементарные конструкции в основных языках программирования, — это изображения, аудио- или видеоданные и гипертекст. Тем не менее такие типы, как форматы GIF, JPEG и HTML, могут в скором будущем стать общепринятыми типами данных, такими как integer или float. Далее в этой книге (разделы 6.5 и 8.4) вы познакомитесь с тем, как объектно-ориентированная парадигма предоставляет программистам возможность расширить диапазон доступных им типов данных за пределы примитивных типов, встроенных непосредственно в язык. В действительности такая возможность является очень важной и значимой особенностью объектно-ориентированной парадигмы.

В завершение приведем фрагмент кода, который может использоваться в программах на языке C и его производных, языках C++, C# и Java, для объявления переменных Длина и Ширина типа float, переменных Цена, Налог и Всего типа integer и переменной Символ типа character.

```
float Длина, Ширина;
int Цена, Налог, Всего;
char Символ;
```

В разделе 6.4 будет показано, как транслятор использует сведения о типах данных, полученные им из операторов объявления, при переводе программы с языка высокого уровня на машинный. Заметим, что эту информацию можно использовать и для обнаружения ошибок. Например, если транслятор встретит оператор, в котором предпринимается попытка сложить значения двух переменных, которые ранее были объявлены с типом Boolean, он, безусловно, расценит этот оператор как содержащий ошибку и выдаст пользователю сообщение об этом факте.

Структура данных

Помимо типа данных, переменные в программах обычно ассоциируются со **структурой данных**, определяющей общую форму организации данных. Например, текст обычно рассматривается как длинная строка символов, тогда как информация о продажах может рассматриваться как прямоугольная таблица с числовыми значениями, в которой каждая отдельная строка представляет сделки, заключенные определенным работником, а каждый столбец представляет информацию о сделках, заключенных в определенный день.

Одной из общих структур данных является однородный **массив**, который представляет собой блок значений одного типа, например одномерный линейный список, двухмерную таблицу из строк и столбцов или таблицу более высокой размерности. Для объявления такого массива в программе в большинстве языков программирования используются специальные операторы объявления, помимо имени массива содержащие указания о длине каждой его размерности. В качестве примера на рис. 6.5 представлена концептуальная структура, объявляемая следующим оператором:

```
int Оценки[2][9];
```

В языке C этот оператор имеет следующий смысл: "В программе переменная `Оценки` будет использоваться как двухмерный массив целых чисел, состоящий из двух строк и девяти столбцов". Тот же самый по смыслу оператор в языке FORTRAN будет выглядеть так:

```
INTEGER Оценки(2,9)
```

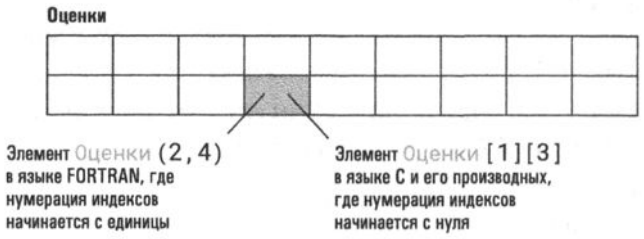


Рис. 6.5. Двухмерный массив с двумя строками и девятью столбцами

Объявив однородный массив, мы можем ссылаться на него по имени в любом месте программы, а доступ к отдельным элементам массива можно получить с помощью **индексов**, указывающих номер нужной строки, столбца и т.д. Однако диапазон изменения этих индексов меняется от языка к языку. Например, в программе на языке C (и производных от него языках C++, Java и C#) нумерация индексов начинается с нуля, а это означает, что на элемент, находящийся на пересечении второй строки и четвертого столбца объявленного выше массива `Оценки`, нужно будет ссылаться как на `Оценки[1][3]`, а элементом на

пересечении первой строки и первого столбца будет `Оценки[0][0]`. В противоположность этому в языке FORTRAN отсчет индексов начинается с единицы, поэтому на тот же элемент во второй строке и четвертом столбце нужно будет ссылаться уже как на `Оценки(2,4)` (см. рис. 6.5).

В противоположность однородному массиву, в котором все элементы имеют один и тот же тип, **неоднородный массив** (иначе называемый **структурой** или **записью**) представляет собой блок данных, в котором отдельные элементы могут иметь разные типы. Например, блок данных, относящийся к некоторому работнику, может содержать элемент `Имя` типа `character`, элемент `Возраст` типа `integer`, а также элемент `Квалификация` типа `real`. Такой тип данных, представляющий собой структуру `Работник`, в языке C можно объявить с помощью следующего оператора:

```
struct { char  Имя[25];
         int   Возраст;
         float Квалификация;
       } Работник;
```

Эта запись означает, что переменная `Работник` предназначена для ссылок на структуру (сокращение — `struct`), состоящую из трех элементов с именами `Имя` (строковое значение длиной 25 символов), `Возраст` и `Квалификация`, как показано на рис. 6.6). Как только такой неоднородный массив будет объявлен, программист может использовать имя структуры (`Работник`) для ссылок на весь неоднородный массив либо для ссылок на отдельные **поля** в этой структуре посредством добавления его имени к имени структуры через точку (например, `Работник.Возраст`).



Рис. 6.6. Концептуальный макет построения структуры `Работник`

В главе 8 вы увидите, как концептуальные структуры, подобные массивам, реализуются в компьютерах. В частности, будет показано, что относящиеся к массиву данные могут быть распределены по большой области основной памяти или внешнего запоминающего устройства. Вот почему мы рассматриваем структуры данных лишь как *концептуальные* формы их представления.

В действительности же реальная форма представления данных в запоминающих устройствах машины может совершенно отличаться от теоретической концептуальной формы.

Константы и литералы

Иногда в программе необходимо использовать фиксированное, заранее определенное значение. Например, программа управления воздушными полетами в окрестности некоторого аэропорта может содержать многочисленные ссылки на высоту аэропорта над уровнем моря. При создании подобной программы можно конкретно указывать это значение (скажем, 194 метра), когда оно потребуется. Такое явное указание конкретного значения называется **литералом**. Использование литералов приводит к появлению в программах операторов, подобных приведенному ниже:

```
EffectiveAlt = Altimeter + 194
```

Здесь `EffectiveAlt` и `Altimeter` являются переменными, а `194` — литералом. Таким образом, этот оператор требует, чтобы в переменную `EffectiveAlt` был помещен результат сложения литерала `194` с текущим значением переменной `Altimeter`.

Из главы 1 вспомним, что любую комбинацию из переменных, литералов и констант, соединяемых с помощью арифметических и логических операторов, называют **выражением**. В приведенном выше примере фраза `Altimeter + 194` как раз и является таким выражением.

В большинстве языков программирования текстовые литералы заключаются в одиночные или двойные кавычки, что позволяет отличить их от других элементов программ. Например, рассмотрим следующий оператор:

```
Фамилия = "Иванов"
```

Этот оператор может быть использован для присвоения переменной `Фамилия` текстового значения `"Иванов"`. А теперь слегка изменим этот оператор:

```
Фамилия = Иванов
```

Такой вариант оператора может быть использован для присвоения переменной `Фамилия` текущего значения переменной `Иванов`.

Как правило, применение литералов не считается лучшим стилем программирования, поскольку они затрудняют понимание тех выражений, в которых используются. Например, вернемся к нашему первому примеру.

```
EffectiveAlt = Altimeter + 194
```

Как читающий программу сможет узнать, что *именно* означает число 194? Более того, литералы могут существенно усложнить внесение в программу изменений, если это будет необходимо. Когда потребуется использовать данную программу управления воздушным движением для другого аэропорта, то значение высоты аэропорта над уровнем моря придется изменить. Если в программе для ссылки на эту высоту используется литерал 194, то каждую такую ссылку в программе нужно будет найти и изменить. Задача еще более усложнится, если окажется, что литерал 194 в некоторых случаях представляет также и некую другую величину, а не только высоту аэропорта над уровнем моря. Как тогда узнать, какой из литералов следует изменить, а какой оставить неизменным?

Для решения подобных проблем языки программирования позволяют присвоить описательные имена конкретным, неизменяемым значениям. Такие имена называют **константами**. В качестве примера приведем следующий оператор объявления, который может быть использован в программах на языках C++ и C#.

```
const int AirportAlt = 194;
```

Здесь идентификатор `AirportAlt` связывается с фиксированным значением, равным 194 (которое в программе будет иметь тип `integer`). Та же самая концепция на языке Java выражается следующим оператором:

```
final int AirportAlt = 194;
```

В свою очередь, на языке Swift этот оператор будет выглядеть так:

```
let AirportAlt = 194
```

Введя в программу подобное объявление, описательное имя `AirportAlt` можно будет использовать во всех тех случаях, в которых прежде использовался литерал 194. Например, если воспользоваться этой константой в нашем псевдокоде, то исходная инструкция

```
EffectiveAlt = Altimeter + 194
```

может быть заменена инструкцией

```
EffectiveAlt = Altimeter + AirportAlt
```

Согласитесь, что она гораздо лучше отражает смысл выполняемых действий. Более того, если во всей программе вместо литерала 194 будет использоваться такая константа, то при передаче этой программы в другой аэропорт, расположенный на высоте 80 метров над уровнем моря, изменить всего лишь один оператор объявления, в котором определяется значение высоты аэропорта над уровнем моря, — это все, что потребуется для приведения всех ссылок на эту высоту в программе к новому значению.

Операторы присваивания

Как только специальная терминология, которая будет использоваться в программе (например, переменные и константы), будет объявлена, программист сможет начать описывать реализуемые в ней алгоритмы. Эта задача решается посредством задания выполняемых операторов. Наиболее важным выполняемым оператором является **оператор присваивания**, предназначенный для присвоения переменной значения, определяемого как результат вычисления некоторого выражения (если говорить точнее, он требует сохранить вычисленное значение выражения в области основной памяти, идентифицируемой данной переменной). Синтаксически форма этого оператора обычно состоит из имени переменной, символа операции присваивания и выражения, определяющего то значение, которое должно быть присвоено переменной. Семантика этого оператора заключается в вычислении выражения, стоящего в его правой части, и сохранении полученного результата в переменной, указанной в левой части оператора. Например, в языках C, C++, C# и Java в результате выполнения приведенного ниже оператора переменной `Z` присваивается сумма значений переменных `X` и `Y`:

```
Z = X + Y;
```

Точка с запятой в конце строки, которая во многих императивных языках программирования используется как символ конца оператора, является единственным синтаксическим отличием от аналогичного оператора присваивания, записанного на языке Python. В некоторых других языках (таких, как Ada) эквивалентный оператор будет выглядеть следующим образом:

```
Z := X + Y;
```

Обратите внимание, что приведенные выше две строки кода различаются только синтаксисом представления самой операции присваивания, который в языках C, C++, C# и Java выглядит просто как знак равенства, а в языке ADA это двоеточие, за которым следует знак равенства. Вероятно, наилучший вариант нотации операции присваивания можно найти в языке APL, — этот язык был разработан Кеннетом Иверсеном в 1962 году. (APL — аббревиатура от *A Programming Language*.) В этом языке операция присваивания указывается стрелкой, направленной влево. Таким образом, приведенный выше оператор на языке APL записывается как

```
Z ← X + Y
```

К сожалению, символ "←" отсутствует в большинстве существующих клавиатур.

Основная сила оператора присваивания определяется широтой диапазона выражений, которые могут появляться в правой части оператора. В общем случае здесь может использоваться любое алгебраическое выражение,

построенное с использованием арифметических операций сложения, вычитания, умножения и деления, обычно представляемых знаками +, -, * и / соответственно. В некоторых языках (включая языки Ada и Python, но исключая языки C и Java) комбинация ** используется для представления операции возведения в степень, а значит, для представления выражения x^2 можно воспользоваться записью `x ** 2`.

Однако часто языки различаются способами, которые выбраны в них для интерпретации алгебраических выражений. Например, выражение 2 * 4 + 6 / 2 может дать результат 14, если его вычисление проводится справа налево, или результат 7, если вести вычисления слева направо. Подобные неоднозначности, как правило, устраняются посредством введения правил **предшествования операторов**, согласно которым определенные операции всегда выполняются раньше других. Традиционные правила алгебры требуют, чтобы умножение и деление всегда предшествовали сложению и вычитанию. Это означает, что при вычислении выражений умножение и деление выполняются первыми и только потом их результаты складываются и вычитаются. Если следовать этому соглашению, вычисление приведенного выше выражения даст результат 11. В большинстве языков программирования для изменения порядка выполнения операций используются скобки. В этом случае вычисление выражения 2 * (4 + 6) / 2 даст результат 10.

Многие языки программирования позволяют использовать один и тот же символ для обозначения нескольких типов операций. В таких случаях значение символа определяется типом операндов. Например, символ + обычно означает операцию сложения, если операнды являются числами, но в некоторых случаях, например в языках Java, Python и Swift, этот символ также означает операцию конкатенации, когда операндами являются строки символов. Таким образом, в результате вычисления `'abra' + 'cadabra'` будет получено строковое значение `'abracadabra'`. Такое многозначное использование символов операций называется **перегрузкой**. Хотя в одних языках программирования обеспечивается встроенная перегрузка некоторых часто употребляемых символов операторов, в других языках, таких как Ada, C++, C# и Swift, программистам разрешается определять собственные варианты перегрузки символов операций и даже добавлять собственные новые символы операций.

Управляющие операторы

Управляющие операторы предназначены для изменения порядка выполнения программы. Из всех операторов именно они привлекают к себе наибольшее внимание и порождают большинство споров. Главным виновником этого является самый простой из всех управляющих операторов — оператор `goto`.

Он позволяет изменить порядок выполнения программы путем перехода к другому месту программы, обозначенному специально для этой цели именем или числом. Таким образом, этот оператор является не чем иным, как прямым применением машинной команды JUMP, осуществляющей передачу управления в другое место программы. Проблема оператора goto заключается в том, что в языках программирования высокого уровня он позволяет программисту писать очень запутанные тексты, пронизанные операциями перехода, как крысиными норами. Вот пример:

```
    goto 40
20  Уклониться()
    goto 70
40  if (УровеньОпасности < Смертельный) then goto 60
    goto 20
60  СпастиДевушку()
70  ...
```

При этом единственный оператор, приведенный ниже, позволяет выполнить в точности то же самое.

```
if (УровеньОпасности < Смертельный):
  СпастиДевушку()
else:
  Уклониться()
```

Чтобы избежать подобных ситуаций, современные языки программирования включают более продуманный набор управляющих операторов, позволяющий представлять разветвленные последовательности выполнения с помощью единственной лексической структуры. Выбор, какие именно управляющие операторы будут включены в язык, является одним из аспектов его разработки. Основная цель здесь — предоставить разработчикам язык, который не только позволит выражать алгоритмы в наиболее удобной для прочтения форме, но также поможет программистам действительно достичь этого. Для достижения указанной цели следует идти путем ограничения использования тех языковых конструкций, которые исторически приводили к небрежному программированию, и одновременно путем поощрения использования хорошо продуманных проектных решений. Результатом такого подхода является практика, известная как **структурное программирование**, которая включает в себя методологию организованного проектирования в сочетании с надлежащим использованием управляющих структур языка программирования. Назначение этого подхода заключается в создании понятных и хорошо организованных программ, которые при этом полностью отвечают требованиям всех своих спецификаций.

Традиции в языках программирования

Как и при использовании естественных языков, пользователи различных языков программирования стремятся выработать собственные традиции, отличающие их от остальных программистов, и часто вступают в дебаты по поводу преимуществ, присущих, по их мнению, тем воззрениям, которых они придерживаются. Иногда отличия могут быть очень существенными, особенно при использовании различных парадигм, в других же случаях они оказываются совершенно незначительными. Например, несмотря на различия, существующие между процедурами и функциями (подробно об этом рассказывается в разделе 6.3), в языках C и Python оба конструкта называют функциями. Это происходит по той причине, что в языке Python процедура объявляется точно так же, как и как функция, но без определения возвращаемого ей значения. Аналогичный пример можно привести в отношении пользователей языка C++, которые ссылаются на функции, входящие в состав объектов, как на функции-члены, тогда как в объектно-ориентированной парадигме для них используется термин "метод". Это расхождение имеет место по той причине, что C++ был разработан как расширение языка C. Другим примером подобных расхождений является то, что в программах на языке Ada зарезервированные слова принято выделять либо всеми прописными буквами, либо полужирным шрифтом, тогда как пользователи языков C, C++, C#, Fortran и Java не придерживаются этой традиции.

Хотя в этой книге для представления примеров в большинстве глав используется язык Python, каждый конкретный пример представлен в форме, совместимой с традициями и стилем того языка, который был использован для его записи. Встретив подобный пример, вы должны понимать, что это всего лишь образец того, как теоретические идеи реализованы в реальном языке программирования, и он вовсе не является инструментом обучения конкретным деталям работы с тем или иным языком программирования. Постарайтесь увидеть лес за отдельными деревьями.

В главе 5 вы уже познакомились с двумя популярными структурами ветвления, представленными операторами `if-else` и `while`. Эти операторы присутствуют почти во всех императивных, функциональных и объектно-ориентированных языках программирования. Если говорить конкретнее, то операторы, которые в языке Python имеют вид

```
if (условие):
  оператор A
else:
  оператор B
```

и

```
while (условие):
  тело цикла
```

в языках C, C++, C# и Java следует записывать в таком виде:

```
if (условие) оператор A; else оператор B;
```

и

```
while (условие) { тело цикла }
```

Обратите внимание, что эти операторы совершенно идентичны во всех четырех этих языках, что является следствием того факта, что языки C++, C# и Java являются объектно-ориентированными расширениями императивного языка C. В противоположность этому в языке Ada те же операторы должны записываться в виде

```
IF условие THEN
  оператор A;
ELSE
  оператор B;
END IF;
```

и

```
WHILE условие LOOP
  тело цикла
END LOOP;
```

Другая широко распространенная структура ветвления часто представляется оператором `switch` (*переключить*) или `case` (*в случае, если*). Она представляет собой инструмент выбора одной последовательности операторов среди нескольких возможных в зависимости от текущего значения определенной переменной. Например, в языках C, C++, C# и Java оператор

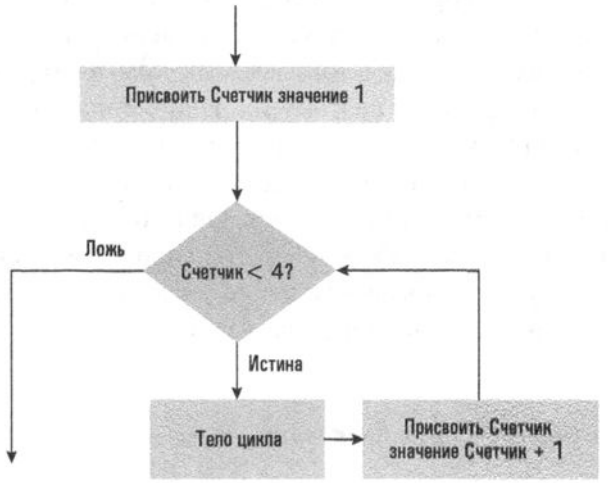
```
switch (переменная) {
  case 'A': оператор A; break;
  case 'B': оператор B; break;
  case 'C': оператор C; break;
  default: оператор D}
```

предполагает выполнение ветви `оператор A`, `оператор B` или `оператор C` в зависимости от того, содержит ли в данный момент `переменная` значение `A`, `B` или `C` соответственно. Если в текущий момент `переменная` содержит какое-то иное значение, выполняется ветвь `оператор D`. В языке Swift синтаксис эквивалентного оператора аналогичный за исключением того, что символы точки с запятой и ключевые слова `break` в нем не требуются. В языке Ada аналогичная структура должна быть представлена следующим образом:

```
CASE переменная IS
  WHEN 'A'=> оператор A;
  WHEN 'B'=> оператор B;
  WHEN 'C'=> оператор C;
```

```
  WHEN OTHERS=> оператор D;
END CASE;
```

Совершенно другая широко распространенная управляющая структура, которую часто называют циклом `for`, показана на рис. 6.7 вместе с ее представлением в языках C++, C# и Java. Эта структура отличается от структуры `for` в языке Python, которая обсуждалась в главе 5. Вместо неявного задания компонентов инициализации, модификации и проверки условия для цикла, в котором выполняется итерация по списку данных, в семействе языков, производных от языка C, в структуре `for` инициализация и модификация счетчика цикла, а также проверка условия выхода задаются явно в пределах одного оператора. Такой оператор удобен, когда тело цикла выполняется один раз для каждого значения переменной из заданного диапазона. В частности, представленные на рис. 6.7 операторы указывают, что тело цикла должно выполняться несколько раз: сначала — со значением переменной `Счетчик`, равным `1`, затем — со значением переменной `Счетчик`, равным `2`, и еще раз — со значением переменной `Счетчик`, равным `3`. В языках программирования, включающих оба типа структур `for`, их синтаксис может отличаться и выглядеть, например, как `foreach` для первого варианта (из главы 5) и `for...in` для второго, представленного в этом абзаце.




```
for (int Счетчик = 1; Счетчик < 4; Счетчик++)
    тело цикла;
```

Рис. 6.7. Структура цикла `for` и ее представление в языках C++, C# и Java

Назначение этих примеров — продемонстрировать, что типичные структуры ветвления программы присутствуют (с незначительными отличиями) во всех существующих императивных и объектно-ориентированных языках программирования. Однако, хотя это может показаться несколько неожиданным, с теоретической точки зрения компьютерной науки лишь немногие из этих структур являются действительно необходимыми для записи алгоритма решения любой задачи, имеющей алгоритмическое решение. Мы обсудим это позже, в главе 12. Пока же просто отметим, что изучение языка программирования не сводится к бесконечному изучению различных управляющих операторов. В действительности большинство управляющих структур, имеющихся в современных языках программирования, по сути, являются лишь вариантами тех структур, которые мы уже рассмотрели здесь.

Комментарии

Как показывает практика, независимо от того, насколько хорошо разработан язык программирования и насколько правильно в программе использовались его возможности, всякий раз, когда человек пытается разобраться в программе сколько-нибудь значительного размера, дополнительная информация о ней оказывается либо полезной, либо просто необходимой. По этой причине в языках программирования предусмотрены синтаксические конструкции для вставки в программу поясняющих операторов, называемых **комментариями**. Транслятор игнорирует эти конструкции, поэтому их наличие или отсутствие с машинной точки зрения никак не влияет на саму программу. Машинная версия программы, созданная транслятором, будет одной и той же независимо от того, содержит исходный код комментарии или нет. Однако содержащаяся в этих комментариях информация является важной частью программы с точки зрения человека. Без такого документирования большие и сложные программы часто могут превышать пределы понимания человека-программиста.



Основные положения для запоминания

- **Документация помогает разрабатывать и сопровождать программы как в индивидуальной работе программиста, так и в среде коллективной разработки программного обеспечения.**

Язык Visual Basic

Язык Visual Basic — это объектно-ориентированный язык программирования, разработанный компанией Microsoft в качестве инструмента, с помощью которого пользователи операционной системы Microsoft Windows могли бы создавать собственные приложения с графическим интерфейсом пользователя (GUI). В действительности Visual Basic представляет собой нечто больше, чем просто язык программирования, — это мощный интегрированный пакет разработки программного обеспечения, позволяющий программисту создавать графический интерфейс пользователя из заранее определенных компонентов (таких, как кнопки, флажки опций, текстовые поля, полосы прокрутки и т.п.) и настраивать работу этих компонентов в приложении, описывая их реакцию на различные события. Например, если речь идет о кнопке, программисту следует описать, что должно случиться, если пользователь щелкнет на ней. В главе 7 будет показано, что подобная стратегия создания программ из готовых компонентов представляет собой важнейшую современную тенденцию в области разработки программного обеспечения.

Растущая популярность операционной системы Windows в сочетании с удобством использования пакета Visual Basic в качестве инструмента для разработки программ способствовали тому, что язык Visual Basic стал одним из наиболее известных и широко используемых языков программирования в 1990-х годах. Потомки языка Visual Basic, такие как VB.NET, и в настоящее время остаются популярным выбором языка программирования для быстрого создания прототипов программ с графическим интерфейсом пользователя.

Язык Swift

Язык Swift — это мультипарадигмальный компилируемый язык программирования, представленный корпорацией Apple в 2014 году. Он включает в себя многие функции, характерные для императивных, объектно-ориентированных и функциональных языков. Будучи доступными как в среде операционной системы Linux, так и в среде операционных систем от Apple, библиотеки языка Swift и его среда разработки оптимизированы для использования больших объемов уже существующего программного кода на языке Objective C, доступного на различных платформах Apple. При использовании в среде разработки Xcode код на языке Swift может компилироваться и выполняться на виртуальных машинах "игровых площадок", имитирующих среду множества смартфонов и других устройств, работающих под управлением операционных систем от корпорации Apple, таких как iOS, watchOS и т.д. Многие приложения для смартфонов сейчас разрабатываются на языке Swift, предоставляющем полезные современные языковые функции, такие как наследование типов, поддержка шаблонов, автоматическое управление основной памятью и прочие функции обеспечения безопасности.

Существуют два основных способа выделения комментариев из прочего текста программы. Первый состоит в том, чтобы ограничить весь комментарий специальными маркерами, поставив один из них в начале, а второй — в конце текста комментария. Другой способ — отметить начало комментария и считать, что он занимает всю оставшуюся часть строки справа от маркера. Оба способа применяются в языках C++, C# и Java. В общем случае комментарий должен размещаться между парами символов `/*` и `*/`, однако в этих языках комментарий может также начинаться символами `//` и продолжаться до конца текущей строки. Таким образом, в языках C++, C# и Java оба приведенные ниже варианта оператора комментария являются правильными:

```
/* Это комментарий */
```

и

```
// Это тоже комментарий
```

Скажем несколько слов о том, что следует считать *осмысленным* комментарием. Начинающие программисты при внесении в программу комментариев в целях создания ее внутренней документации склонны делать это следующим образом:

```
Сумма = Стоимость + налог;    // Вычисляем Сумма, складывая Стоимость
и Налог
```

Подобные комментарии совершенно излишни и скорее просто увеличивают размер текста программы, нежели проясняют ее суть. Запомните, что задача внутренней документации — объяснить другому программисту смысл программы, а не просто перифразировать выполняемые в ней действия. Более приемлемым комментарием для приведенного выше оператора было бы краткое пояснение, зачем вычисляется значение `Сумма`, если это не очевидно из предыдущего текста. Например, приведенный ниже комментарий, несомненно, окажется полезнее, чем предыдущий:

```
/* Переменная Сумма используется ниже при вычислении значения
   переменной ОбщийИтог и после этого будет уже не нужна */
```

Кроме того, программа, в которой комментарии беспорядочно разбросаны среди выполняемых операторов, может быть еще более непонятной, чем программа вовсе без комментариев. Лучше собрать все относящиеся к некоторому модулю программы комментарии в одном месте, например в начале модуля, где читатель программы сможет найти необходимые объяснения. Здесь также целесообразно будет описать задачу и общие свойства данного программного модуля. Если этот подход принять для всех модулей, то получится единообразно выполненная программа, каждая часть которой будет состоять из блока поясняющих комментариев и следующих за ними выполняемых операторов. Такое единообразие программы существенно улучшает ее читабельность.



Основные положения для запоминания

- **Документация с описанием всех компонентов программы, таких как отдельные сегменты ее кода и процедуры, будет очень полезна как при ее разработке, так и в процессе сопровождения.**

Блочное программирование

Языки визуального программирования, которые иногда также называют *языками блочного программирования*, представляют собой альтернативу традиционной среде программирования, ориентированной на текстовое представление программного кода. В них для создания программы пользователям предоставляется возможность манипулировать на экране графическими блоками. Базовые языковые конструкты, такие как циклы, условные операторы, функции и другие, представляются в этих языках в виде графических блоков, которые скрепляются между собой подобно элементам конструкторов Lego с целью создания необходимой последовательности программных инструкций.

Благодаря простоте использования визуальные языки программирования стали популярным выбором для вводных курсов по компьютерным наукам, особенно для школьников и студентов. Языки Alice, Scratch, StarLogo и App Inventor — вот несколько примеров блочных языков, широко используемых в образовании.

В принципе, визуальные блочные языки являются не менее мощными, чем текстовые языки программирования, но преобладание упрощенных образовательных визуальных языков оставило у многих учащихся ошибочное впечатление, что их можно использовать только для простейших задач. Фактически мощные блочные языки могут использоваться для программирования встроенных систем, манипулирования сложными медиаартефактами и имитационного моделирования.

6.2. Вопросы и упражнения

1. Почему использование констант вместо литералов считается лучшим стилем программирования?
2. В чем разница между оператором объявления и выполняемым оператором?
3. Перечислите некоторые из наиболее распространенных типов данных.
4. Назовите некоторые из наиболее распространенных управляющих структур, существующих в императивных и объектно-ориентированных языках программирования.
5. Чем различаются однородные и неоднородные массивы?

6.3. Процедурные элементы программ

В предыдущих главах мы убедились в преимуществах разделения больших программ на более мелкие и удобные в работе модули. В этом разделе мы сосредоточимся на концепции функции, представляющей собой основной способ получения модульного представления программы в императивных языках программирования. Как уже упоминалось в главе 5, в языках программирования на всем протяжении их развития использовалось много терминов для обозначения этой важной концепции: подпрограмма, процедура, метод, функция, — иногда с тонкими оттенками различного смыслового значения. Среди строго императивных языков преобладает термин "функция", тогда как в объектно-ориентированных языках ему часто предпочитается термин "метод", — в тех случаях, когда программист описывает, как объект должен реагировать на различные события.

Функции

Функция в самом общем смысле представляет собой набор инструкций, необходимых для решения определенной задачи, который может использоваться как абстрактный инструмент другими программными единицами. Управление передается функции в тот момент, когда требуются ее услуги, и возвращается вызывающей программной единице после того, как функция завершит свое выполнение (рис. 6.8). Процесс передачи функции управления часто называют *вызовом функции*. Также мы будем называть тот элемент программы, который потребовал выполнения функции, *вызывающим* элементом.

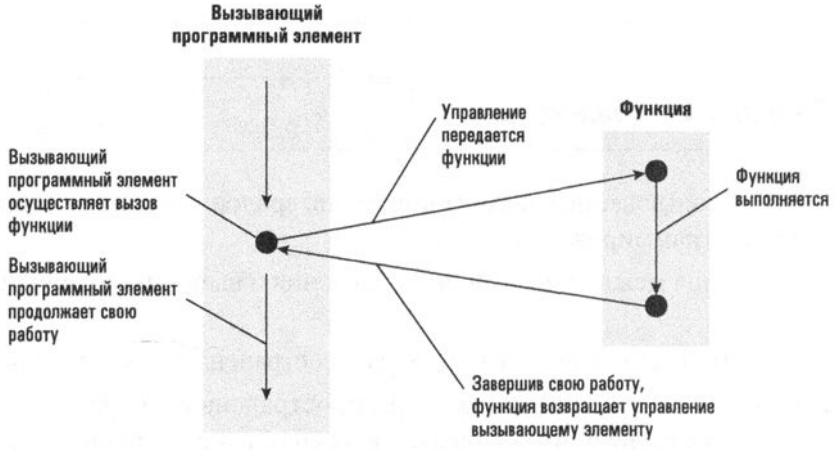


Рис. 6.8. Передача и возврат управления при вызове функции

Как и в наших примерах на языке Python в главе 5, функции обычно записываются как отдельные программные единицы. Такая единица начинается с оператора, называемого **заголовком**, в котором, помимо всего прочего, указывается имя этой функции. После заголовка следуют операторы, детально описывающие данную функцию. Эти операторы обычно упорядочивают по тому же принципу, который используется в традиционных императивных программах: в начале находятся операторы объявления, описывающие используемые в функции переменные, за которыми следуют выполняемые операторы, определяющие этапы выполнения алгоритма, реализуемого в результате выполнении функции.

Как правило, все переменные, объявленные в функции, чаще всего являются **локальными переменными**. Это означает, что на них можно ссылаться только внутри данной функции. Подобный подход исключает возможные недоразумения, которые могут иметь место, когда две процедуры, написанные независимо одна от другой, используют переменные с одинаковыми именами. Та часть программы, в которой можно ссылаться на некоторую переменную, называется **областью видимости** этой переменной. Таким образом, областью видимости локальной переменной будет та функция, в которой она была объявлена. Переменные, области видимости которых не ограничиваются определенной частью программы, называют **глобальными переменными**. Большинство языков программирования предоставляет средства определения, какой именно является объявляемая переменная: локальной или глобальной.

Большинство современных языков программирования позволяет вызвать функцию на выполнение, просто указав ее имя. Например, если `ПолучитьДанные`, `СортироватьДанные` и `ВывестиДанные` — это имена функций, осуществляющих считывание списка имен, его сортировку и вывод на печать, то программа, выполняющая чтение, сортировку и распечатку такого списка, может выглядеть следующим образом:

```
ПолучитьДанные()
СортироватьДанные()
ВывестиДанные()
```

Обратите внимание, что за счет назначения каждой из функций имени, указывающего на выполняемые ею действия, такая сжатая форма записи программы выглядит как последовательность команд, отражающих смысл программы.

Разработка новых функций изначально может оказаться достаточно сложной задачей для начинающих программистов. Хотя сам синтаксис определения новых функций в большинстве языков прост, процесс распознавания корректных абстракций и их кодирование в виде краткого фрагмента кода на выбранном языке программирования может потребовать определенной практики. Начинающим часто приходится реализовать несколько конкретных примеров решения

разных вариантов поставленной перед ними задачи, прежде чем им удается найти тот обобщающий подход, который позволит создать единственную функцию, успешно удовлетворяющую этим и всем прочим возможным ситуациям. Лишь отвлекшись от избыточного внимания к деталям конкретных случаев, можно четко уяснить те важные общие черты, которые свойственны им всем. Основная интеллектуальная задача при создании хороших функций — отыскание абстракции необходимого уровня.



Основные положения для запоминания

- **Процесс выявления абстракции предполагает отвлечение от деталей и обобщение требуемой функциональности.**
- **Переход к абстракции требует выработки обобщенных концепций за счет извлечения общих функции из конкретных примеров.**

Параметры

Функции часто пишутся с использованием обобщенных элементов, которые становятся конкретными, только когда функция вызывается на выполнение. Например, в предыдущей главе на рис. 5.11 функция сортировки выражена в терминах некоторого обобщенного понятия списка, а не какого-то конкретного списка имен. В псевдокоде этой функции было решено идентифицировать это обобщенное понятие именем, которое в заголовке функции помещается в скобки. В результате на рис. 5.11 эта функция имеет заголовок

```
def Сортировка(Список):
```

Под ним следует описание процедуры сортировки, в котором обобщающее имя `Список` используется везде, где необходима ссылка на сортируемый список. Если потребуется воспользоваться этой функцией для сортировки списка гостей на свадьбу, достаточно будет просто следовать всем инструкциям в теле функции, предполагая, что обобщающее имя `Список` в данном случае будет ссылаться как раз на тот список гостей, который требуется отсортировать. Если же потребуется отсортировать список участников соревнований, достаточно будет просто интерпретировать имя `Список` как ссылку на перечень имен участников соревнований.

Такие обобщенные элементы в функциях принято называть **параметрами**. Если говорить точнее, используемые в функциях обобщающие элементы называют **формальными параметрами**, а те конкретные значения, которые назначаются этим формальным параметрам при вызове функции, принято называть

фактическими параметрами. В определенном смысле формальные параметры в функции представляют собой некое гнездо, в которое при вызове функции на выполнение будет помещено фактическое значение.



Основные положения для запоминания

- **Параметризация позволяет обобщить конкретное решение.**

Как и в языке Python, многие языки программирования требуют, чтобы при определении функции все ее формальные параметры были перечислены в скобках непосредственно в заголовке функции. В качестве еще одного примера на рис. 6.9 представлено определение функции с именем `ProjectPopulation` в таком виде, в каком оно должно быть записано на языке C. В этой функции ожидается, что при вызове ей будет передано конкретное значение годового темпа роста некоей популяции. Используя это значение, функция вычисляет предполагаемый размер популяции на следующие 10 лет, приняв 100 как ее исходный размер, и сохраняет вычисленные значения в глобальном массиве с именем `Population`.

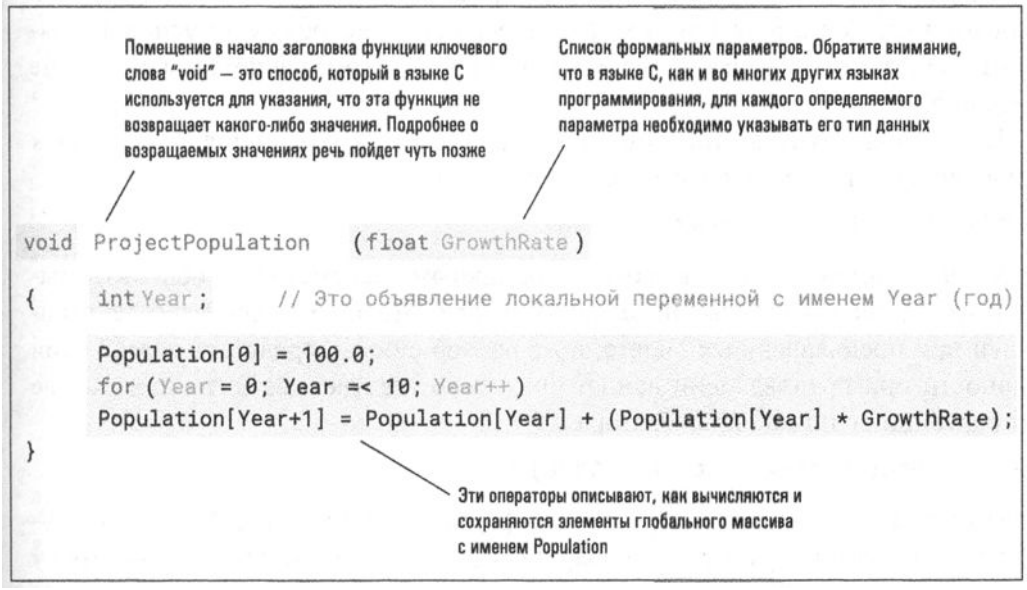


Рис. 6.9. Функция `ProjectPopulation`, написанная на языке программирования C

Во многих языках программирования скобки используются и как средство идентификации значений фактических параметров, передаваемых функции при ее вызове. При этом оператор вызова функции на выполнение состоит из имени функции, за которым следует список фактических значений параметров, взятый в скобки. Следовательно, в нашем псевдокоде для вызова функции `ProjectPopulation` на выполнение можно использовать оператор в том виде, в каком он приведен ниже:

```
Вызвать ProjectPopulation() используя 0,03 как значение годового
темпа роста
```

Однако в программе на языке C для вызова функции `ProjectPopulation`, представленной на рис. 6.9 с использованием значения 0,03 как годового темпа роста популяции, следует использовать оператор

```
ProjectPopulation(0.03);
```

Тот же самый синтаксис будет иметь и аналогичный оператор в языке Python, но без завершающего символа точки с запятой.

Когда функции передается более одного параметра, значения фактических параметров, одно за другим, ассоциируются с ее формальными параметрами в порядке их следования, определенном в заголовке функции. Первый фактический параметр ассоциируется с первым формальным параметром, второй — со вторым и т.д. Когда значения всех фактических параметров будут успешно связаны с соответствующим им формальным параметром, начнется выполнение функции.

Чтобы подчеркнуть этот момент, предположим, что функция `ПечататьЧек` была определена с заголовком следующего вида:

```
def ПечататьЧек(Плательщик, Сумма):
```

Здесь `Плательщик` и `Сумма` являются формальными параметрами, используемыми в этой функции для ссылок на личность, для которой этот чек будет действителен при предъявлении к оплате, и на размер суммы, предоставляемой этой личности при предъявлении данного чека, соответственно. Тогда при вызове этой функции с помощью оператора

```
ПечататьЧек('Сергей Соколов', 1500)
```

она будет выполнена таким образом, что формальный параметр `Плательщик` будет ассоциирован с фактическим параметром, представляющим собой строку символов `'Сергей Соколов'`, а формальный параметр `Сумма` будет ассоциирован с числовым значением `1500`. А теперь посмотрим, что произойдет, если для вызова этой функции использовать оператор

```
ПечататьЧек(1500, 'Сергей Соколов')
```

В этом случае числовое значение 1500 будет ассоциировано с формальным параметром Плательщик, а строка символов 'Сергей Соколов' будет ассоциирована с формальным параметром Сумма. Понятно, что в этом случае выполнение функции приведет к получению ошибочных результатов.

Задача передачи данных между фактическими и формальными параметрами может решаться различными способами в различных языках программирования. В некоторых языках программирования передача данных от фактических параметров к формальным осуществляется посредством копирования данных, представляемых формальными параметрами, и передачи этих копий в вызываемую функцию. При таком подходе функция может манипулировать лишь предоставленными ей копиями данных, а сами исходные данные в вызывающей части программы всегда останутся неизменными. Говорят, что такие параметры **передаются по значению**. Обратите внимание, что передача данных по значению защищает данные в вызывающем модуле от ошибочного их изменения плохо разработанной функцией. Например, если вызывающий модуль передает функции табельный номер работника, то крайне нежелательно, чтобы она этот номер изменила.

К сожалению, передача параметров по значению неэффективна, особенно когда параметрами являются большие блоки данных. Более эффективный способ передачи параметров функции состоит в предоставлении ей прямого доступа к фактическим параметрам посредством указания их адресов. В этом случае говорят, что параметры **передаются по ссылке**. Напомним, что передача параметров по ссылке позволяет вызываемой функции модифицировать данные в вызывающем модуле. Такой подход был бы желателен, например, при сортировке списка. И действительно, в этом случае вызов функции сортировки имел бы результатом переупорядочивание исходного списка.

В качестве примера предположим, что функция Demo была определена так, как показано ниже.

```
def Demo (Formal):
  Formal = Formal + 1
```

Далее предположим, что переменной Actual было присвоено значение 5, а затем функция Demo была вызвана с помощью оператора

```
Demo(Actual)
```

В результате, если параметры передаются по значению, изменения, внесенные в переменную Formal при выполнении функции Demo, не отразятся на значении переменной Actual (рис. 6.10). Однако, если параметры передаются по ссылке, после выполнения функции Demo значение переменной Actual увеличится на единицу (рис. 6.11).

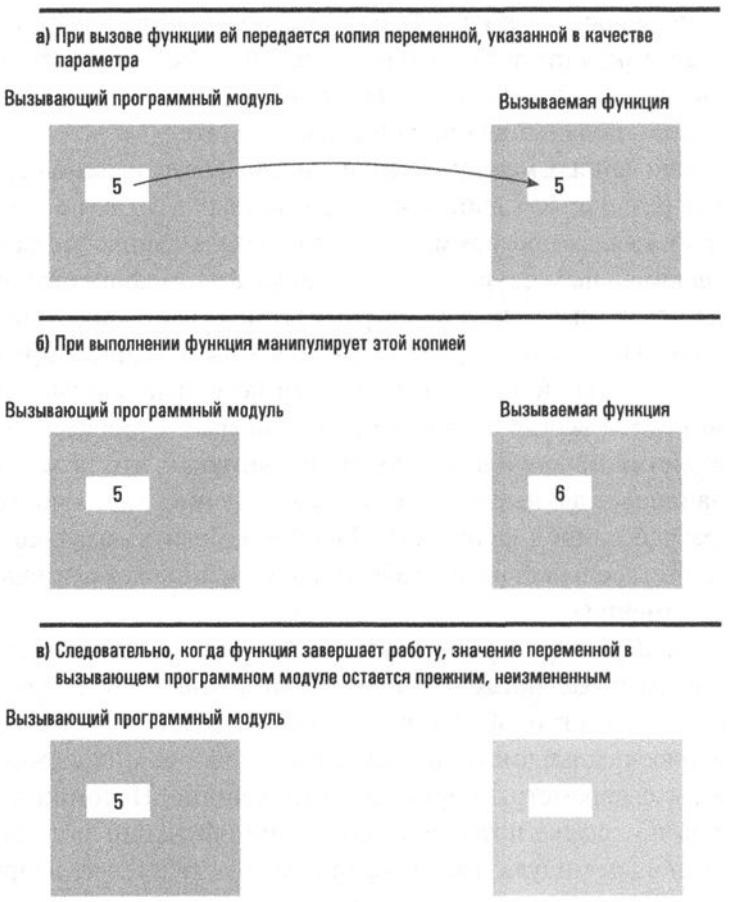


Рис. 6.10. Выполнение функции `Demo` с передачей параметров по значению

В разных языках программирования передача параметров осуществляется различными методами, но в любом случае использование параметров позволяет писать функции в абстрактном представлении и применять их к конкретным данным в нужный момент. Обобщенные функции, манипулирующие абстрактными данными, являются краеугольным камнем эффективного повторного использования программного обеспечения. Опытные программисты всегда держат под рукой подборку излюбленных, хорошо протестированных функций, которые они используют для решения типичных, часто возникающих проблем.

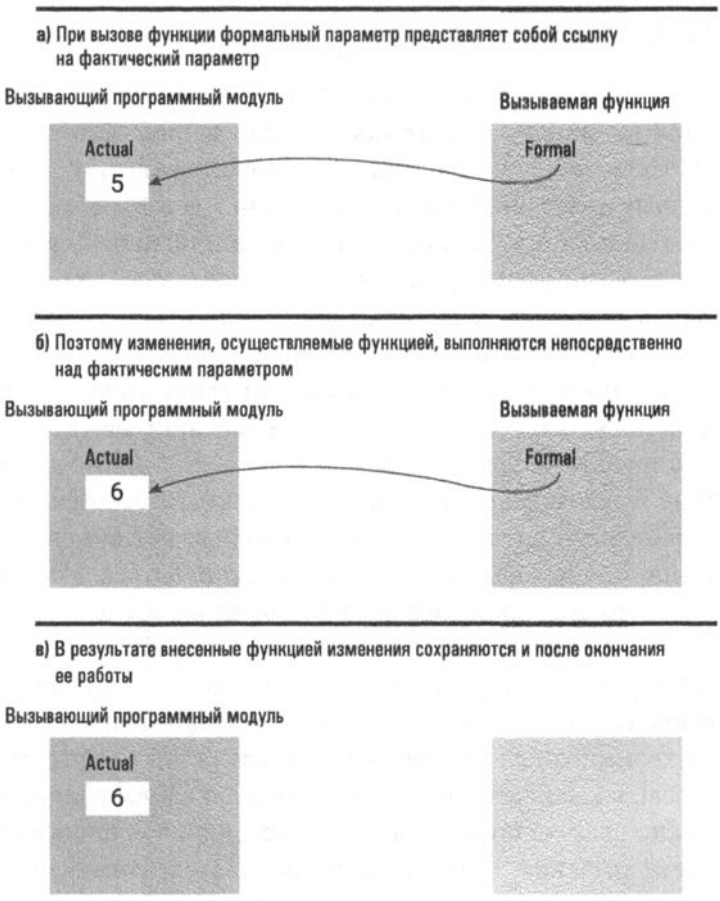


Рис. 6.11. Выполнение функции `Demo` с передачей параметров по ссылке



Основные положения для запоминания

- **Именно абстрактные обобщенные функции, реальные данные которым передаются при вызове как входные параметры, обеспечивают возможность повторного использования программного обеспечения.**

Функции, возвращающие значения

А теперь давайте обратимся к слегка измененной концепции функции, которая встречается во многих языках программирования. Иногда назначение функции может заключаться в получении некоторого значения, а не в выполнении определенных действий. (Обратите внимание на тонкое различие между функцией, назначение которой состоит в получении оценки количества изделий, которые могут быть проданы, и функцией, реализующей простую компьютерную игру: в первой акцент делается на получении единственного значения, а во второй — на выполнении действия.) В действительности в компьютерных науках термин *функция* является производным от математической концепции *функция*, которая является представлением взаимосвязи между набором исходных и результирующих значений. В этом смысле все примеры функций на языке Python или псевдокоде, приведенные здесь до этого момента, представляют собой лишь особый случай функций, у которых нет выходных или возвращаемых значений, о которых следовало бы беспокоиться. (Формально все функции в языке Python должны возвращать значение, а следовательно, подходят для применения к ним математического термина *функция*. Те функции, в которых отсутствует оператор `return`, по умолчанию возвращают значение `None`.) В родословном древе языков программирования использовалось много специальных терминов для разделения этих двух типов программных единиц. В языке Pascal, в свое время пользовавшимся большим влиянием, термин **процедура** был специально предназначен для ссылок на подпрограммы, которые *не возвращали* значения. В языках C и Java для идентификации функций или методов, не возвращающих значения, используется ключевое слово **void**. В среде программистов на языке Python к функциям, в которых возвращаемое значение определяется, принято применять термин **fruitful function** (*плодотворная функция*), чтобы отличать их от тех функций, в которых возвращаемое значение не определяется. Независимо от специфических языковых терминов, большинство языков программирования включает название *функция* для одного и того же понятия — программной единицы, которая передает назад в вызывающую программу единственное значение, называемые значением функции. Таким образом, как результат выполнения функции некоторое значение будет вычислено и передано обратно в вызывающую программу. Это значение может быть сохранено в переменной для последующего обращения либо немедленно использовано в вычислениях. Например, в языках C, C++, Java и C# программист может написать следующий оператор:

```
ОценкаПродажЯнварь = ОценкаПродаж(Январь);
```

В этом операторе переменной ОценкаПродажЯнварь присваивается результат выполнения функции ОценкаПродаж с целью определения, какой объем продаж изделий можно ожидать в январе. Однако программист может написать и иначе:

```
if (ПродажиПрошлыйЯнварь < ОценкаПродаж(Январь)) ...
else ...
```

В этом случае в программе предполагается выполнение различных действий в зависимости от того, окажется ожидаемый объем продаж изделий в январе больше объема их продаж в январе прошлого года. Обратите внимание, что во втором варианте вычисленное функцией значение непосредственно используется для определения, какая ветвь программы будет выполняться далее, и нигде не сохраняется.

Функции, возвращающие значение, определяются в программах почти точно так, как и функции, которые ничего не возвращают. Различия состоят лишь в том, что в первом случае заголовок функции обычно начинается с указания типа данных того значения, которое возвращает функция, а ее определение обычно заканчивается оператором `return`, в котором и указывается само возвращаемое значение. На рис. 6.12 представлено определение функции `CylinderVolume`, предназначенной для вычисления объема цилиндра, как оно может быть написано на языке C. (В действительности программисты на языке C предпочитают использовать более сжатую форму записи, но здесь, из чисто педагогических соображений, был выбран данный более многословный вариант записи.) При вызове функция получает конкретные значения для своих формальных параметров `Radius` (радиус) и `Height` (высота), а возвращает результат вычисления объема цилиндра указанного размера. В результате эта функция при необходимости может быть использована в любом месте программы с помощью оператора, подобного следующему, с помощью которого вычисляется стоимость содержимого цилиндра радиусом 3,45 и высотой 12,7 единиц при заданной в переменной `CostPerVolUnit` стоимости единицы объема этого содержимого:

```
Cost = CostPerVolUnit * CylinderVolume(3.45, 12.7);
```

В предыдущих главах в примерах использовалось несколько встроенных функций языка Python, возвращающих значение, включая функции `input()` и `math.sqrt()`, но пока еще ни разу не приводилось определения своей собственной. Если обратиться к функции `CylinderVolume`, представленной на рис. 6.12, то ее версия на языке Python будет выглядеть следующим образом.

```
def CylinderVolume(Radius, Height):
  Volume = math.pi * Radius * Radius * Height
  return Volume
```

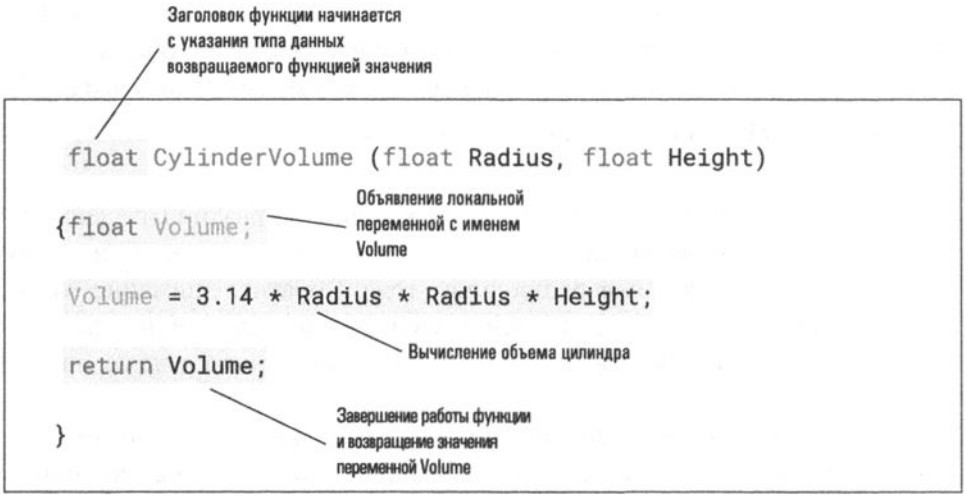


Рис. 6.12. Возвращающая значение функция `CylinderVolume`, написанная на языке C

Событийно-управляемое программное обеспечение

Выше рассматривались случаи, когда функции явно вызывались операторами, расположенными в различных местах программы. Однако иногда функции должны активизироваться неявно, при наступлении какого-то события. Например, в графическом интерфейсе пользователя (GUI) функция, описывающая реакцию программы на щелчок на командной кнопке, не вызывается из другого программного модуля, а активизируется непосредственно в ответ на щелчок мышью на данном элементе. Программное обеспечение, содержащее подобные функции, называется **событийно-управляемым**. Короче говоря, такое программное обеспечение состоит из функций, описывающих реакцию системы на различные события. При функционировании системы эти функции бездействуют, пока не наступит требуемое событие; после этого они активизируются, выполняют свою задачу и вновь возвращаются в состояние покоя.



Основные положения для запоминания

- **Использование уже существующих корректных алгоритмов как строительных блоков при конструировании новых алгоритмов придает уверенности в том, что новый алгоритм также будет корректным.**

Языки программирования позволяют разработчикам программного обеспечения создавать собственные функции как средство, обеспечивающее повторное использование программного кода, повышающее наглядность и читабельность программ и способствующее написанию корректного кода. Сложный, цельный алгоритм сложнее протестировать, чем эквивалентную программу, которая предварительно была продуманно разбита на более мелкие функциональные элементы. Хорошо написанные функции можно тестировать независимо от общего алгоритма, что увеличивает возможности поиска скрытых ошибок в программном обеспечении. Если разработчик убедится, что отдельные функциональные блоки работают верно, это укрепит его уверенность в том, что вся система также будет функционировать корректно.

6.3. Вопросы и упражнения

1. Что понимается под термином "область видимости" переменной?
2. Чем обычная функция отличается от функции, возвращающей значение?
3. Почему многие языки программирования реализуют операторы ввода-вывода так, будто они представляют собой вызов функции?
4. Чем отличаются формальные параметры от фактических?
5. В чем различие между вызовом функции с использованием передачи параметров по ссылке и с использованием передачи параметров по значению?

6.4. Реализация языка

В этом разделе вы познакомитесь с процессом перевода программы с языка высокого уровня в такую форму, которая может быть выполнена машиной.

Процесс трансляции

Процесс перевода программы с одного языка на другой называется **трансляцией**. Программа в своем оригинальном виде называется **исходной программой**, а оттранслированная ее версия называется **объектным кодом**. Процесс трансляции состоит из трех этапов — лексического анализа, синтаксического анализа и генерации кода, которые выполняются элементами транслятора, называемыми **лексическим анализатором**, **синтаксическим анализатором** и **генератором кода** (рис. 6.13).

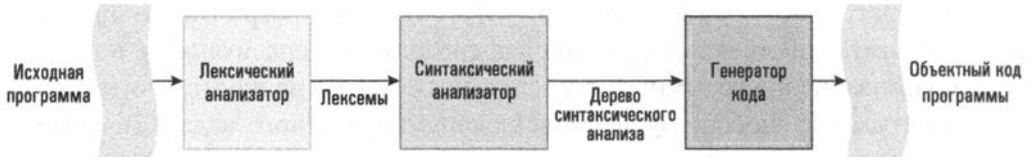


Рис. 6.13. Процесс трансляции программы

Лексический анализ — это процесс выделения отдельных символьных строк из текста исходной программы. Например, символы 153 должны интерпретироваться транслятором не как совокупность цифр, состоящая из единицы, пятерки и тройки, а как единое числовое значение, равное ста пятидесяти трем. Аналогично этому слова в программе представляют собой самостоятельные и неразделимые единицы текста (лексемы), хотя и состоят из отдельных символов. Большинство людей выполняют лексический анализ без каких-либо видимых усилий. И действительно, когда нас просят прочитать текст вслух, мы произносим целые слова, а не отдельные буквы, из которых они состоят.

Реализация языков Java и C#

В некоторых случаях, таких как управление анимированной веб-страницей, управляющее анимацией программное обеспечение передается через Интернет вместе с самой страницей, а затем выполняется на машине клиента. Если программное обеспечение представляет собой исходный текст программы, то при просмотре страницы возникает дополнительная задержка, связанная с трансляцией программы в соответствующий машинный язык, прежде чем ее можно будет выполнить. Однако если это программное обеспечение пересылать сразу на машинном языке, то может оказаться, что для разных типов клиентских машин потребуются различные версии программы.

Компании Sun Microsystems и Microsoft решили эту проблему, разработав "универсальный машинный язык", названный *байт-кодом* в случае языка Java и *.NET Common Intermediate Language* в случае языка C#, на который переводятся исходные тексты программ. Хотя эти языки не являются настоящим машинным языком, они разработаны так, что транслировать их можно достаточно быстро. Поэтому, если программное обеспечение, написанное на языке Java или C#, будет оттранслировано на соответствующий "универсальный машинный язык", то его можно будет передавать на другие машины в Интернете, на которых оно сможет достаточно эффективно выполняться. В одних случаях такое выполнение осуществляется интерпретатором. В других случаях программа непосредственно перед выполнением может быстро транслироваться на настоящий машинный язык, — этот процесс называют **компиляцией "на лету"** или ЛТ-компиляцией (от англ. *Just-in-time compilation*).

Таким образом, лексический анализатор символ за символом считывает текст исходной программы, определяя, какие группы символов образуют самостоятельные единицы текста, так называемые **лексемы**. Затем эти единицы классифицируются, чтобы выяснить, что они собой представляют — числа, слова, арифметические операторы и т.д. Как только единица текста классифицирована, лексический анализатор генерирует ее битовый образ, называемый лексемой (*token*), и передает его синтаксическому анализатору. При выполнении этого процесса лексический анализатор игнорирует все комментарии, содержащиеся в тексте исходной программы.

Из сказанного выше можно прийти к заключению, что синтаксический анализатор (parser) анализирует программу уже в терминах лексических единиц (лексем), а не отдельных символов. Задачей синтаксического анализатора является объединение этих единиц в операторы. Действительно, синтаксический анализ — это процесс идентификации грамматической структуры программы и распознавания роли каждого ее компонента. Это техническая сторона синтаксического анализа, которая может привести к некоторой задержке при чтении следующего предложения.

> Человек, которого лошадь, проигравшая скачки, сбросила, не был ранен.
>
> (Попробуйте еще и это: "То, что есть, то есть. То, чего нет, того нет. То, чего нет, не есть то, что есть"!)

Чтобы упростить процесс синтаксического анализа, ранние языки программирования требовали, чтобы каждый оператор программы размещался в определенном месте бланка исходного текста. Такие языки называются языками **с фиксированным форматом**. В настоящее время большинство языков программирования являются языками **со свободным форматом**. Это означает, что расположение операторов в тексте не имеет значения. Преимущество свободного формата состоит в том, что программист теперь может писать программу так, чтобы человеку было проще ее читать. В таких программах обычно принято использовать отступы, чтобы помочь читателю понять внутреннюю структуру оператора. Например, рассмотрим следующий оператор:

```
if Цена < Наличные then платить наличными else использовать кредитную карточку
```

При использовании свободного формата программист может записать этот же оператор в более удобном виде:

```
if Цена < Наличные
  then Платить наличными
  else использовать кредитную карточку
```

Для того чтобы машина могла выполнять синтаксический анализ программы, написанной на языке со свободным форматом, синтаксис языка следует разрабатывать таким образом, чтобы структуру программы можно было идентифицировать, не обращая внимания на пробелы в исходном тексте программы. По этой причине во многих языках со свободным форматом для обозначения конца оператора используются знаки пунктуации (например, точка с запятой), а также **ключевые слова**, такие как `if`, `then` и `else`, предназначенные для выделения начала отдельных фраз. Эти ключевые слова обычно называют *зарезервированными словами*, чтобы подчеркнуть, что программист не может использовать их в своей программе для иных целей. В этом отношении язык Python является нетипичным, поскольку ему свойственны некоторые аспекты языков со свободным форматом, но при этом в нем для разметки структуры требуется строго указывать отступы, а не использовать знаки препинания, такие как точка с запятой и фигурные скобки.

Процесс синтаксического анализа базируется на совокупности правил, определяющих синтаксис языка программирования. В своей совокупности эти правила называют **грамматикой** языка. Один из способов представления этих правил состоит в использовании **синтаксических диаграмм**, наглядно иллюстрирующих грамматическую структуру программы. На рис. 6.14 представлена синтаксическая диаграмма оператора `if-then-else` языка Python, обсуждавшегося в главе 4. Эта диаграмма показывает, что структурно оператор `if-then-else` начинается с ключевого слова `if`, за которым следует *Логическое выражение*, завершаемое двоеточием, вслед за которым следует *Оператор с отступом*. Эта комбинация может дополняться (но необязательно) ключевым словом `else`, двоеточием и еще одним *Оператором с отступом*. Обратите внимание, что члены выражения, или термы, которые действительно присутствуют в операторе `if-then-else`, заключены в овалы, тогда как те термы, которые подлежат дальнейшему уточнению, например *Логическое выражение* или *Оператор с отступом*, помещены в прямоугольники. Термы, подлежащие дальнейшему уточнению (т.е. те, которые заключены в прямоугольники), называются **нетерминальными**, а термы, окруженные овалами, — **терминальными**. В полном описании синтаксиса языка программирования **нетерминальные** термы описываются дополнительными диаграммами.

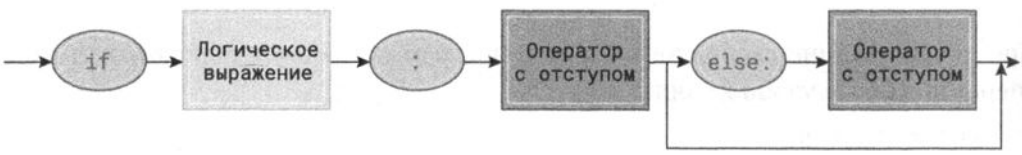


Рис. 6.14. Синтаксическая диаграмма оператора `if-then-else` языка Python

Как более сложный пример на рис. 6.15 представлен набор синтаксических диаграмм, описывающих синтаксис структуры *Выражение*, являющейся структурой простого арифметического выражения. Первая диаграмма описывает структуру *Выражение* как состоящую из компонента *Терм*, за которым сначала могут (но необязательно) следовать символы + или -, а затем — другой компонент *Выражение*. Вторая диаграмма описывает структуру компонента *Терм* как состоящую либо из отдельного компонента *Фактор*, либо из компонента *Фактор*, за которым следует символ × или символ ÷, за которым следует другой компонент *Терм*. Наконец, последняя, третья диаграмма описывает структуру компонента *Фактор* как одного из символов x, y или z.

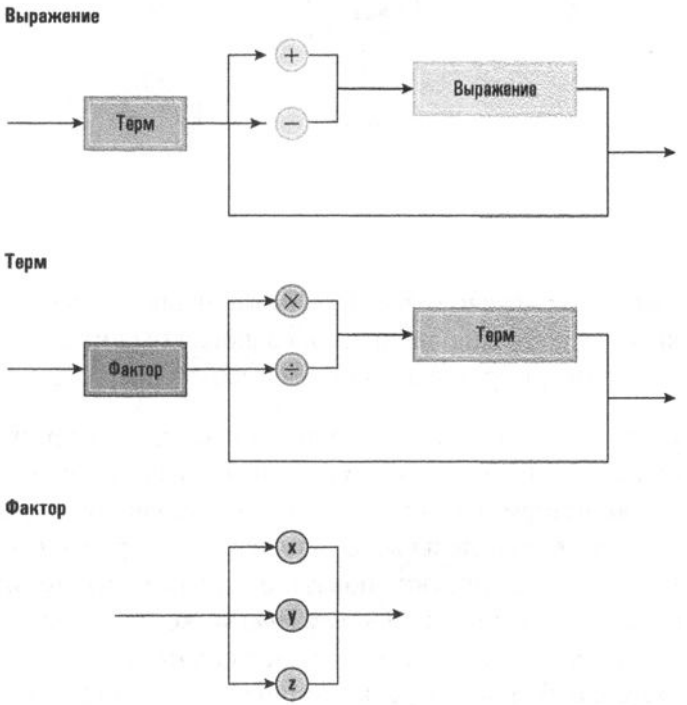


Рис. 6.15. Синтаксическая диаграмма, описывающая структуру простого алгебраического выражения

Метод, с помощью которого отдельная строка программы проверяется на соответствие некоторой совокупности синтаксических диаграмм, можно наглядно представить с помощью **дерева синтаксического анализа** (рис. 6.16). На этом рисунке приведено дерево синтаксического анализа, выполненного на основании набора диаграмм, представленных на рис. 6.15, для следующей строки:

```
x + y × z
```

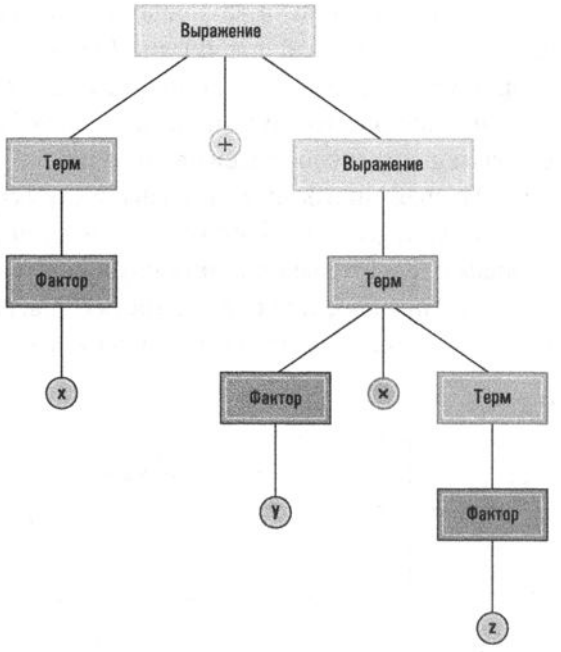


Рис. 6.16. Дерево синтаксического анализа строки `x + y × z`, выполненного на основании синтаксических диаграмм, представленных на рис. 6.15

Обратите внимание, что данное дерево анализа начинается расположенным вверху нетерминальным термом *Выражение* и на каждом очередном уровне дерева показано, как нетерминальные термы этого уровня раскладываются на составные части, вплоть до отдельных символов, из которых и состоит исходная строка. В частности, на рисунке показано, что (в соответствии с первой диаграммой на рис. 6.15) компонент *Выражение* может быть разложен на компонент *Терм*, за которым следует символ +, после которого следует еще один компонент *Выражение*. В свою очередь, компонент *Терм* может быть разложен (в соответствии со второй диаграммой на рис. 6.15) на компонент *Фактор* (который оказывается символом x), а последующий компонент *Выражение* может быть разложен (с использованием третьей диаграммы на рис. 6.15) на компонент *Терм* (который оказывается выражением y × z).

Процесс синтаксического анализа программы, по сути, сводится к построению дерева синтаксического анализа для ее исходного текста. Фактически дерево синтаксического анализа отражает, как синтаксический анализатор понял грамматическую структуру программы. По этой причине синтаксические правила, описывающие эту грамматическую структуру, не должны допускать построения двух разных деревьев синтаксического анализа для одной и той же строки, поскольку это приведет к неоднозначности в работе синтаксического

анализатора. Грамматика, которая допускает два различных дерева синтаксического анализа для одной строки, называется **неоднозначной грамматикой**.

Неоднозначности в грамматиках могут быть практически незаметными. И действительно, показанное на рис. 6.14 синтаксическое правило содержит подобную неточность. Оно допускает построение двух разных деревьев синтаксического анализа для одного и того же оператора, показанного ниже, что отражено на рис. 6.17:

```
if B1: if B2: S1 else: S2
```

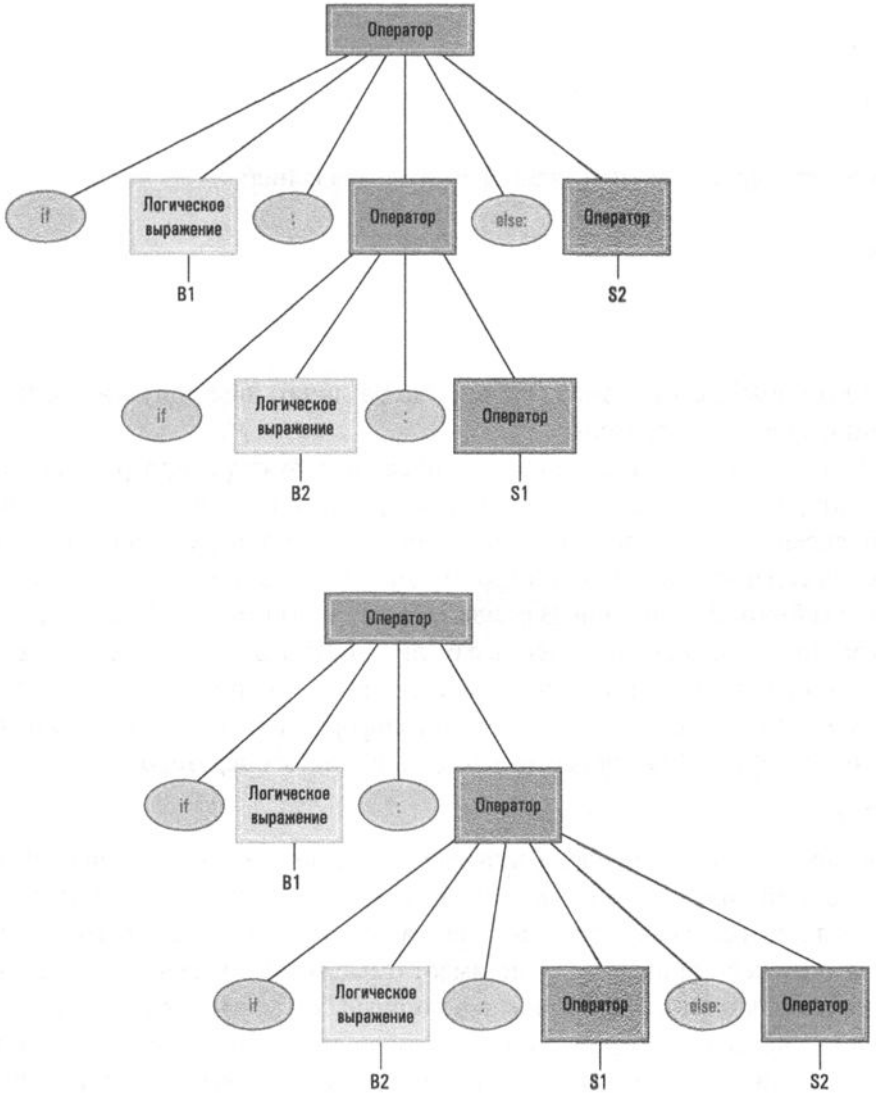


Рис. 6.17. Два различных дерева синтаксического анализа для оператора `if B1: if B2: S1 else: S2`

Заметим, что эти интерпретации существенно различаются. Первая предполагает, что оператор S2 будет выполнен, если выражение B1 окажется ложным. Из второй интерпретации следует, что оператор S2 должен выполняться только тогда, когда выражение B1 истинно, а выражение B2 ложно.

Синтаксические определения формальных языков программирования разрабатываются так, чтобы избежать подобных неоднозначностей. В языке Python этой проблемы можно избежать за счет применения отступов. В частности, этот оператор следовало бы записать как

```
if B1:
  if B2:
    S1
  else:
    S2
```

Но возможен и другой вариант записи этого выражения:

```
if B1:
  if B2:
    S1
else:
  S2
```

В результате появляется возможность четко различать обе допустимые интерпретации исходного оператора.

При синтаксическом анализе грамматической структуры программы можно идентифицировать отдельные операторы и провести различие между операторами объявления и выполнимыми операторами. При обнаружении операторов объявления содержащаяся в них информация записывается в таблицу, которую называют **таблицей символов**. В результате таблица символов будет содержать информацию о том, какие переменные были описаны в программе, а также какие типы и структуры данных связаны с этими переменными. В дальнейшем синтаксический анализатор использует эту информацию в ходе анализа выполняемых операторов. Ниже представлен один из таких операторов:

```
z = x + y
```

В частности, чтобы определить значение символа +, синтаксический анализатор должен знать, какой тип данных связан с переменными x и y. Если переменная x имеет тип `float`, а переменная y — тип `character`, то операция суммирования переменных x и y не имеет смысла; ее появление должно рассматриваться как ошибка. Если обе переменные, x и y, имеют тип `integer`, то синтаксический анализатор потребует от генератора кода создать на машинном языке команду с кодом операции сложения двух целых чисел, а если эти переменные имеют тип `float`, то синтаксический анализатор потребует использовать операцию сложения двух чисел с плавающей точкой. Если же обе

переменные, x и y, имеют тип character, то синтаксический анализатор может потребовать, чтобы генератор кода создал такую последовательность команд на машинном языке, которая потребуется для выполнения операции конкатенации этих двух строк символов.

Несколько особый случай возникает, если переменная x имеет тип integer, а переменная y — тип float. Здесь концепция операции сложения вполне допустима, но суммируемые значения представлены в несовместимом виде. В этом случае синтаксический анализатор может предложить генератору кода создать цепочку команд, предназначенных для предварительного преобразования значения одной из переменных в другой тип, а затем выполнить требуемое сложение. Такое неявное преобразование типов называется **приведением типов** (coercion).

Неявное приведение типов не одобряется многими разработчиками языков, поскольку неявное преобразование типа может привести к изменению значения элемента данных и как следствие — к программным ошибкам. Они считают, что неявное приведение типов свидетельствует о неправильной разработке программы и что синтаксический анализатор ни в коем случае не должен покрывать эти недостатки. В результате большинство современных языков программирования являются **строго типизированными**, т.е. они требуют, чтобы все операции в программе выполнялись над данными согласованных типов без необходимости неявного приведения типов. Синтаксические анализаторы этих языков рассматривают любые несоответствия типов как ошибку. Некоторые языки, такие как Java, разрешают неявное приведение типов в тех пределах, в которых оно является **продвижением типа**, что означает преобразование значения с меньшей точностью в значение с большей точностью. Однако неявное приведение типов, при котором исходное значение может оказаться измененным, рассматривается как ошибка. В большинстве таких случаев программист все же может потребовать выполнить необходимое преобразование типов, описав его явным образом. Иначе говоря, он как бы уведомляет компилятор о том, что знает, что здесь будет применено преобразование типов.

Последним этапом трансляции является **генерация кодов** — процесс создания команд машинного языка, реализующих выполнение операторов, распознанных синтаксическим анализатором. Этот процесс включает множество различных аспектов, один из которых — повышение эффективности генерируемого программного кода. Например, рассмотрим задачу трансляции последовательности из двух операторов:

```
x = y + z
w = x + z
```

Если эти операторы будут оттранслированы по отдельности, каждый из них потребует передачи данных из основной памяти в ЦП, прежде чем можно будет

выполнить требуемую операцию сложения. Для достижения более высокой эффективности генератор кода должен суметь распознать, что, когда первый оператор будет выполнен, переменные x и z уже будут находиться в регистрах общего назначения центрального процессора, и, следовательно, нет необходимости снова загружать их из памяти перед выполнением второго оператора. Реализация подобных нюансов при построении программы называется **оптимизацией кода** и является важной задачей генератора кода.



Основные положения для запоминания

- **Текст программы на определенном языке программирования часто транслируется в код на другом языке (более низкого уровня), прежде чем его можно будет выполнить на компьютере.**

И наконец, необходимо отметить, что этапы лексического анализа, синтаксического анализа и генерации кода никогда не выполняются строго в указанной последовательности. На самом деле они тесно переплетаются между собой. Лексический анализатор начинает с чтения символов текста исходной программы и идентификации первых лексем. Затем он передает эти лексемы синтаксическому анализатору. Каждый раз, когда синтаксический анализатор получает очередную лексему, он анализирует считываемую в данный момент грамматическую структуру. В результате он может запросить у лексического анализатора следующую лексему либо, если он распознал законченную фразу или оператор, обратиться к генератору кода для порождения соответствующих машинных инструкций. В свою очередь, каждый поступивший запрос вынуждает генератор кода строить соответствующие машинные команды и добавлять их к объектному коду программы. Как можно заметить, задача перевода программы с одного языка на другой естественно согласуется с объектно-ориентированной парадигмой. Исходный текст программы, лексический анализатор, синтаксический анализатор, генератор кода и, наконец, созданный им объектный код могут рассматриваться как независимые объекты, которые взаимодействуют, посылая друг другу сообщения по мере выполнения своих функций (рис. 6.18).

Пакеты для разработки программ

Программные инструменты, такие как редакторы и трансляторы, часто объединяют с другими элементами, используемыми в процессе разработки программного обеспечения, в общий пакет, функционирующий как единая инте-

грированная программная система. В соответствии со схемой классификации, предложенной в разделе 3.2, такую систему следует отнести к прикладному программному обеспечению. Используя такой программный пакет, программист получает удобный доступ к текстовому редактору, предназначенному для написания программ, транслятору, необходимому для перевода программы на машинный язык, и разнообразным инструментам отладки программ, позволяющим отслеживать выполнение неправильно работающих программ и обнаруживать имеющиеся в них ошибки.

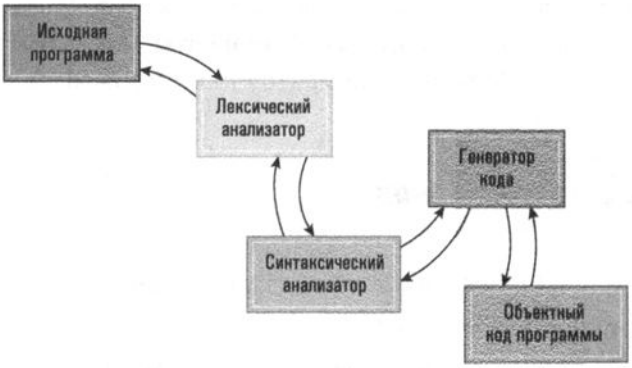


Рис. 6.18. Объектно-ориентированный подход к процессу трансляции программ

Такая интегрированная система имеет много достоинств. Вероятно, наиболее важное из них — возможность легко переходить от текстового редактора к отладчику и обратно при написании и проверке программ. Более того, многие пакеты разработки программ позволяют связывать разрабатываемые программные модули таким образом, что доступ к ним заметно упрощается. Некоторые пакеты обеспечивают ведение записей, относящихся к тем программным единицам в группе взаимосвязанных модулей, которые были изменены со времени последнего сеанса проверки их функционирования. Подобные средства очень удобны при разработке больших программных систем, в состав которых входит множество отдельных модулей, разрабатываемых разными программистами.

Кроме того, текстовые редакторы в пакетах обычно настроены для работы с тем языком программирования, который используется в данном пакете. Такие текстовые редакторы, предназначенные специально для разработки программного обеспечения, обычно позволяют автоматически применять отступы строк, что фактически уже стало стандартом для большинства языков программирования. В некоторых случаях текстовый редактор способен распознавать и автоматически дописывать ключевые слова сразу же после того, как программист введет лишь несколько их первых букв. Более того, подобные редакторы часто

могут автоматически выделять ключевые слова в исходных текстах программ (например, цветом), поскольку такое выделение значительно упрощает чтение программ.

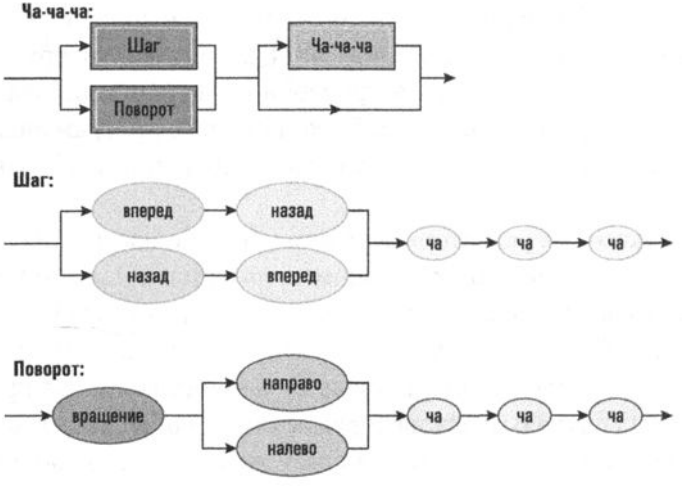
В следующей главе вы узнаете, что разработчики программного обеспечения все чаще ищут способы, с помощью которых новые программные системы могут быть построены из готовых блоков, называемых компонентами, что приводит к новой модели разработки программного обеспечения, называемой *компонентной архитектурой*. В пакетах разработки программного обеспечения, основанных на модели компонентной архитектуры, часто используются графические интерфейсы, в которых компоненты могут быть представлены на дисплее в виде графических элементов или пиктограмм. В такой среде

6.4. Вопросы и упражнения

1. Опишите три основных этапа процесса трансляции.
2. Что такое таблица символов?
3. В чем различие между терминальными и нетерминальными термами?
4. Нарисуйте дерево синтаксического анализа для выражения

   `x × y + x + z`

   исходя из синтаксических диаграмм, представленных на рис. 6.15.
5. Напишите несколько строк, формат которых будет соответствовать структуре **Ча-ча-ча**, определенной с помощью следующих синтаксических диаграмм.



программист (или сборщик компонентов) выбирает нужные ему компоненты с помощью мыши. Далее выбранный компонент может быть необходимым образом настроен с помощью специального редактора, входящего в состав пакета, а затем присоединен к другим компонентам посредством указания позиции и щелчка мышью. Такие пакеты представляют собой важный шаг вперед в поиске лучших инструментов разработки программного обеспечения.

6.5. Объектно-ориентированное программирование

В разделе 6.1 уже говорилось, что объектно-ориентированная парадигма предусматривает разработку активных программных модулей, называемых **объектами**, каждый из которых содержит процедуры, описывающие реакцию объекта на различные входные сигналы. Объектно-ориентированный подход к решению задачи состоит в выявлении и описании необходимых объектов, а также связанных с ними методов в виде самодостаточного отдельного программного модуля. В свою очередь, объектно-ориентированные языки программирования предоставляют операторы и другие средства для описания объектов и их поведения. В этом разделе вы познакомитесь с некоторыми из этих операторов, представленными в том виде, в каком они записываются в языках C++, Java и C#, которые являются тремя наиболее известными объектно-ориентированными языками из числа используемых на сегодняшний день.

Классы и объекты

Рассмотрим задачу разработки простой компьютерной игры, в которой игрок должен защитить Землю от падающих метеоров, стреляя в них мощными лазерами. Каждый лазер содержит ограниченный внутренний источник питания, который частично разряжается при каждом выстреле. Как только этот источник истощается, лазер становится бесполезным. Каждый лазер должен иметь возможность реагировать на соответствующие команды, чтобы нацелить луч дальше вправо, нацелить луч дальше влево и, наконец, выстрелить своим лучом.

В объектно-ориентированной парадигме каждый лазер в компьютерной игре следует реализовать в виде объекта, содержащего сведения о количестве оставшейся в его источнике питания энергии и функции, обеспечивающие смещение его луча вправо и влево, а также осуществление выстрела. Поскольку все объекты лазеров имеют одни и те же свойства, они могут быть описаны с помощью единственного, общего для всех них шаблона. В объектно-ориентированной парадигме программирования такой шаблон для коллекции однотипных объектов называют **классом**.

В главе 8 будет подробно обсуждаться сходство между классом и типом данных. На данный момент мы просто отметим, что класс описывает коллекцию объектов практически точно так, как концепция примитивного типа данных `integer` включает в себя общие характеристики таких чисел, как 1, 5 и 82. Как только программист включит описание класса в программу, этот шаблон можно будет использовать для создания и манипулирования объектами данного "типа" во многом таким же образом, каким примитивный тип целых чисел позволяет манипулировать "объектами" типа `integer`.

В языках C++, Java и C# класс описывается с помощью оператора следующего вида.

```
class Name
{
    .
    .
    .
}
```

Здесь `Name` — это имя, по которому на этот класс можно будет ссылаться в любом месте программы. Все прочие свойства класса описываются в фигурных скобках. Например, на рис. 6.19 представлено описание класса с именем `LaserClass`, описывающего структуру объекта, представляющего в нашей компьютерной игре лазер. Класс состоит из объявления переменной с именем `RemainingPower` (тип `integer`) и трех функций — `turnRight`, `turnLeft` и `fire`. Эти функции содержат набор операторов, которые необходимо выполнить для реализации соответствующих действий. В результате любой объект, который будет создан с использованием этого шаблона, будет включать следующие элементы: переменную с именем `RemainingPower` и три функции с именами `turnRight()`, `turnLeft()` и `fire()`.

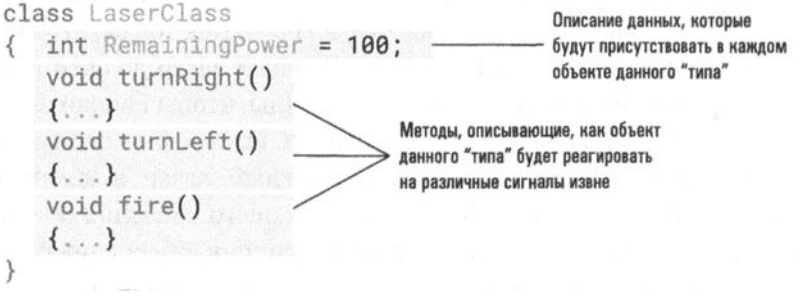


Рис. 6.19. Структура класса, описывающего в компьютерной игре лазер

Переменную, которая находится внутри объекта, такую как `RemainingPower`, называют **переменной экземпляра**, а функции внутри объекта называют **методами** (или *функциями-членами* в терминологии языка C++). Обратите

внимание, что на рис. 6.19 переменная экземпляра `RemainingPower` описывается с использованием оператора объявления, подобного тем, которые обсуждались в разделе 6.2, а методы описываются в форме, напоминающей определение функций, как это было представлено в разделе 6.3. В конце концов, объявление переменных экземпляра и описание методов в сущности представляют собой концепции императивного программирования.

Как только в программе будет описан класс `LaserClass`, появится возможность объявить три переменные, `Laser1`, `Laser2` и `Laser3`, как имеющие "тип" `LaserClass`, с помощью следующего оператора:

```
LaserClass Laser1, Laser2, Laser3;
```

Обратите внимание, что формат этого оператора ничем не отличается от формата оператора

```
int x, y, z;
```

который мы использовали для объявления трех переменных с именами `x`, `y` и `z` типа `integer`, как это было представлено в разделе 6.2. Оба оператора состоят из названия "типа", за которым следует список имен объявляемых переменных. Все различие состоит лишь в том, что в последнем операторе объявляется, что переменные с именами `x`, `y` и `z` будут использоваться в программе для ссылок на элементы данных типа `integer` (который является примитивным типом), тогда как в предыдущем операторе объявляется, что переменные с именами `Laser1`, `Laser2` и `Laser3` будут использоваться в программе для ссылок на элементы "типа" `LaserClass` (который является "типом", определенным в пределах программы).

Как только переменные `Laser1`, `Laser2` и `Laser3` были объявлены как имеющие "тип" `LaserClass`, им можно присваивать значения. В нашем случае значения должны быть объектами, соответствующими "типу" `LaserClass`. Такое присваивание может быть выполнено с помощью операторов присваивания, но чаще оказывается удобнее присваивать переменным начальные значения в том же операторе объявления, который используется для их объявления. Такие присвоения исходных значений автоматически выполняются в случае объявлений в языке C++. Таким образом, оператор

```
LaserClass Laser1, Laser2, Laser3;
```

не только объявляет в программе переменные `Laser1`, `Laser2` и `Laser3`, но и одновременно создает в ней три объекта с "типом" `LaserClass`, по одному в качестве значения каждой из переменных. В языках Java и C# присвоение исходных значений объектным переменным осуществляется практически таким же образом, как и присвоение исходных значений создаваемым переменным примитивных типов. В частности, как оператор

```
int x = 3;
```

не только объявляет переменную x с типом integer, но и присваивает этой переменной исходное значение 3, так и оператор

```
LaserClass Laser1 = new LaserClass();
```

не только объявляет переменную Laser1 с "типом" LaserClass, но и создает новый объект на основании шаблона LaserClass и присваивает этот объект как исходное значение переменной Laser1.

А сейчас нам необходимо сделать паузу и подчеркнуть различия между *классом* и *объектом*. Класс представляет собой шаблон, на основании которого создается конкретный объект. Один класс может использоваться для создания любого количества однотипных объектов. Часто на объект ссылаются как на **экземпляр** того класса, на основании которого он был создан. Следовательно, в нашей компьютерной игре Laser1, Laser2 и Laser3 — это три переменные, значения которых являются экземплярами класса LaserClass.

Воспользовавшись операторами объявления для создания переменных Laser1, Laser2 и Laser3 и назначив им в качестве исходных значений объекты, мы можем продолжить создание программы нашей игры, написав императивные операторы, которые будут активизировать соответствующие методы в этих объектах (в объектно-ориентированной терминологии это называется отправкой объектам *сообщений*). Например, можно заставить объект, назначенный переменной Laser1, выполнить его метод fire, воспользовавшись оператором

```
Laser1.fire();
```

Или же можно потребовать от объекта, назначенного переменной Laser2, выполнить его метод turnLeft(), воспользовавшись оператором

```
Laser2.turnLeft();
```

В действительности эти операторы — нечто большее, чем просто вызов функции. Так, первый оператор представляет собой вызов функции (метода) fire() в *пределах объекта*, присвоенного в качестве значения переменной Laser1, а последний оператор представляет собой вызов функции turnLeft() в *пределах объекта*, назначенного в качестве значения переменной Laser2.

На этом этапе создания наша компьютерная игра уже позволяет получить представление об общей структуре типичной объектно-ориентированной программы, как показано на рис. 6.20. Такая программа будет состоять из определений различных классов, подобных тем, которые были приведены на рис.6.19. Каждое из этих определений будет описывать структуру одного или более объектов, используемых в программе. Кроме того, программа будет содержать императивный программный сегмент (обычно связанный с именем main), описывающий последовательность действий, которые должны быть выполнены в начале работы программы. В этом сегменте будут содержаться операторы

объявления, подобные приведенным выше объявления объектов, представляющих лазеры, предназначенные для создания используемых в программе объектов, а также императивные операторы, вызывающие на выполнение методы в пределах этих объектов.

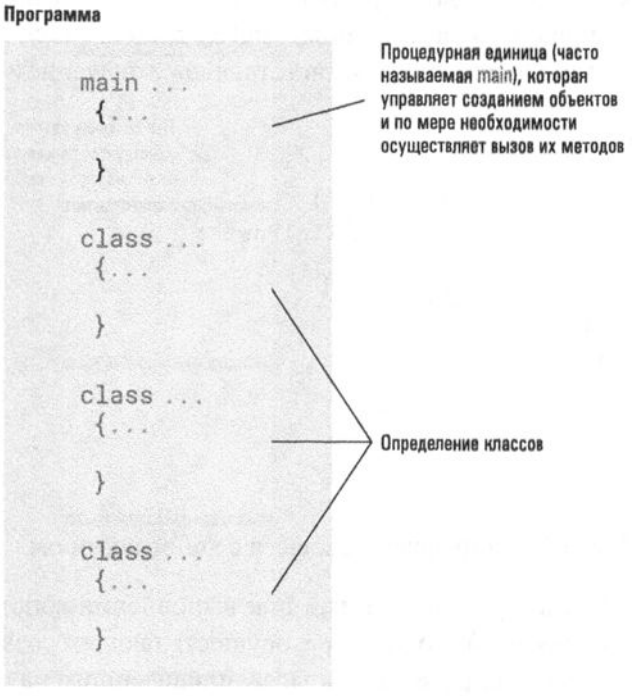


Рис. 6.20. Структура типичной объектно-ориентированной программы

Конструкторы

При создании объекта часто необходимо выполнять определенные действия по его настройке. Например, в нашем примере компьютерной игры может потребоваться присвоить различным лазерам различные значения исходного заряда, а это будет означать, что переменным экземпляра с именем `RemainingPower` в различных объектах нужно будет присвоить различные исходные значения. Такие задачи при инициализации объекта решаются с помощью специальных методов, называемых конструкторами и определяемых в соответствующих классах. Методы-конструкторы выполняются автоматически при создании экземпляра объекта на основании описания класса. Метод-конструктор в пределах определения класса идентифицируется по тому факту, что его имя будет таким же, как имя самого класса.

На рис. 6.21 представлено расширенное определение класса `LaserClass`, в сравнении с приведенным ранее на рис. 6.19. Обратите внимание, что это определение содержит конструктор в виде метода с именем `LaserClass`. В этом методе переменной экземпляра `RemainingPower` присваивается значение, переданное ему в качестве параметра. Таким образом, при создании объекта на основании этого описания класса данный метод выполняется и переменная `RemainingPower` инициализируется соответствующим значением.

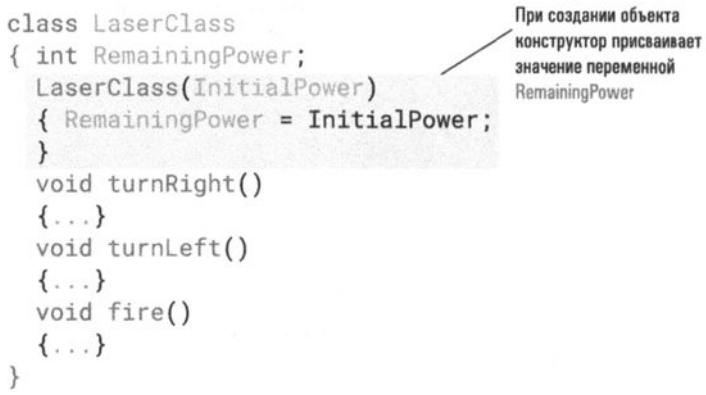


Рис. 6.21. Определение класса с конструктором

Фактический параметр, используемый при выполнении конструктора, указывается в списке параметров в операторе, осуществляющем создание объекта. Таким образом, исходя из определения класса, приведенного на рис. 6.21, программист на языке C++ может воспользоваться, например, оператором

```
LaserClass Laser1(50), Laser2(100);
```

для создания двух объектов класса `LaserClass`: одного — с именем `Laser1` и исходным значением уровня мощности, равным `50`, и другого — с именем `Laser2` и исходным значением уровня мощности, равным `100`. Программисты на языках Java и C# могли бы решить ту же задачу, воспользовавшись операторами

```
LaserClass Laser1 = new LaserClass(50);
LaserClass Laser2 = new LaserClass(100);
```

Дополнительные свойства и функциональные возможности

А теперь давайте предположим, что необходимо улучшить нашу компьютерную игру так, чтобы игрок, набравший определенное количество очков, был вознагражден посредством подзарядки некоторых лазеров до их первоначального уровня мощности. Эти лазеры будут иметь те же свойства, что и другие лазеры, за исключением того, что они будут перезаряжаемыми.

Для того чтобы упростить описание объектов, имеющих больше одинаковых свойств, чем разных, многие объектно-ориентированные языки программирования позволяют одному классу включать свойства другого посредством механизма, называемого **наследованием**. В качестве примера предположим, что для разработки нашей компьютерной игры использовался язык Java. Сначала в ней может использоваться описанный ранее оператор определения класса `LaserClass`, представляющий те свойства, которые являются общими для всех лазеров в программе. Затем для описания еще одного класса `RechargeableLaser`, определяющего перезаряжаемый лазер, можно использовать другой оператор, приведенный ниже.

```
class RechargeableLaser extends LaserClass
{
    .
    .
    .
}
```

(В языках C++ и C# программист может заменить в этом операторе слово `extends` просто двоеточием.) Здесь фраза `extends` указывает, что данный класс *наследует* все свойства и функции класса `LaserClass`, и, помимо этого, содержит собственные свойства и функции, которые описываются в фигурных скобках. В нашем случае в скобках должно присутствовать описание нового метода (вероятно, с именем `recharge`), в котором будут реализованы все действия, необходимые для восстановления в переменной экземпляра `RemainingPower` ее исходного значения. Как только этот класс будет определен, можно будет воспользоваться оператором

```
LaserClass Laser1, Laser2;
```

для объявления переменных `Laser1` и `Laser2`, представляющих обычные лазеры, а затем оператором

```
RechargeableLaser Laser3, Laser4;
```

для объявления переменных `Laser3` и `Laser4`, представляющих лазеры с дополнительными возможностями, реализованными в классе `RechargeableLaser`.

Использование механизма наследования ведет к появлению разнообразных объектов, имеющих сходные, но все же различающиеся характеристики, что, в свою очередь, приводит к явлению, напоминающему перегрузку, речь о которой шла в разделе 6.2. (Напомним, что понятие перегрузки означает использование одного и того же символа, например знака `+`, для представления разных операций в зависимости от типа указанных операндов.) Предположим, что объектно-ориентированный графический пакет состоит из разнообразных объектов, каждый из которых описывает некоторую фигуру (окружность, прямоугольник,

треугольник и т.п.). Каждое изображение состоит из совокупности таких объектов. Для каждого объекта известны его размер, положение и цвет, а также то, как он реагирует на сообщения, требующие от него определенных действий, например перемещение в новое положение или отображение самого себя на экране. Для того чтобы нарисовать изображение в целом, мы просто отсылаем сообщение "нарисуй себя" каждому из объектов, образующих это изображение. Однако программы, рисующие отдельные объекты, изменяются в зависимости от их формы — процесс рисования квадрата отличается от процесса рисования окружности. Данный механизм специфической интерпретации одного и того же сообщения называется **полиморфизмом**, а соответствующее сообщение — полиморфным.

Другим свойством, связанным с объектно-ориентированным программированием, является **инкапсуляция**, которая означает ограничение доступа к внутренним свойствам объекта. Если сказать, что некоторое свойство объекта является *инкапсулированным*, это будет равноценно утверждению, что доступ к этому свойству может иметь только сам объект. Инкапсулированные свойства называются закрытыми (`private`), а свойства, доступные извне объекта, — открытыми (`public`).

В качестве примера давайте вернемся к описанию класса `LaserClass`, представленному ранее на рис. 6.19. Напомним, что в нем присутствовало определение переменной экземпляра `RemainingPower` и трех методов: `turnRight()`, `turnLeft()` и `fire()`. Эти методы предназначались для того, чтобы другие элементы программы могли запросить у экземпляра объекта `LaserClass` выполнение соответствующих действий. С другой стороны, изменение значения переменной `RemainingPower` разрешалось только внутренним методам объекта. Никаким иным программным элементам прямой доступ к ее значению не предоставлялся. Чтобы явно описать эти требования, можно было просто объявить переменную `RemainingPower` с модификатором `private` (*закрытая*), а методы `turnRight()`, `turnLeft()` и `fire()` — с модификатором `public` (*открытые*), как показано на рис. 6.22. После вставки этих модификаторов при трансляции программы любая попытка получить доступ к значению `RemainingPower` извне объекта, в котором он находится, будет идентифицирована как ошибка, и это вынудит программиста внести необходимые изменения, прежде чем он получит возможность продолжить работу.

Инкапсуляция — это мощный инструмент абстракции, позволяющий разработчикам программного обеспечения отделить описание поведения объекта (то, что выполняется его методами) от реализации объекта (от внутренней информации о том, как называются имеющиеся в нем переменные, какие значения в них находятся в тот или иной момент времени и т.д.). В результате другие объекты в системе будут вынуждены полагаться только на сервисы, явно предоставляемые открытыми методами и свойствами объекта, что делает всю

систему более устойчивой к ошибкам в тех случаях, когда во внутренний код объекта впоследствии могут вноситься какие-либо изменения.

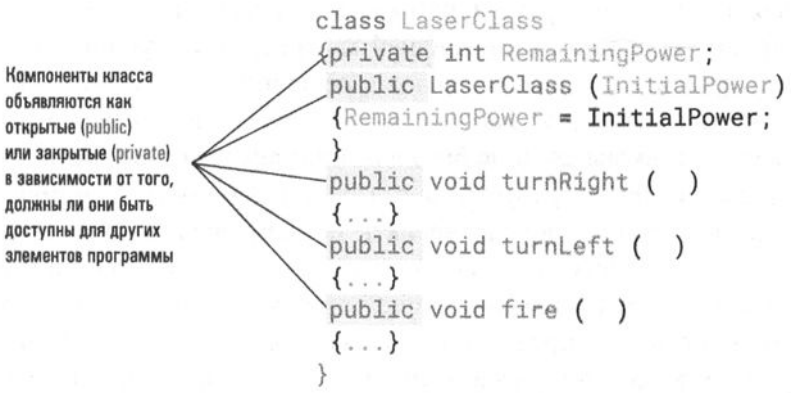


Рис. 6.22. Определение класса `LaserClass` с использованием механизма инкапсуляции при записи на языках Java и C#



Основные положения для запоминания

- **Абстракция данных предоставляет средства отделения поведения от реализации.**

6.5. Вопросы и упражнения

1. В чем различие между объектом и классом?
2. Какие классы объектов, помимо `LaserClass`, могут использоваться в компьютерной игре, обсуждавшейся в этом разделе в качестве примера? Какие переменные экземпляра в дополнение к `RemainingPower` могут присутствовать в классе `LaserClass`?
3. Предположим, что классы `PartTimeEmployee` (*частично занятый работник*) и `FullTimeEmployee` (*работник на полной ставке*) наследуют свойства класса `Employee` (*работник*). Какие, по вашему мнению, определенные свойства и функции могут присутствовать в каждом из классов?
4. Что такое конструктор?
5. Почему в пределах класса некоторым его элементам присваивается модификатор `private`?

6.6. Программирование параллельных процессов

Предположим, что нам поручили разработать программу для анимации объектов в компьютерной игре, включающей множество атакующих вражеских космических кораблей. Один из подходов к решению этой задачи — создание единой программы, которая управляла бы всей анимацией на экране дисплея. В задачу такой программы входило бы рисование каждого космического корабля, что предполагает (если требуется достаточно реалистичная анимация), что программа должна непрерывно вычислять индивидуальные характеристики каждого из этих атакующих кораблей. Альтернативный подход состоит в разработке программы, управляющей анимацией отдельного космического корабля, характеристики которого определяются параметрами, задаваемыми при запуске этой программы. В этом случае анимацию в пределах всего экрана можно было бы создать посредством выполнения ста запусков этой программы, в каждом случае используя собственный набор параметров. Выполнив эти запуски одновременно, можно получить иллюзию множества отдельных космических кораблей, одновременно перемещающихся по экрану.

Одновременное выполнение нескольких запусков программы называется **параллельной обработкой**. Для действительно параллельной обработки данных необходимо несколько центральных процессоров, каждый из которых будет выполнять отдельную запущенную копию программы. Когда в наличии есть только один центральный процессор, для создания иллюзии параллельной обработки можно разрешить нескольким процессам совместно использовать время единственного процессора аналогично тому, как это реализуется в мультипрограммных системах (этот вопрос обсуждался в главе 3).

Многие современные компьютерные приложения легче реализовать в контексте параллельной обработки, чем традиционным способом, предусматривающим запуск единственной программы. В свою очередь, современные языки программирования имеют синтаксические конструкции, предназначенные для выражения семантических структур, необходимых при параллельной обработке данных. Разработка таких языков требует выявления подобных синтаксических структур и определения синтаксиса для их выражения.

Каждый из языков программирования стремится реализовать парадигму параллельной обработки со своей точки зрения, выраженной с помощью собственной терминологии. Например, то, что мы неформально называем запуском, в языке Ada именуется *задачей* (task), а в языке Java — *потоком* (thread). Таким образом, в программе на языке Ada одновременные действия осуществляются путем создания нескольких *задач*, тогда как программа на языке Java создает несколько *потоков*. В любом случае результатом является порождение нескольких процессов, которые выполняются так же, как и процессы под

управлением многозадачной операционной системы. Поэтому далее мы договоримся ссылаться и на запущенную программу, и на задачу, и на поток как на *процесс*.

Возможно, наиболее важным действием, которое потребуется описывать в программе, выполняющей параллельную обработку данных, является создание новых процессов. Если мы хотим осуществить запуск многих программ анимации движения космического корабля одновременно, нам потребуется синтаксическая конструкция, позволяющая выполнить эти действия. Запуск новых процессов очень напоминает традиционный вызов функции на выполнение. Разница лишь в том, что при традиционном подходе вызывающая функцию программная единица приостанавливает свою работу, пока вызываемая функция не закончит свое выполнение (вспомните рис. 6.8), тогда как в контексте параллельной обработки вызывающая программная единица продолжает свою работу одновременно с вызванной ею функцией, как показано на рис. 6.23. Таким образом, чтобы воспроизвести на экране движение множества космических кораблей одновременно, нужно написать основную программу, которая просто осуществляет множество запусков программы, имитирующей движение космического корабля; причем каждый запуск выполняется со своим набором параметров, описывающим отличительные характеристики каждого из кораблей.

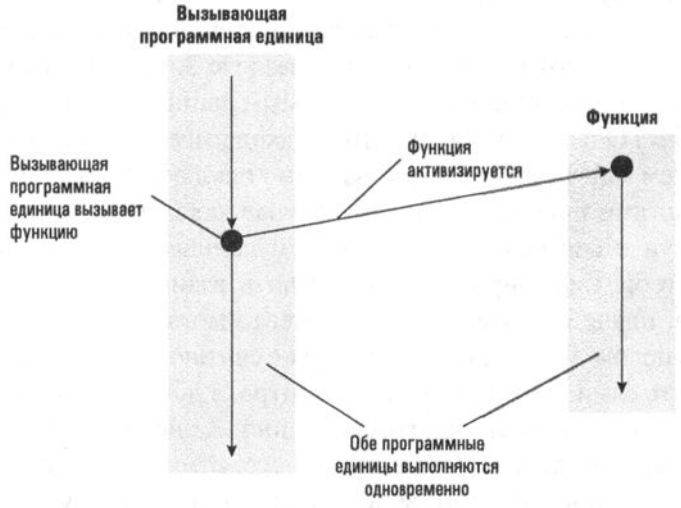


Рис. 6.23. Создание параллельных потоков

Более сложный вопрос связан с параллельной обработкой данных, которая предусматривает обмен данными между процессами. Например, в нашей космической игре вычислительные процессы, соответствующие отдельным космическим кораблям, должны согласовывать между собой их взаимное расположение, что необходимо для координации их действий. В иных случаях один процесс

должен ожидать, пока другой не достигнет определенной точки в своих вычислениях, или одному потоку может потребоваться остановить другой, пока он не выполнит определенную поставленную перед ним задачу.

Необходимые виды взаимодействия процессов долгое время тщательно изучались специалистами по компьютерным наукам, и многие современные языки программирования отражают различные подходы к проблеме организации взаимосвязи между процессами. В качестве примера рассмотрим проблему взаимосвязи, возникающую в том случае, когда два процесса обрабатывают один и тот же набор данных. (Этот пример детально освещается в разделе 3.4.) В частности, если каждый из двух параллельных процессов должен добавить число 3 к общему элементу данных, требуется найти метод, гарантирующий, что один процесс сможет беспрепятственно завершить начатую им операцию, прежде чем другой получит право на выполнение своей задачи. В противном случае оба процесса могут начать собственную обработку с одного и того же исходного значения, и в результате исходное значение будет увеличено только на три, а не на шесть. О данных, которые в каждый момент времени могут быть доступны только одному процессу, говорят, что они нуждаются во **взаимно исключающем доступе**.

Один из способов реализации взаимно исключающего доступа заключается в написании программных элементов, которые описывают задействованные потоки таким образом, что, когда поток использует общие данные, он блокирует другие потоки от доступа к этим данным до тех пор, пока такой доступ не станет безопасным. (Этот подход описан в разделе 3.4, где та часть процесса, которая имеет доступ к совместно используемым данным, была названа *критической областью*.) Опыт показал, что этот подход имеет недостаток, связанный с распределением задачи организации взаимно исключающего доступа между различными частями программы. В этом случае каждый элемент программы, имеющий доступ к совместно используемым данным, должен быть спроектирован так, чтобы гарантированно обеспечить взаимоисключаемый подход к этим данным, иначе ошибка в одном сегменте может вызвать повреждение всей системы в целом. По этой причине многие считают, что лучшим решением будет объединить сами данные и средства контроля доступа к ним в единое целое. Проще говоря, в этом случае ответственность за исключение множественного доступа к данным налагается не на процесс, который стремится получить доступ, а на сами данные, к которым необходим доступ. В результате контроль над доступом к данным сосредоточивается в одном месте программы, а не распыляется среди множества ее элементов. Элемент данных, дополненный способностью контролировать доступ к самому себе, часто называют **монитором**.

Таким образом, проектирование языков программирования для параллельной обработки данных включает разработку способов выражения таких действий, как активизация процесса, приостановка и повторный запуск процесса, идентификация критических областей и создание мониторов.

Создание программного обеспечения для смартфонов

Программное обеспечение для портативных, мобильных и встроенных устройств часто разрабатывается с использованием тех же языков программирования общего назначения, которые используются в других контекстах. При наличии достаточно большой клавиатуры и необходимого терпения некоторые приложения для смартфонов могут быть написаны непосредственно на самом смартфоне. Однако в большинстве случаев программное обеспечение для смартфонов разрабатывается на настольных компьютерах с использованием специальных программных систем, которые предоставляют разработчикам инструменты для редактирования, трансляции и тестирования программного обеспечения для смартфонов. Простые приложения часто пишутся на языках Java, Swift, C++ и C#. Однако для написания более сложных приложений или компонентов программного обеспечения самой операционной системы дополнительно может потребоваться поддержка параллельного и управляемого событиями программирования.

В заключение следует отметить, что хотя анимация и представляет собой интересный пример для исследования различных аспектов проблемы параллельных вычислений, это всего лишь одна из многих областей, которые нуждаются в организации параллельной обработки данных. К таким областям следует отнести составление прогнозов погоды, управление потоками авиаперевозок, моделирование сложных систем (от ядерных реакторов до пешеходных потоков), организацию работы компьютерных сетей и сопровождение баз данных.

6.6. Вопросы и упражнения

1. Какие свойства характерны для языков программирования, поддерживающих параллельную обработку данных, но отсутствуют в традиционных языках программирования?
2. Опишите два метода организации взаимно исключающего доступа к данным.
3. Укажите несколько областей, отличных от анимации, в которых использование параллельных вычислений дает существенные преимущества.

6.7. Декларативное программирование

В разделе 6.1 утверждалось, что формальная логика позволяет создать общий алгоритм решения задач, на основе которого можно построить систему декларативного программирования. В данном разделе мы исследуем это утверждение, создав сначала некоторый упрощенный алгоритм, а затем кратко обсудив возможности основанного на нем языка декларативного программирования.

Логический вывод

Предположим, мы знаем, что лягушонок Кермит либо играет в спектакле, либо болен, и нам сказали, что сейчас лягушонок Кермит не играет в спектакле. Тогда мы можем сделать вывод, что лягушонок Кермит, должно быть, болен. Это — пример дедуктивного рассуждения, которое называется **резолюцией**. Резолюция является одним из многих методов, называемых **правилами вывода** и предназначенных для получения следствия из набора утверждений.

Чтобы лучше понять этот принцип, примем сначала соглашение представлять простые высказывания отдельными буквами, а отрицание высказывания — символом ¬. Например, мы можем представить высказывание "Кермит — принц" буквой A, а высказывание "Мисс Пигги — актриса" — буквой B. Тогда выражение

```
A OR B
```

будет означать "Кермит — принц или мисс Пигги — актриса", а выражение

```
B AND ¬A
```

будет означать "Мисс Пигги — актриса, а Кермит — не принц". Для обозначения отношения "следует" мы будем использовать стрелку. Например, выражение

```
A→B
```

означает "Если Кермит — принц, то мисс Пигги — актриса".

В общем виде принцип резолюции утверждает, что из двух высказываний вида

```
P OR Q
```

и

```
R OR ¬Q
```

мы можем вывести высказывание

```
P OR R
```

В этом случае мы говорим, что два исходных высказывания сводятся к третьему, которое называется **резольвентой**. Важно подчеркнуть, что резольвента — это логическое следствие исходных высказываний. Иначе говоря, если исходные высказывания истинны, то резольвента также должна быть истинной. (Если Q — истинно, то R должно быть истинным, но если Q — ложно, то P должно быть истинным. Следовательно, независимо от того, является Q истинным или ложным, либо P, либо R должно быть истинным.)

Графически мы будем представлять резолюцию двух высказываний так, как показано на рис. 6.24, на котором мы написали исходные высказывания и изобразили линии, соединяющие их со своей резольвентой. Заметим, что резолюцию можно применять только к парам высказываний, которые выражаются в **форме предложения**, т.е. к высказываниям, элементарные компоненты которых можно соединять булевой операцией OR (ИЛИ). Таким образом, высказывание

```
P OR Q
```

выражено в форме предложения, в то время как для высказывания

```
P→Q
```

это не так. Эта потенциальная проблема не очень серьезна, поскольку в математической логике существует теорема, утверждающая, что любое высказывание, записанное средствами логики предикатов первого порядка (системы для представления высказываний, имеющей очень большие выразительные возможности), можно сформулировать в форме предложения. Мы не будем обсуждать здесь эту важную теорему, но для дальнейших ссылок заметим, что высказывание

```
P→Q
```

эквивалентно высказыванию в форме предложения

```
Q OR ¬P
```

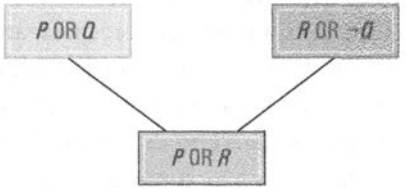


Рис. 6.24. Резолюция высказываний (P OR Q) и (R OR ¬Q) с получением высказывания (P OR R)

Совокупность высказываний является **противоречивой**, если все они не могут быть истинными одновременно. Другими словами, такая совокупность

высказываний состоит из противоречащих одно другому высказываний. Простой пример такой совокупности — объединение высказывания P и его отрицания ¬P. Специалисты-логики доказали, что повторное применение резолюции — это систематический метод проверки непротиворечивости множества предложений. Если повторный метод резолюции приводит к пустому предложению (результату резолюции предложения P с предложением ¬P), то исходная совокупность высказываний является противоречивой. Например, на рис. 6.25 показано, что совокупность высказываний

```
P OR Q
R OR ¬Q
¬R
¬P
```

является противоречивой.



Рис. 6.25. Резолюция высказываний (P OR Q), (R OR ¬Q), ¬R и ¬P

Предположим, нам необходимо проверить, что из некоторой совокупности высказываний действительно следует высказывание P. Учитывая, что высказывание P эквивалентно отрицанию высказывания ¬P, чтобы показать, что из исходной совокупности высказываний следует высказывание P, нам нужно лишь применять процедуру резолюции к исходным высказываниям плюс высказывание ¬P до тех пор, пока мы не получим пустое предложение. Получив пустое предложение, мы можем прийти к выводу, что высказывание ¬P противоречит исходным высказываниям, а значит, из исходных высказываний следует высказывание P.

Остается решить всего одну последнюю проблему, и мы будем готовы к применению процедуры резолюции в реальной программной среде. Предположим, что имеются два высказывания:

```
(Маша находится в X) → (Машин ягненок находится в X)
```

где X означает произвольное местоположение и еще одно высказывание

(Маша находится дома).

В форме предложения эти высказывания принимают следующий вид:

(Машин ягненок находится в X) OR ¬(Маша находится в X)

и

(Маша находится дома).

На первый взгляд, может показаться, что эти предложения не имеют компонентов, которые можно подвергнуть резолюции. Однако, с другой стороны, компоненты (Маша находится дома) и ¬(Маша находится в X) почти противоречат один другому. Задача заключается в том, чтобы установить, что высказывание (Маша находится в X), которое является высказыванием о местонахождении вообще, является высказыванием и о *доме* в частности. Таким образом, частный случай первого высказывания имеет вид

(Машин ягненок находится дома) OR ¬(Маша находится дома)

который может быть сведен путем резолюции с высказыванием

(Маша находится дома)

к высказыванию вида

(Машин ягненок находится дома)

Процесс присваивания значений переменным (например, присваивание значения "дома" переменной X), который делает возможным выполнение резолюции, называется **унификацией**. Это тот процесс, который позволяет применять общие высказывания к частным приложениям в дедуктивной системе.

Язык Prolog

Язык Prolog (*PROgramming in LOGic* — программирование в логике) — это декларативный язык программирования, базовый алгоритм решения задач которого основан на методе повторной резолюции. Такие языки называют языками **логического программирования**. Программа на языке Prolog состоит из совокупности исходных высказываний, с помощью которых базовый алгоритм выполняет свои дедуктивные рассуждения. Компоненты, из которых состоят эти высказывания, называются **предикатами**. Предикат состоит из идентификатора предиката, за которым в скобках следуют аргументы предиката. Отдельный предикат представляет собой некоторый факт в отношении его аргументов, а идентификатор предиката обычно выбирается так, чтобы отразить его семантику. Таким образом, если мы хотим выразить тот факт, что Билл — отец Мери, мы можем использовать следующую предикатную форму:

```
parent(bill, mary)
```

Обратите внимание, что аргументы в этом предикате начинаются со строчных букв, даже если речь идет об именах собственных. Это происходит потому, что язык Prolog различает аргументы, которые являются константами, и аргументы, которые являются переменными, выдвигая требование, чтобы константы начинались со строчных букв, а переменные — с прописных. Следует отметить, что здесь мы использовали терминологию сообщества пользователей языка Prolog, в котором термин *константа* используется вместо более общего термина *литерал*. Точнее, терм `bill` (обратите внимание на первую строчную букву) используется в языке Prolog для представления литерала, который может быть представлен как "`Bill`" в более общей записи. В свою очередь, терм `Bill` (обратите внимание на прописную первую букву) используется в языке Prolog для обозначения переменной.

Операторы в языке Prolog являются либо фактами, либо правилами, и каждый из операторов заканчивается точкой. Факт состоит из отдельного предиката. Например, тот факт, что черепаха (`turtle`) двигается быстрее улитки (`snail`), может быть представлен оператором языка Prolog следующего вида:

```
faster(turtle, snail).
```

В свою очередь, тот факт, что кролик (`rabbit`) бегает быстрее черепахи, выражается оператором

```
faster(rabbit, turtle).
```

Правило в языке Prolog — это оператор импликации (следования). Однако вместо записи оператора в виде `X → Y` программист на языке Prolog пишет "`Y`, если `X`", используя вместо слова "если" символы `:-` (двоеточие, за которым следует дефис). Таким образом, правило "`X` стар (`old`), следовательно `X` мудр (`wise`)", которое математик-логик может выразить в виде

```
old(X) → wise(X)
```

на языке Prolog будет выражено следующим образом:

```
wise(X) :- old(X).
```

Вот еще один пример. Правило

```
(faster(X, Y) AND faster(Y, Z)) → faster(X, Z)
```

на языке Prolog может быть выражено следующим образом:

```
faster(X, Z) :- faster(X, Y), faster(Y, Z).
```

Запятая, разделяющая термы `faster(X, Y)` и `faster(X, Y)`, представляет здесь оператор конъюнкции `AND` (И). Такие правила легко могут быть преобразованы программным обеспечением языка Prolog в форму предложения.

Учтите, что система языка Prolog ничего не знает о значении предикатов в программе, она просто манипулирует высказываниями, формально применяя правило резолюции. Таким образом, описание всех относящихся к делу свойств предикатов в терминах фактов и правил входит в обязанности программиста. В этом смысле факты в языке Prolog обычно используются для конкретизации примеров предиката, а правила — для описания общих принципов. Этому подходу соответствуют предыдущие операторы, относящиеся к предикату `faster`. Два факта описывают конкретные примеры свойства "двигаться быстрее", а правило — некое общее свойство. Заметим, что факт "кролик двигается быстрее улитки" хотя и не высказан явно, является следствием двух фактов, объединенных в соответствии с существующим правилом.

При создании программного обеспечения с использованием языка Prolog задача программиста — разработать совокупность фактов и правил, описывающих известную информацию. Эти факты и правила образуют множество исходных высказываний, используемых затем в дедуктивной системе. Установив такую совокупность высказываний, можно ввести (обычно с клавиатуры) предложение (в терминологии языка Prolog называемое *целью*), которое система должна проверить. Как только перед дедуктивной системой языка Prolog будет поставлена некоторая цель, она применит операцию резолюции, пытаясь найти подтверждение того, что указанная цель следует из исходных высказываний. С помощью нашего набора высказываний, описывающих отношение `faster`, приведенные ниже цели

```
faster(turtle, snail).
faster(rabbit, turtle).
faster(rabbit, snail).
```

будут подтверждены, поскольку каждое из них является логическим следствием исходных высказываний. Первые два факта идентичны фактам, приведенным в исходных высказываниях, а третий является результатом дедукции.

Более интересные результаты получаются, если в задачах используются не константы, а переменные. В этих случаях программа на языке Prolog пытается вывести цель из исходных высказываний, отслеживая требуемые унификации. Затем, если цель достигнута, программа указывает эти унификации. Например, рассмотрим цель

```
faster(W, snail).
```

В результате программа сообщает

```
faster(turtle, snail).
```

Действительно, это — следствие исходных высказываний, которое согласуется с поставленной целью с помощью унификации. Более того, если мы попросим

программу сообщить нам больше фактов, она найдет и выведет на печать следующее следствие:

```
faster(rabbit, snail).
```

И наоборот, мы можем попросить программу найти примеры животных, которые более медлительны, чем кролик, поставив программе цель

```
faster(rabbit, W).
```

Если же мы поставим задачу

```
faster(V, W).
```

то система Prolog в конце концов найдет все отношения `faster`, которые могут быть выведены из исходных высказываний. Таким образом, единственная программа на языке Prolog может быть использована для подтверждения, что одно конкретное животное быстрее другого; для поиска всех тех животных, которые быстрее указанного животного; для поиска животных, которые медленнее указанного животного; а также для поиска всех отношений "быстрее/медленнее" между животными.

Подобная гибкость просто захватывает воображение специалистов в области компьютерных наук. К сожалению, при реализации в системе Prolog процедура резолюции наследует ограничения, которые отсутствуют в ее теоретической форме, а это означает, что программы на языке Prolog могут оказаться неспособными проявить теоретически ожидаемую от них гибкость. Чтобы понять, что здесь имеется в виду, сначала обратите внимание на то, что диаграмма на рис. 6.25 отображает только те разрешения, которые имеют отношение к поставленной задаче. Однако есть и другие направления, которым мог бы следовать процесс резолюции. Например, крайние слева и крайние справа пункты могут быть разрешены для получения резольвенты `Q`. Таким образом, помимо утверждений, описывающих основные факты и правила, связанные с приложением, программа Prolog часто должна содержать дополнительные утверждения, назначение которых состоит лишь в том, чтобы направлять процесс решения в желаемую сторону. По этой причине реальные программы на языке Prolog могут в действительности не охватывать множество целей, предложенных в предыдущем примере.

6.7. Вопросы и упражнения

1. Какие из высказываний R, S, T, U и V являются логическим следствием совокупности высказываний (¬R OR T OR S), (¬S OR V), (¬V OR R), (U OR ¬S), (T OR U) и (S OR V)?
2. Является ли следующая совокупность высказываний непротиворечивой? Обоснуйте свой ответ.

```
P OR Q OR R
¬R OR Q
R OR ¬P
¬Q
```

3. Завершите два правила в конце приведенной ниже программы на языке Prolog таким образом, что предикат `mother(X, Y)` будет означать "X является матерью Y", а предикат `father(X, Y)` будет означать "X является отцом Y".

```
female(carol).
female(sue).
male(bill).
male(john).
parent(john, carol).
parent(sue, carol).
mother(X,Y) :-
father(X,Y) :-
```

4. В контексте программы на языке Prolog, приведенной в предыдущем упражнении, следующее правило должно означать, что X является родным братом или сестрой Y, если X и Y имеют общего родителя.

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

Какое неожиданное заключение может сделать система Prolog исходя из этого определения братской или сестринской связи?

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. Что означает высказывание "язык программирования является машинно-независимым"?
2. Переведите следующую программу с псевдокода на машинный язык Vole, описанный в приложении B.

```
x = 0
while (x < 3):
  x = x + 1
```

3. Переведите приведенный ниже оператор

```
Halfway = Length + Width
```

на машинный язык Vole, описанный в приложении B, предполагая, что переменные `Length`, `Width` и `Halfway` представлены как числа с плавающей точкой.

4. Переведите следующий оператор языка высокого уровня

```
if (X == 0):
  Z = Y + W
else:
  Z = Y + X
```

на машинный язык Vole, описанный в приложении B, предполагая, что числа `W`, `X`, `Y` и `Z` представлены в двоичном дополнительном коде и занимают в основной памяти один байт.

5. Для чего при трансляции оператора, приведенного в задании 4, необходимо указывать тип данных, связанный с используемыми в нем переменными? Почему многие языки высокого уровня требуют от программистов указывать тип каждой переменной в начале программы?
6. Назовите и опишите четыре существующие парадигмы программирования.
7. Предположим, что функция f получает два числа в качестве параметров и возвращает меньшее из них в качестве результата. Если переменные w, x, y и z представляют собой числа, то какой результат будет возвращен этой функцией при вычислении выражения $f(f(w, x), f(y, z))$?
8. Предположим, что f — это функция, возвращающая в качестве результата строку, в которой все символы из входной строки переставлены в обратном порядке, а g — это функция, осуществляющая конкатенацию двух входных строк. Если x — строка "abed", что вернет функция $g(f(x), x)$?

9. Предположим, что вы собираетесь написать объектно-ориентированную программу для ведения своих финансовых записей. Какие данные нужно хранить в объекте, представляющем ваш текущий счет в банке? На какие сообщения должен реагировать этот объект? Какие еще объекты следует использовать в программе?
10. Опишите отличия, существующие между машинным языком и языком ассемблера.
11. Разработайте язык ассемблера для машины Vole, описанной в приложении B.
12. Некий Джон Программер утверждает, что возможность объявления констант в программе является излишней, поскольку вместо них можно использовать переменные. Так, в нашем примере в разделе 6.2 можно описать переменную `AirportAlt`, а затем присвоить ей нужное значение в начале программы. Почему это решение хуже, чем вариант с использованием константы?
13. Опишите различия, существующие между операторами объявления и выполняемыми операторами.
14. Объясните разницу между литералом, константой и переменной.
15. **а.** Что такое приоритет оператора?

    б. Рассуждая о приоритетах операторов, какие их значения могут быть связаны с выражением $6 + 2 \times 3$?
16. Что такое структурное программирование?
17. Чем отличается смысл оператора, представленного двойным символом "=" в операторе

    ```
    if (X == 5):
       ...
    ```

    от смысла оператора, представленного одним символом "=" в операторе присваивания

    ```
    X = 2 + Y
    ```
18. Начертите блок-схему структуры, выраженной следующим оператором.

    ```
    for (int x = 2; x < 8; ++x)
    { . . . }
    ```
19. Преобразуйте следующий оператор `for` в эквивалентную последовательность операторов, в которой используется оператор `while` языка Python.

    ```
    for (int x = 2; x < 8; ++x)
    { . . . }
    ```

20. Если вы знакомы с нотной записью, проанализируйте принятый принцип записи нот с точки зрения языка программирования. Что здесь является управляющими структурами? Какой предусмотрен синтаксис для вставки комментариев? Какие музыкальные обозначения похожи на оператор `for`, представленный на рис. 6.7?

21. Начертите блок-схему структуры, выраженной следующим оператором.

```
switch (suit)
  {case 'clubs': bid(1);
    case 'diamonds': bid(2);
    case 'hearts': bid(3);
    case 'spades': bid(4);
  }
```

22. Перепишите следующий фрагмент программы, используя один оператор `case` вместо серии вложенных операторов `if-else`.

```
if (W == 5):
  Z = 7
else:
  if (W == 6):
    Y = 7
  else:
    if (W == 7):
      X = 7
```

23. Выразите приведенную ниже запутанную последовательность операторов с помощью единственного оператора `if-else`.

```
if X > 5 then goto 80
X = X + 1
goto 90
80  X = X + 2
90  stop
```

24. Кратко опишите основные управляющие структуры, присутствующие в императивных и объектно-ориентированных языках программирования, предназначенные для выполнения следующих действий.

а. Определение, какой оператор должен быть выполнен следующим.

б. Повторение некоторой последовательности операторов.

в. Изменение значения переменной.

25. Опишите различия между транслятором и интерпретатором.

26. Предположим, что переменная `X` в программе описана как переменная типа `integer`. Какая ошибка будет обнаружена транслятором при выполнении следующего оператора?

```
X = 2.5
```

27. Что означает выражение "язык программирования со строгой типизацией"?

28. Почему большой массив не всегда можно передать в вызываемую процедуру по значению?

29. Предположим, что в сценарии на языке Python функция `Modify` определена следующим образом:

```
def Modify (Y):
  Y = 7
  print(Y)
```

Если параметры ей передаются по значению, что будет напечатано при выполнении следующего фрагмента программы? А если параметры будут передаваться ей по ссылке?

```
X = 5
Modify(X)
print(X)
```

30. Предположим, что в сценарии на языке Python функция `Modify` определена следующим образом:

```
def Modify (Y):
  Y = 9
  print(X)
  print(Y)
```

Допустим также, что `X` — это глобальная переменная. Если параметры передаются функции `Modify` по значению, что будет напечатано при выполнении следующего фрагмента программы? Что будет напечатано, если параметры передаются по ссылке?

```
X = 5
Modify(X)
print(X)
```

31. Иногда фактический параметр передается в процедуру путем создания его копии, предназначенной для использования в процедуре (как если бы параметр передавался по значению), но после выполнения процедуры значение копии присваивается фактическому параметру перед тем, как будет продолжено выполнение вызывающей процедуры. В таких случаях говорят, что параметр передается по значению-результату. Что будет

напечатано фрагментом программы из задания 30, если параметры передаются функции `Modify` по значению-результату?

32. **а.** В чем заключаются преимущества передачи параметров по значению в сравнении с передачей их по ссылке?

б. В чем заключаются преимущества передачи параметров по ссылке в сравнении с передачей их по значению?

33. Какая неоднозначность кроется в следующем операторе?

```
X = 3 + 2 × 5
```

34. Предположим, что небольшая компания имеет пять сотрудников и планирует увеличить их число до шести. Также предположим, что одна из используемых компанией программ содержит следующие операторы присваивания.

```
DailySalary = TotalSal / 5;
AvgSalary   = TotalSal / 5;
DailySales  = TotalSales / 5;
AvgSales    = TotalSales / 5;
```

Как можно было бы упростить задачу обновления программы, если бы в ней исходно использовались константы с именами `NumberOfEmp` (количество работников) и `WorkWeek` (рабочая неделя), обе имеющие значение 5, благодаря чему те же операторы можно было бы записать следующим образом:

```
DailySalary = TotalSal / DaysWk;
AvgSalary   = TotalSal / NumEmpl;
DailySales  = TotalSales / DaysWk;
AvgSales    = TotalSales / NumEmpl;
```

35. **а.** В чем состоит различие между формальными языками и традиционными языками?

б. Приведите пример языка каждого типа.

36. Нарисуйте синтаксическую диаграмму, представляющую структуру оператора `while` языка Python, обсуждавшегося в главе 5.

37. Разработайте совокупность синтаксических диаграмм для описания синтаксиса телефонных номеров, написанных в формате (444) 555-1234, т.е. кода, состоящего из кода региона, кода района и четырехзначного индивидуального кода.

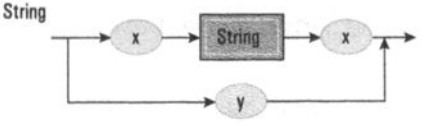
38. Разработайте совокупность синтаксических диаграмм для описания простого предложения на вашем родном языке.

39. Разработайте совокупность синтаксических диаграмм для описания различных способов представления дат, таких как *день/месяц/год* или *день месяц, год*.

40. Разработайте совокупность синтаксических диаграмм для описания грамматической структуры "предложений", которые состоят из нескольких вхождений слова *да*, за которым следует такое же количество вхождений слова *нет*. Например, последовательность "*да да нет нет*" будет являться таким предложением, а последовательности "*нет да*", "*да нет нет*" и "*да нет да*" — не будут.

41. Приведите аргументы в пользу того, что нельзя разработать набор синтаксических диаграмм, описывающих грамматическую структуру "предложений", состоящих из нескольких вхождений слова *да*, за которыми следует такое же количество вхождений слова *нет*, за которыми следует то же самое число вхождений слова *возможно*. Например, последовательности "*да нет возможно*" и "*да да нет нет возможно возможно*" будут такими "предложениями", тогда как последовательности "*да возможно*", "*да нет нет возможно возможно*" и "*возможно нет*" — не будут.

42. Напишите предложение, описывающее структуру строки, в соответствии с приведенной ниже синтаксической диаграммой. Затем нарисуйте дерево синтаксического анализа строки *xxyxx*.



43. Добавьте синтаксические диаграммы к диаграммам из вопроса 5 в разделе 6.4, чтобы получить совокупность диаграмм, определяющих структуру *Танец*, которая может быть структурой *Ча-ча-ча* или *Вальс*, где *Вальс* состоит из одной или более копий шаблона

вперед по_диагонали каданс
или
назад по_диагонали каданс.

44. Нарисуйте дерево синтаксического анализа для выражения `x * y + y / x`, используя синтаксические диаграммы, представленные на рис. 6.15.

45. Какую оптимизацию кода может выполнить генератор кода при создании машинного кода для следующего оператора?

```
if (X == 5):
  Z = X + 2
else:
  Z = X + 4
```

46. Упростите следующий фрагмент программы:

```
Y = 5
if (Y == 7):
  Z = 8
else:
  Z = 9
```

47. Упростите следующий фрагмент программы:

```
while (X != 5):
  X = 5
```

48. Что общего между типами данных и классами в среде объектно-ориентированного программирования? Чем эти понятия различаются?

49. Поясните, как механизм наследования может использоваться для разработки классов, описывающих различные типы строений?

50. В чем состоит различие между открытыми (`public`) и закрытыми (`private`) компонентами класса?

51. **а.** Приведите пример ситуации, в которой переменная экземпляра должна быть закрытой (private).

б. Приведите пример ситуации, в которой переменная экземпляра должна быть открытой (public).

в. Приведите пример ситуации, в которой метод должен быть закрытым (private).

г. Приведите пример ситуации, в которой метод должен быть открытым (public).

52. Опишите некоторые объекты, которые могут присутствовать в программе моделирования пешеходного движения в холле отеля. Включите в свой ответ описание тех действий, которые эти объекты должны уметь выполнять.

***53.** Что означает термин *монитор* в контексте языка программирования?

***54.** Какие свойства параллельной обработки делают желательным использование языков программирования с поддержкой многопоточности?

***55.** Нарисуйте диаграмму (подобную диаграмме, показанной на рис. 6.25), представляющую последовательность операций резолюции, необходимых для того, чтобы показать, что совокупность высказываний `(Q OR ¬R)`, `(T OR R)`, `¬P`, `(P OR ¬T)` и `(P OR ¬Q)` противоречива.

***56.** Является ли совокупность высказываний `¬R`, `(T OR R)`, `(P OR ¬Q)`, `(Q OR ¬T)` и `(R OR ¬P)` противоречивой? Обоснуйте свой ответ.

***57.** Дополните программу на языке Prolog, приведенную в упражнениях 3 и 4 раздела 6.7, таким образом, чтобы включить в нее поддержку дополнительных вариантов родственных связей, таких как *дядя*, *тетя*, *бабушка*, *дедушка*, *двоюродный брат* и *двоюродная сестра*. Также добавьте в нее правило, которое определяет `parents (X, Y, Z)` в том смысле, что `X` и `Y` являются родителями `Z`.

***58.** Полагая, что первый оператор в приведенной ниже программе на языке Prolog должен означать "Алиса любит спорт", переведите последние два оператора этой программы. Затем перечислите все, что, исходя из этой программы, интерпретатор языка Prolog смог бы выявить как то, что Алиса любит. Поясните свой ответ.

```
likes(alice, sports).
likes(alice, music).
likes(carol, music).
likes(david, X) :- likes(X, sports).
likes(alice, X) :- likes(david, X).
```

***59.** Какие проблемы могут возникнуть при выполнении следующего фрагмента программы машиной, в которой числа хранятся в 8-битовом формате с плавающей точкой, описанном в разделе 1.7?

```
X = 0.01
while (X != 1.00):
  print(X)
  X = X + 0.01
```

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также разобраться, почему вы ответили именно так, а не иначе и насколько ваши суждения по различным вопросам согласуются между собой.

1. В общем случае законы об авторском праве допускают присвоение права собственности на способ выражения идеи, но не на саму идею. В результате на текст параграфа в книге авторские права распространяются, а на изложенную в этом параграфе идею — нет. Как распространить это право на исходные тексты программ и алгоритмы, которые они выражают?

В какой степени можно разрешать людям, которые знают алгоритмы, лежащие в основе коммерческих программ, писать собственные программы, выражающие те же алгоритмы, и продавать эти версии программного обеспечения на рынке?

2. Используя язык программирования высокого уровня, программист может выразить алгоритм, используя такие слова, как *если* (`if`), *иначе* (`else`) и *пока* (`while`). В каких пределах компьютер понимает значения этих слов? Означает ли способность правильно реагировать на использование слов понимание их значения? Как узнать, что другой человек понял то, что вы сказали?
3. Должен ли человек, который разработал новый и полезный язык программирования, иметь право на доход от использования этого языка? Если да, то как можно защитить это право? До какой степени язык программирования может быть чьей-то собственностью? В какой степени компания может обладать правом владения на творческие, интеллектуальные достижения ее сотрудников?
4. В случае угрозы срыва установленных сроков разработки считаете ли вы приемлемым для программиста отказаться от документирования программы с помощью комментариев, если это позволит ему получить работающую программу вовремя? (Студенты часто удивляются, насколько серьезно профессиональные разработчики программного обеспечения относятся к программной документации.)
5. Множество исследований в области языков программирования посвящено языкам, позволяющим программистам писать программы, которые удобочитаемы и понятны человеку. Насколько программист нуждается в таких возможностях? Достаточно ли, чтобы программа работала правильно, будучи даже не очень хорошо записанной с точки зрения человека?
6. Предположим, что программист-любитель пишет программу для себя и при этом относится к ее разработке достаточно небрежно. В результате в программе не используются возможности языка программирования, способные сделать ее более читабельной; программа работает неэффективно; текст содержит упрощения и специфические приемы, ориентированные на данный конкретный случай, для которого программист и написал свою программу. Со временем программист делает копии этой программы для своих друзей, которые хотят использовать ее в собственных целях, а эти, в свою очередь, передают ее своим знакомым. Какую ответственность несет программист за проблемы с этой программой, которые могут возникнуть у ее пользователей?

7. В какой степени профессионал в компьютерных науках должен разбираться в различных парадигмах программирования? Некоторые компании настаивают, чтобы все программное обеспечение, разрабатываемое их сотрудниками, было написано на одном и том же заранее определенном языке программирования. Изменится ли ваш первоначальный ответ, если профессионал работает в такой компании?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Aho A.V., Lam M. S., Sethi R., Ullman J.D. *Compilers: Principles, Techniques, and Tools*, 2nd ed. — Boston, MA: Addison-Wesley, 2007. (Имеется русский перевод этой книги: Ахо А.В., Лам М.С., Рави С., Ульман Дж.Д. *Компиляторы: принципы, технологии и инструментарий*, 2-е изд.: — М.: Издательский дом "Вильямс", 2007.)
2. Barnes J. *Programming in Ada 2005*. — Boston, MA: Addison-Wesley, 2006.
3. Clocksin W.F., Mellish C.S. *Programming in Prolog*, 5th ed. — New York: Springer-Verlag, 2013.
4. Friedman D.P., Felleisen M. *The Little Schemer*, 4th ed. — Cambridge, MA: MIT Press, 1995.
5. Hamburger H., Richards D. *Logic and Language Models for Computer Science*. — Upper Saddle River, NJ: Prentice-Hall, 2002.
6. Kernighan B.W., Ritchie D.M. *The C Programming Language*, 2nd ed. — Englewood Cliffs, NJ: Prentice Hall, 1988.
7. Metcalf M., Reid J. *Fortran 90/95 Explained*, 2nd ed. — Oxford, England: Oxford University Press, 1999.
8. Pratt T.W., Zelkowitz M.V. *Programming Languages, Design and Implementation*, 4th ed. — Upper Saddle River, NJ: Prentice-Hall, 2001.
9. Savitch W., Mock K. *Absolute C++*, 5th ed. — Boston, MA: Addison-Wesley, 2012.
10. Savitch W., Mock K. *Absolute Java*, — 5th ed. Boston, MA: Addison-Wesley, 2012.
11. Savitch W. *Problem Solving with C++*, 8th ed. Boston, MA: Addison-Wesley, 2011. (Имеется русский перевод 3-го издания этой книги: Сэвитч У. *Язык C++. Курс объектно-ориентированного программирования*, 3-е изд. — М.: Издательский дом "Вильямс", 2001).
12. Scott M.L. *Programming Language Pragmatics*, 3rd ed. — New York: Morgan Kaufmann, 2009.
13. Sebesta R.W. *Concepts of Programming Languages*, 10th ed. — Boston, MA: AddisonWesley, 2012. (Имеется русский перевод 5-го издания этой книги:

Себеста Р.В. *Основные концепции языков программирования*, 5-е изд. — М.: Издательский дом "Вильямс", 2002).

14. Wu C.T. *An Introduction to Object-Oriented Programming with Java*, 5th ed. — Burr Ridge, IL: McGraw-Hill, 2009.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Мартелли А., Рейвенскрофт А., Холден С. *Python. Справочник. Полное описание языка*, 3-е изд. — СПб.: ООО "Диалектика", 2018.
2. Прата С. *Язык программирования С. Лекции и упражнения*, 6-е изд. — М.: Издательский дом "Вильямс", 2015.
3. Кейденхед Р. *Java за 24 часа*, 8-е изд. — СПб.: ООО "Диалектика", 2019.
4. Готтшлинг П. *Современный C++ для программистов, инженеров и ученых*. — М.: Издательский дом "Вильямс", 2016.
5. Троелсен Э., Джепикс Ф. *Язык программирования C# 6.0 и платформа .NET 4.6*, 7-е изд. — М.: Издательский дом "Вильямс", 2016.
6. Грэхем И. *Объектно-ориентированные методы. Принципы и практика.* 3-е изд. — М.: Издательский дом "Вильямс", 2004.

Глава 7

В этой главе мы обсудим проблемы, возникающие в процессе разработки крупных и сложных систем программного обеспечения. Эту область компьютерных наук называют *технологией разработки программного обеспечения*, поскольку создание ПО, безусловно, следует понимать как процесс технический. Цель исследований, проводимых в этой области компьютерных наук, — найти принципы, определяющие ход процесса разработки программного обеспечения и способные обеспечить создание эффективных и надежных программных продуктов.

Технология разработки программного обеспечения

7.1. ПРЕДМЕТ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.2. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ
- Жизненный цикл в целом
- Традиционные этапы разработки программ

7.3. МЕТОДОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.4. МОДУЛЬНОСТЬ
- Реализация модулей
- Связанность модулей
- Связность элементов модуля
- Сокрытие информации
- Компоненты

7.5. ИНСТРУМЕНТЫ И МЕТОДЫ ПРОЕКТИРОВАНИЯ
- Традиционные инструменты разработки
- UML — унифицированный язык моделирования
- Шаблоны проектирования

7.6. ОБЕСПЕЧЕНИЕ КАЧЕСТВА ПРОГРАММ
- Область применения средств обеспечения качества ПО
- Тестирование программного обеспечения

7.7. ДОКУМЕНТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.8. ИНТЕРФЕЙС "ЧЕЛОВЕК–МАШИНА"

7.9. ПРАВО СОБСТВЕННОСТИ И ОТВЕТСТВЕННОСТЬ ЗА СОЗДАВАЕМОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Технология программного обеспечения — это отрасль компьютерных наук, которая ищет основные принципы, определяющие успешную разработку больших и сложных программных систем. Проблемы, с которыми приходится сталкиваться при разработке таких систем, — это нечто большее, чем просто расширенные версии задач, возникающих при написании небольших программ. В частности, разработка подобных систем, как правило, требует усилий многих людей на протяжении длительного времени, причем за этот период могут меняться как требования к создаваемой системе, так и состав персонала, занимающегося ее разработкой. Поэтому предмет технологии разработки программного обеспечения включает и такие аспекты, как управление проектом и персоналом, которые обычно ассоциируются с менеджментом, а не с областью компьютерных наук. Однако в этой главе мы сконцентрируем наше внимание только на вопросах, относящихся к компьютерным наукам.

7.1. Предмет технологии разработки программного обеспечения

Чтобы иметь представление о проблемах, возникающих при проектировании программного обеспечения, рассмотрим любой достаточно большой и сложный объект (автомобиль, многоэтажное офисное здание или даже кафедральный собор), а затем представим себе, что в наши обязанности входит как разработка его проекта, так и управление процессом его сооружения. Как оценить время, деньги и другие ресурсы, которые потребуются для выполнения проекта? Как разделить проект на управляемые части? Как убедиться, что созданные части совместимы? Как обеспечить взаимодействие сотрудников, работающих над разными частями проекта? Как оценить скорость выполнения работы? Как справиться с обилием всевозможных деталей (выбором дверных ручек, проектированием скульптурных украшений, поисками голубого стекла для витражей, расчетом прочности колонн, планированием работ по монтажу системы отопления)? На подобные вопросы приходится отвечать и в процессе разработки крупной системы программного обеспечения.

Поскольку проектирование является хорошо разработанной областью, можно полагать, что уже существует множество разработанных методов проектирования, предоставляющих ответы на такие вопросы. Частично эти рассуждения верны, однако в них не учитываются фундаментальные различия в свойствах программного обеспечения и объектов проектирования из других областей науки и техники. Эти различия существенно усложняют реализацию проектов по разработке программного обеспечения, что ведет к перерасходу средств, задержкам в поставках готовых продуктов и неудовлетворенности клиентов.

В свою очередь, выявление этих различий и стало первым шагом в развитии дисциплины технологии разработки программного обеспечения.

Одно из этих различий связано с возможностью создания новой системы из предварительно изготовленных компонентов общего назначения. Уже многие годы при создании сложных объектов в традиционных областях проектирования значительные преимущества достигаются просто за счет использования в качестве строительных блоков некоторых готовых компонентов. Конструктору автомобиля не нужно конструировать для него новый мотор или трансмиссию, — вместо этого он может воспользоваться уже созданными версиями этих компонентов. Однако в области разработки программного обеспечения в этом отношении имеется значительное отставание. В прежние времена ранее разработанные компоненты, как правило, не являлись универсальными, т.е. их внутренняя конструкция зависела от характерных особенностей конкретного приложения. А это означало, что возможности использования общих компонентов были весьма ограничены, — попытка повторно использовать подобные компоненты неизбежно требовала их перепроектирования. В результате исторически сложилось так, что достаточно сложные системы программного обеспечения чаще всего создавались практически "с нуля". Как мы увидим далее в этой главе, в настоящее время в этом отношении уже достигнут значительный прогресс, хотя многое все еще предстоит сделать.

Другое различие между разработкой программного обеспечения и традиционным инженерным проектированием связано с отсутствием количественных методов, называемых **метриками**, которые можно было бы использовать для измерения характеристик программного обеспечения. Например, чтобы спрогнозировать стоимость разработки программной системы, хотелось бы оценить сложность предлагаемого продукта, но методы измерения "сложности" программного обеспечения уклончивы. Аналогичным образом оценка качества программного продукта является сложной задачей. В случае механических устройств важной мерой качества является среднее время между отказами, которое, по существу, является показателем того, насколько хорошо устройство выдерживает износ. В противоположность этому программное обеспечение не изнашивается, поэтому данный метод измерения качества нельзя применить в разработке программного обеспечения.

Трудности, связанные с количественным измерением свойств программного обеспечения, являются одной из причин того, что в технологии разработки программного обеспечения предпринимаются настойчивые попытки найти строгую основу в том же смысле, какой имеет место в механике и электротехнике. В то время как эти последние области инженерии стоят на фундаменте устоявшейся науки физики, в технологии разработки программного обеспечения поиск своих корней все еще продолжается.

Поэтому в настоящее время исследования в области технологии разработки программного обеспечения разворачиваются на двух уровнях. Одни исследователи, которых иногда называют *практиками*, работают над развитием методов разработки, предназначенных для немедленного применения, тогда как другие исследователи, которых называют *теоретиками*, ведут поиск основополагающих принципов и теорий, на основании которых впоследствии можно будет разработать более надежные технологические методы. Основанные на субъективных представлениях, многие разработанные (и применяемые) в прошлом практиками методы были заменены другими подходами, которые со временем также могут устареть. Между тем успехи теоретиков остаются весьма относительными.

Прогресс в исследованиях как практиков, так и теоретиков имеет очень большое значение. Жизнь нашего общества уже невозможна без использования компьютерных систем и связанного с ними программного обеспечения. Экономика, здравоохранение, государственные учреждения, законодательство, транспорт и оборона любого современного государства зависят от больших систем программного обеспечения. Тем не менее надежность этих систем по-прежнему остается большой проблемой. Ошибки в программном обеспечении уже приводили к таким катастрофическим (или почти катастрофическим) последствиям, как принятие растущей луны за ядерную атаку, потеря 5 миллионов долларов банком Нью-Йорка только за один день, утрата проб космического пространства, собранных космической станцией, радиационное облучение людей, вызвавшее их смерть или паралич, и даже одновременное нарушение работы линий телефонной связи в обширных географических регионах.

Тем не менее не следует считать, что ситуация совсем уж мрачная. В действительности уже достигнут большой прогресс в преодолении таких проблем, как отсутствие готовых компонентов и метрик. Более того, применение компьютерных технологий в процессе разработки программного обеспечения привело к появлению того, что сейчас называют **средствами автоматизации разработки программ** (*CASE* — Computer-Aided Software Engineering). Этот подход позволяет упростить и рационализировать весь процесс разработки программного обеспечения. Появление концепции CASE привело к разработке разнообразных компьютеризированных систем, известных как **CASE-инструменты**, включающих *системы планирования проектов* (предназначены для оказания помощи в проведении оценки затрат, составлении графика проекта и распределении персонала), *системы управления проектами* (используются для организации мониторинга хода выполнения проекта), *средства документирования* (предназначены для оказания помощи в написании и организации документации), *системы прототипирования и моделирования* (используются в разработке прототипов), *системы проектирования интерфейсов* (предназначены

для ускорения разработки графических интерфейсов) и *системы программирования* (используются для организации и упрощения процедур написания и отладки программ). Одни из этих инструментов представляют собой нечто чуть большее, чем обычные текстовые процессоры, программное обеспечение для работы с электронными таблицами или системы связи по электронной почте, которые изначально разрабатывались для общего использования, а затем были адаптированы программистами для своих нужд. Другие представляют собой довольно сложные пакеты, разработанные, в первую очередь, для среды разработки программного обеспечения. И действительно, системы, известные как **интегрированная среда разработки** (*IDE* — Integrated Development Environments), объединяют в себе инструменты для разработки программного обеспечения (редакторы, компиляторы, средства отладки и т.д.) в единый интегрированный пакет. Ярким примером таких систем являются системы для разработки приложений для смартфонов. Они не только включают все инструменты программирования, необходимые для написания и отладки программного обеспечения смартфона, но также предоставляют программисту программы-эмуляторы, которые позволяют ему увидеть на дисплее обычного персонального компьютера то, как разрабатываемое им программное обеспечение будет работать и выглядеть непосредственно на экране смартфона.

В дополнение к прилагаемым исследователями усилиям профессиональные организации и организации, отвечающие за стандартизацию, включая ISO, Ассоциацию по вычислительной технике (*ACM* — Association for Computing Machinery) и Институт инженеров электротехники и электроники (*IEEE* — Institute of Electrical and Electronics Engineers), также вступили в борьбу за улучшение состояния дел в отрасли разработки программного обеспечения. Их усилия варьируются от принятия кодексов профессионального поведения и этики, направленных на повышение профессионализма разработчиков программного обеспечения и противодействие небрежному отношению к обязанностям отдельных исполнителей, до установления стандартов по измерению качества организаций, занимающихся разработкой программного обеспечения, а также предоставления рекомендаций, призванных помочь этим организациям улучшить свои позиции.

Одна из важнейших проблем, охватывающих все аспекты разработки больших программных систем, заключается в том, что объем необходимых усилий обязательно предполагает участие в работе многих людей. Хотя небольшие системы вполне могут быть спроектированы одним целеустремленным человеком, сама природа процесса разработки программного обеспечения в крупных проектах требует, чтобы его многочисленные участники имели возможность общаться и эффективно сотрудничать. Поэтапная совместная работа над большим программным проектом предполагает совсем иной набор навыков в

сравнении с написанием кода в индивидуальном порядке, но одновременно позволяет снизить сложность задач, стоящих перед каждым отдельным исполнителем. В действительности именно потому, что большие программные системы являются настолько сложными, возникает необходимость разбить всю систему на более мелкие компоненты — такие, чтобы исполнители-люди смогли полностью понять возложенную на них часть общей работы.

Ассоциация по вычислительной технике

Основанная в 1947 году, Ассоциация по вычислительной технике (*ACM* — Association for Computing machinery) является международной научной и образовательной организацией, задача которой — распространение навыков, теорий и приложений из области информационных технологий. Штаб-квартира этой организации, расположенная в Нью-Йорке, руководит деятельностью множества групп по различным направлениям (*SIG* — Special Interest Group), занимающихся такими проблемами, как архитектура компьютеров, искусственный интеллект, применение компьютеров в биомедицинских исследованиях, компьютеры и общество, обучение в области компьютерных наук, компьютерная графика, гипертекст/гипермедиа, операционные системы, языки программирования, имитация и моделирование, а также разработка программного обеспечения.

Веб-сайт этой ассоциации находится по адресу `http://www.acm.org`. Разработанный в этой ассоциации Кодекс этики и профессионального поведения можно найти по адресу `http://ethics.acm.org/code-of-ethics`.



Основные положения для запоминания

- **Сотрудничество позволяет сократить размер и сложность задачи, возложенной на отдельного программиста.**
- **Сотрудничество в итеративной разработке программного обеспечения требует иного набора умений в сравнении с созданием программы в одиночку.**
- **При разработке программного обеспечения эффективное общение между отдельными участниками является необходимым условием их успешного сотрудничества.**

В остальной части этой главы будут обсуждаться как определенные фундаментальные принципы технологии разработки программного обеспечения (такие, как жизненный цикл программного обеспечения и модульность), так и некоторые из направлений, в которых сейчас ведутся исследования в этой области (такие, как выявление и применение шаблонов проектирования и

появление повторно используемых программных компонентов). Также будет проанализирован тот эффект, который парадигма объектно-ориентированного программирования произвела на эту область компьютерных наук.

7.1. Вопросы и упражнения

1. Почему количество строк в программе нельзя считать надежной мерой ее сложности?
2. Предложите метрику для оценки качества программного обеспечения. Какие, по вашему мнению, слабые места она имеет?
3. С помощью какого метода можно определить, сколько ошибок содержится в определенной части программного обеспечения?
4. Приведите два примера направлений в области технологии разработки программного обеспечения, в которых определенный успех уже был достигнут либо продвижение наблюдается непосредственно в данный момент.

7.2. Жизненный цикл программного обеспечения

Важнейшим понятием в технологии разработки программного обеспечения является жизненный цикл программы.

Жизненный цикл в целом

Жизненный цикл программы представлен на рис. 7.1. Здесь отражен тот факт, что, будучи однажды созданной, программа входит в цикл, включающий ее использование и модификацию (другой термин — *сопровождение*); продолжительность этого цикла распространяется на все время жизни программы. Такая картина характерна и для многих промышленных изделий. Отличие заключается лишь в том, что для таких изделий фазу модификации точнее было бы назвать ремонтом или техническим обслуживанием, тогда как в случае программного обеспечения фаза модификации обычно предполагает внесение исправлений или обновлений. В действительности программное обеспечение переходит в фазу модификации потому, что в нем обнаруживаются ошибки, возникают те или иные изменения в области его применения, что требует внесения соответствующих изменений в программное обеспечение, или из-за того, что предыдущая модификация в одной части программного обеспечения, вызвала появление проблем в других его частях.

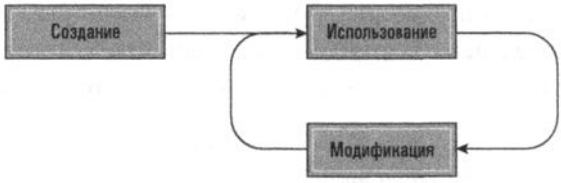


Рис. 7.1. Жизненный цикл программы

Независимо от того, почему программа модифицируется, необходимо, чтобы некто (как правило, не автор ее исходной версии) изучил и понял программу и ее документацию, а если не всю программу, то хотя бы ту часть, которая относится к делу. В противном случае любая модификация может вызвать намного больше новых проблем, чем позволит решить уже имеющихся. Достижение необходимой степени понимания является сложной задачей, даже если программа правильно разработана и документирована. Фактически именно на этой стадии отдельные фрагменты программного обеспечения просто отбрасывают, исходя из утверждений (часто вполне справедливых), что проще разработать новую систему "с нуля", чем успешно модифицировать уже существующий пакет.

Опыт показывает, что незначительные дополнительные усилия, затраченные при разработке программы, впоследствии могут существенно изменить ситуацию, когда потребуется внести в нее изменения. Например, при обсуждении операторов описания данных в главе 6 было показано, как вместо безличного конкретного числового значения в программе может использоваться константа с символическим именем, что значительно упрощает последующие изменения. Как следствие бо́льшая часть исследований в области технологии разработки программного обеспечения концентрирует свое внимание именно на фазе разработки в жизненном цикле программы, имея своей целью достижение преимуществ за счет дополнительных усилий на данной стадии, обеспечивающих определенные выгоды впоследствии.

Традиционные этапы разработки программ

Фаза разработки в жизненном цикле программы традиционно включает следующие этапы: анализ требований, проектирование, реализацию и тестирование (рис. 7.2).



Основные положения для запоминания

- **Решая поставленную задачу, программист разрабатывает, реализует, тестирует, отлаживает и сопровождает (модифицирует) программу.**

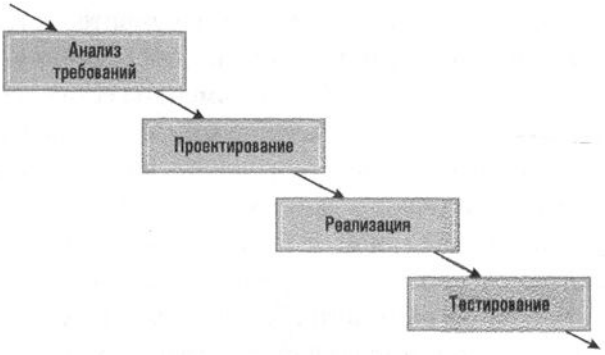


Рис. 7.2. Традиционное представление фазы разработки в жизненном цикле программного обеспечения

Анализ требований

Жизненный цикл программного обеспечения начинается с анализа требований, основная задача которого состоит в определении того, какие услуги будет предоставлять данная система, каковы будут условия предоставления этих услуг (временные ограничения, безопасность и т.д.) и каким образом внешний мир будет взаимодействовать с данной системой.

Анализ требований, выдвигаемых к создаваемой системе, предполагает значительный вклад со стороны **заинтересованных лиц** (это ее будущие пользователи, а также тех, кто имеет иные интересы, например юридические или финансовые). Фактически в тех случаях, когда конечным пользователем является организация, такая как компания или государственное агентство, которая намеревается нанять стороннего разработчика ПО для фактического выполнения требуемого программного проекта, анализ требований может начаться с технико-экономического обоснования, проводимого исключительно заказчиком. В других случаях разработчик ПО может заниматься производством готового **коммерческого программного обеспечения** (*COTS* — Commercial Off-The-Shelf) для массового рынка, которое затем будет продаваться в розничных магазинах или загружаться через Интернет. В такой ситуации пользователь является более расплывчатой фигурой и анализ требований может начинаться с изучения рынка разработчиком программного обеспечения.

В любом случае процесс анализа требований включает сбор и анализ нужд потенциальных пользователей, обсуждение с заинтересованными сторонами возможных компромиссов в отношении их пожеланий и устанавливаемых ограничений, возможного уровня затрат, а также осуществимости поставленных задач. В конечном итоге вырабатывается набор требований, определяющих функции и услуги, которые должна обеспечивать готовая система программного

обеспечения. Эти требования записываются в документе, который называют **спецификацией требований к программному обеспечению**. В некотором смысле этот документ представляет собой письменное соглашение между всеми заинтересованными сторонами, назначение которого — направлять процесс разработки программного обеспечения и предоставлять средства для разрешения любых споров, которые могут возникнуть по ходу процесса разработки. Важность спецификации требований к программному обеспечению демонстрируется тем фактом, что профессиональные организации, такие как IEEE, а также крупные заказчики программного обеспечения, такие как Министерство обороны США, приняли соответствующие стандарты по их составлению.

С точки зрения разработчика программного обеспечения, спецификация требований к ПО должна четко определять ту цель, которую необходимо достичь посредством разрабатываемого программного обеспечения. Однако очень часто этот документ оказывается неспособным обеспечить требуемую стабильность. В действительности большинство практиков в области разработки ПО утверждают, что недостаточный уровень общения и изменение требований являются основными причинами перерасхода средств и задержек с поставкой готового продукта во всей отрасли разработки программного обеспечения. Очень немногие клиенты будут настаивать на серьезных изменениях в плане этажа здания после того, как его фундамент уже будет построен, однако существует множество организаций, которые расширили или иным образом изменили желаемые характеристики разрабатываемой программной системы много позже того, как разработка программного обеспечения уже началась. Так бывает, когда компания-заказчик приходит к заключению, что система, которая изначально разрабатывалась только для ее дочерней компании, теперь будет предназначаться для всей корпорации, либо когда новые технологические достижения позволяют достичь намного большего, чем это было возможно в тот момент, когда проводился первоначальный анализ требований. Как бы там ни было, инженеры-программисты доказали, что прямое и частое общение с заинтересованными сторонами проекта является обязательным.



Основные положения для запоминания

- **Консультации и тесное общение с пользователями программы являются важным аспектом разработки программного обеспечения, позволяющим решать любые возникающие проблемы.**
- **Разработка программного обеспечения включает выявление возникающих у программистов и конечных пользователей вопросов, оказывающих влияние на решение проблем.**

Проектирование

Результат этапа анализа требований — это описание создаваемого программного продукта, тогда как этап проектирования предусматривает разработку плана построения создаваемой системы. В некотором смысле анализ требований заключается в определении проблемы, которую предстоит решить, а проектирование — в отыскании способа решения этой проблемы. С точки зрения непрофессионала, анализ требований часто приравнивается к решению, *что* программная система должна делать, тогда как проектирование приравнивается к нахождению решения, *как* система будет это делать. Хотя такое определение можно считать достаточно наглядным, многие разработчики программного обеспечения утверждают, что оно некорректно, поскольку на самом деле многое из категории *как*, рассматривается уже при анализе требований, тогда как многое из категории *что* определяется лишь в процессе проектирования.

Именно на стадии проектирования устанавливается внутренняя структура создаваемой системы программного обеспечения. А значит, результатом этапа проектирования должно быть подробное описание структуры системы программного обеспечения, которое затем будет преобразовано в программы.

Если проект предполагает возведение офисного здания, а не создание системы программного обеспечения, этап проектирования будет включать разработку подробных структурных планов здания, отвечающих всем установленным требованиям. Например, такие планы должны будут включать набор чертежей, описывающих предлагаемое здание на различных уровнях детализации. Именно на основании этих документов и будет построено реальное здание. Методы разработки подобных планов развивались в течение многих лет и включают стандартизированные системы обозначений и многочисленные методологии моделирования и построения чертежей.

Аналогичным образом создание диаграмм и моделирование играют важную роль и в разработке программного обеспечения. Однако методологии и системы обозначений, используемые разработчиками программного обеспечения, не являются настолько устоявшимися, как в области архитектуры и строительства. При сравнении с четко определенной дисциплиной архитектуры практика разработки программного обеспечения выглядит очень динамичной, поскольку исследователи и сейчас всеми силами продолжают попытки найти лучшие подходы к процессу разработки программного обеспечения. Подробнее об этих изменениях в подходах к проектированию ПО речь пойдет в разделе 7.3, а в разделе 7.5 мы обсудим некоторые из существующих систем обозначений и связанные с ними методологии построения диаграмм и моделирования.

Ассоциация IEEE

Международная некоммерческая ассоциация IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и радиоэлектронике) была создана в 1963 году в результате слияния американских обществ IAEE (основанного в 1884 году 25-ю инженерами-электротехниками, среди которых был и Томас Эдисон) и IRE (основанного в 1912 году). IEEE — всемирная организация инженеров в области электротехники, радиоэлектроники и радиоэлектронной промышленности со штаб-квартирой в г. Пискатауэй, штат Нью-Джерси, США. Она направляет деятельность 39-ти технических обществ, таких как общество инженеров аэрокосмических и электронных систем, общество специалистов в области лазеров и электрооптики, общество инженеров по робототехнике и автоматике, общество специалистов по самоходной технике, а также компьютерное общество. Наряду с другими видами деятельности, IEEE участвует в разработке стандартов. В частности, именно усилия этой организации позволили принять стандарты на форматы чисел с плавающей точкой (см. главу 1), которые используются в большинстве современных компьютеров.

Веб-сайт ассоциации IEEE находится по адресу `http://www.ieee.org`; вебсайт компьютерного общества — `http://www.computer.org`; а документ, содержащий кодекс этики организации IEEE, можно найти по адресу `http://www.ieee.org/about/whatis/code.html`.

Реализация

Реализация включает собственно написание программ, создание файлов данных и разработку баз данных. Именно на этапе реализации становится заметным различие между задачами, стоящими перед **аналитиком программного обеспечения** (иначе называемого **системным аналитиком**) и **программистом**. Первый — это человек, принимающий участие во всем процессе разработки, возможно, с акцентом на этапы анализа требований и проектирования. Второй — это исполнитель, вовлеченный, прежде всего, в этап реализации. В самом узком понимании программисту поручается лишь написание программ, реализующих проект, созданный системным аналитиком. Указав на это различие, следует еще раз отметить, что в компьютерном сообществе не существует центрального органа, контролирующего использование терминологии. Многие из тех, кто носит звание системного аналитика, по сути являются программистами, и многие из тех, кого считают программистами (или, возможно, старшими программистами), на самом деле являются системными аналитиками в полном смысле этого слова. Подобное размывание терминологии вызывается тем фактом, что сегодня этапы процесса разработки программного обеспечения часто смешиваются, как будет показано ниже.

Тестирование

В прошлом в традиционной схеме этапов разработки тестирование, по существу, приравнивалось к процессу отладки программ и подтверждению того, что конечный программный продукт совместим со спецификацией требований к программному обеспечению. Однако сегодня подобное видение этапа тестирования считается слишком узким. Программы нельзя считать единственными готовыми продуктами, которые должны быть подвергнуты тестированию в процессе разработки программного обеспечения. В действительности результат каждого промежуточного этапа во всем процессе разработки должен быть "проверен" на точность. Более того, как будет показано в разделе 7.6, теперь тестирование признано лишь одним из элементов в общей борьбе за обеспечение качества ПО, что является требованием, пронизывающим весь жизненный цикл программного обеспечения. Исходя из этого многие разработчики программного обеспечения теперь утверждают, что тестирование больше не следует рассматривать как отдельный этап в процессе разработки программного обеспечения. В действительности этот этап и его многочисленные проявления следует включить в другие этапы, создав тем самым трехступенчатый процесс разработки, составляющие элементы которого должны иметь такие имена, как "анализ *и подтверждение* требований", "проектирование и *проверка правильности* проекта" и "реализация и *тестирование*".

К сожалению, несмотря на использование современных методов обеспечения качества, большие системы программного обеспечения все еще могут содержать ошибки даже после продолжительного тестирования. Многие из этих ошибок могут оставаться незамеченными на протяжении всего жизненного цикла системы, в то время как другие могут стать причиной весьма опасных сбоев или отказов. Устранение таких ошибок является одной из важнейших задач технологии разработки программных систем. Тот факт, что количество обнаруживаемых ошибок все еще весьма значительно, говорит о том, что исследования в этой области необходимо продолжать.

7.2. Вопросы и упражнения

1. Как в жизненном цикле программного обеспечения этап разработки влияет на этап модификации (или сопровождения)?
2. Кратко охарактеризуйте каждый из четырех этапов (анализ требований, проектирование, реализация и тестирование) фазы разработки в жизненном цикле программного обеспечения.
3. В чем состоит назначение спецификаций требований к программному обеспечению?

7.3. Методологии разработки программного обеспечения

Ранние подходы к проектированию программного обеспечения требовали строго последовательного выполнения этапов анализа, проектирования, реализации и тестирования. Предполагалось, что риск, связанный с использованием метода проб и ошибок при разработке больших систем программного обеспечения, слишком велик. В результате разработчики программного обеспечения настаивали на полном завершении разработки спецификаций требований к системе до начала ее проектирования. Точно так необходимо было полностью завершить этап проектирования системы до начала ее реализации. Подобная схема процесса разработки получила название **модель водопада** (по аналогии с движением потока падающей воды, поскольку процесс разработки должен был двигаться только в одном направлении).

Относительно недавно методы проектирования программного обеспечения были изменены таким образом, чтобы отразить противоречие между высокоструктурированной средой, диктуемой моделью водопада, и свободно развивающимся процессом проб и ошибок, который жизненно важен при творческом подходе к решению задач. Эти изменения выразились в появлении **пошаговой модели** разработки программного обеспечения. В соответствии с этой моделью требуемая система программного обеспечения создается поэтапно. Первый вариант является некоторой упрощенной версией требуемой системы с ограниченным набором функций. После того как эта версия будет протестирована и, возможно, оценена будущим пользователем, к ней последовательно добавляют и тестируют другие функции, и так до тех пор, пока система не приобретет законченный вид. Например, если разрабатываемая система является приложением для ведения личных дел пациентов больницы, то ее первое приближение может включать только функцию просмотра личных дел пациентов, составляющих лишь небольшую выборку из обширного общего архива. После того как эта версия окажется работоспособной, в ней поэтапно будут реализованы дополнительные функции, такие как добавление новых записей и обновление уже существующих.



Основные положения для запоминания

- **Последовательное добавление новых проверенных программных сегментов к корректно работающей программе позволяет создавать большие правильно работающие программы.**

Другой моделью, которая представляет собой отход от строгого соблюдения модели водопада, является **итерационная модель**, которая похожа на пошаговую модель и фактически иногда приравнивается к ней, хотя эти две функции различны. В то время как пошаговая модель несет в себе понятие *расширения* каждой предварительной версии продукта в более крупную версию, итерационная модель включает в себя концепцию *уточнения* каждой версии. В действительности пошаговая модель включает в себя базовый итеративный процесс, а итерационная модель может предусматривать постепенное добавление функций.



Основные положения для запоминания

- **Итеративный, пошаговый процесс разработки ПО позволяет создать программу, корректно решающую поставленные задачи.**

Важным примером итеративных методов является **RUP** (*Rational Unified Process* — рациональный унифицированный процесс), который был создан компанией Rational Software, в настоящее время являющейся подразделением корпорации IBM. По сути, RUP — это парадигма разработки программного обеспечения, переопределяющая этапы фазы разработки в жизненном цикле программного обеспечения и предоставляющая рекомендации по выполнению этих этапов. Эти рекомендации, а также CASE-инструменты для их поддержки, продаются корпорацией IBM. Сегодня методология RUP широко применяется в индустрии программного обеспечения. На практике ее высокая популярность привела к разработке непатентованной версии, получившей название "**унифицированный процесс**", которая доступна на некоммерческой основе.

В пошаговых и итеративных моделях иногда используют современную тенденцию в разработке программного обеспечения, предполагающую создание **прототипов**, т.е. когда создаются и оцениваются неполные версии разрабатываемой системы, называемые прототипами. В пошаговой модели прототип развивается в конечную версию системы, поэтому данный вариант метода создания прототипов именуется **эволюционным**. В других случаях прототипы могут отбрасываться, уступая место новым реализациям конечного проекта. Такой вариант метода создания прототипов называется методом с **отбрасыванием прототипов**. Примером использования этого варианта может служить **быстрое создание прототипов**, при котором на ранних стадиях разработки очень быстро создается серия упрощенных вариантов разрабатываемой системы. Такой прототип может состоять всего лишь из нескольких эскизов компоновки экрана, показывающих, как система будет взаимодействовать с пользователем

и каковы будут ее возможности. В данном случае задача состоит не в создании рабочей версии продукта, а в получении средства демонстрации, предназначенного для углубления взаимопонимания между участвующими в разработке сторонами. В частности, метод быстрого создания прототипов доказал свою эффективность в отношении систематизации требований к системе, выдвинутых на этапе анализа, а также в качестве средства для проведения презентаций возможностей системы ее потенциальным заказчикам.

Менее формальное воплощение пошаговых и итеративных идей, которое уже на протяжении многих лет используется компьютерными энтузиастами и любителями, широко известно как Разработка ПО: **разработка с открытым исходным кодом**. Это подход, на основании которого производится бо́льшая часть современного бесплатного программного обеспечения. Возможно, наиболее ярким примером его использования является операционная система Linux, которая изначально разрабатывалась как система с открытым исходным кодом под руководством Линуса Торвальдса. Разработка программного пакета с открытым исходным кодом обычно происходит следующим образом: один автор пишет первоначальную версию программного обеспечения (как правило, для удовлетворения собственных потребностей) и размещает исходный код и сопровождающую его документацию в Интернете. Оттуда все это может быть загружено и использовано другими, причем совершенно бесплатно. Поскольку эти другие пользователи имеют полный исходный код и документацию, у них есть возможность изменять или улучшать это программное обеспечение в соответствии со своими потребностями либо исправлять обнаруженные ошибки. Они сообщают об этих изменениях первоначальному автору, который включает их в опубликованную версию программного обеспечения, делая эту расширенную версию доступной для дальнейших изменений. На практике подобный программный пакет может развиваться очень быстро, проходя через несколько последовательных расширений всего за одну неделю.

Возможно, наиболее заметный отход от модели водопада представлен набором методологий, известных как **гибкие методы** (agile methods), каждый из которых предполагает раннее и быстрое внедрение на пошаговой основе, реагирование на изменяющиеся требования и снижение акцента на строгий анализ и разработку исходных требований к системе. Одним из примеров гибкого метода является **экстремальное программирование** (*XP* — Extreme Programming). Следуя модели XP, программное обеспечение разрабатывается командой из менее чем дюжины человек, работающих в общем рабочем пространстве, в котором они свободно обмениваются идеями и помогают друг другу в процессе разработки проекта. Согласно этой модели программное обеспечение разрабатывается пошагово посредством повторяющихся ежедневных циклов неформального анализа требований, проектирования, реализации и тестирования. В результате новые

расширенные версии создаваемого программного пакета появляются на регулярной основе и каждая из них может быть оценена заинтересованными сторонами проекта и использована для указания направления дальнейшего расширения системы. Подводя итог, можно сказать, что гибкие методы характеризуются высокой изменчивостью, которая резко контрастирует с моделью водопада, с которой у нас ассоциируется образ группы менеджеров и программистов, работающих в отдельных офисах и строго выполняющих четко определенные фрагменты общей задачи разработки программного обеспечения.

Контраст, получаемый при сравнении модели водопада и метода XP, подчеркивает всю широту диапазона методологий, которые сейчас используются в процессах разработки программного обеспечения в надежде найти лучшие способы создания надежного программного обеспечения, причем максимально эффективным способом. Исследования в этой области — это непрекращающийся процесс. Определенный прогресс уже достигнут, но предстоит еще немало работы.

7.3. Вопросы и упражнения

1. Кратко охарактеризуйте различия между традиционной моделью водопада в разработке программного обеспечения и более новыми парадигмами пошаговой и итерационной разработки.
2. Назовите три парадигмы разработки ПО, которые демонстрируют отход от жестко определенной последовательности этапов разработки в модели водопада.
3. В чем заключаются различия между традиционным эволюционным методом создания прототипов и разработкой программ с открытым исходным кодом?
4. Какие потенциальные проблемы, по вашему мнению, могут возникнуть с точки зрения прав собственности на программное обеспечение, разработанное с использованием методологии открытого исходного кода?

7.4. Модульность

Одним из ключевых положений в разделе 7.2 было утверждение о том, что для модификации программы необходимо разобраться в принципах ее работы или по крайней мере тех ее частей, которые относятся к делу. Зачастую этого нелегко достичь даже в небольших программах, а в крупных системах

программного обеспечения это было бы практически невозможно, если бы не принцип **модульности**, предполагающий разделение программного обеспечения на поддающиеся осмыслению элементы, обычно называемые **модулями**, каждый из которых сконструирован так, чтобы выполнять только определенную часть общей задачи.

Реализация модулей

Модульности можно достичь многими способами. В главах 5 и 6 было показано, что в контексте императивной парадигмы модули можно реализовать в виде отдельных функций. В противоположность этому в контексте объектно-ориентированной парадигмы в качестве основных модульных составляющих использовались объекты. Эти различия очень важны, поскольку они определяют основную цель в самом начале процесса разработки программного обеспечения. Что будет такой целью: представить общую задачу в виде отдельных контролируемых процессов или же идентифицировать объекты в системе и понять, как они взаимодействуют?

Чтобы проиллюстрировать это, давайте рассмотрим, как процесс разработки простой модульной программы для симуляции игры в теннис будет происходить при выборе для этой программы императивной либо объектно-ориентированной парадигмы. В императивной парадигме работа начинается с рассмотрения действий, которые должны произойти. Поскольку каждая серия ударов по мячу инициируется игроком, подающим мяч, имеет смысл начать рассмотрение с функции `Serve()`(подача), представляющей процесс подачи. Ее задача — вычислить начальную скорость и направление полета мяча, исходя из характеристик игрока, возможно, с добавлением небольшой вероятности. Далее нужно будет определить путь мяча. (Попадет ли он в сетку? Где произойдет отскок от поверхности корта?) Пусть решено выполнять все эти вычисления в другой функции с именем `ComputePath()` (вычисление пути). На следующем этапе необходимо определить, сможет ли другой игрок вернуть мяч, и если это так, потребуется вычислить новую скорость и направление полета мяча. Все эти вычисления можно поместить в функцию с именем `Return()`(ответный удар).

Продолжая таким образом, в конечном счете мы можем прийти к модульной структуре, представленной на рис. 7.3 в виде **структурной схемы**. На этой схеме функции представлены прямоугольниками, а зависимости функций (реализованные вызовами функций) — стрелками. В частности, на этой схеме видно, что вся игра контролируется функцией с именем `ControlGame()` (управление игрой), а для выполнения своей задачи функция `ControlGame()` вызывает на выполнение функции `Serve()` (подача мяча), `Return()` (ответный удар), `ComputePath()` (вычисление пути) и `UpdateScore()` (обновить счет).

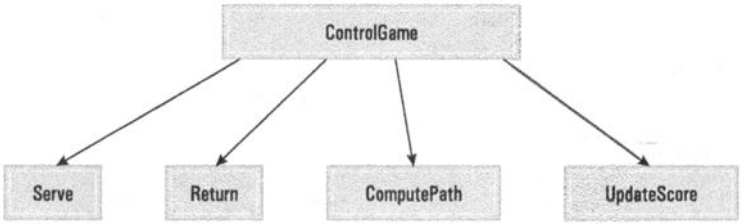


Рис. 7.3. Структурная схема простой игры

Обратите внимание, что структурная схема не показывает, как каждая функция выполняет свою задачу. Скорее, функции на ней просто идентифицируются и указываются зависимости между ними. В действительности функция `ControlGame()` может выполнять свою задачу, сначала вызывая функцию `Serve()`, затем неоднократно вызывая функции `ComputePath`) и `Return()` — до тех пор, пока одна из них не сообщит об ошибке, после чего, перед повторением всего цикла, она вызывает функцию `UpdateScore()`, после чего снова обращается к функции `Serve()`, начиная новый цикл.

На этом этапе мы получили только очень упрощенную схему желаемой программы, но наша точка зрения уже была продемонстрирована. В соответствии с императивной парадигмой мы разрабатывали программу, рассматривая *действия*, которые должны быть выполнены, и поэтому получили проект, в котором модули являются функциями.

А теперь давайте еще раз повторим процедуру разработки этой игры, но на этот раз в контексте объектно-ориентированной парадигмы. На первый взгляд, все просто: в игре есть два игрока, которых можно представить двумя объектами: `PlayerA` и `PlayerB`. Эти объекты будут иметь одинаковую функциональность, но разные характеристики. (Оба должны уметь подавать и отбивать мяч, но могут делать это с разными навыками и силой удара.) Следовательно, эти объекты могут быть экземплярами одного и того же класса. (Напомним, что в главе 6 было введено понятие класса: это шаблон, в котором определены функции, называемые методами, и атрибуты, называемые переменными экземпляра, которые должны быть связаны с каждым объектом.) Этот класс, который мы назовем `PlayerClass`, будет содержать методы `serve()` и `return()`, обеспечивающие имитацию действий игрока, т.е. подачу и ответный удар соответственно. Он также будет содержать атрибуты, такие как `skill` (умение) и `endurance` (выносливость), значения которых будут отражать эти характеристики каждого игрока. На данном этапе проект игры может быть представлен в виде диаграммы, показанной на рис. 7.4. Здесь мы видим, что `PlayerA` и `PlayerB` являются экземплярами класса `PlayerClass` и этот класс содержит атрибуты `skill` и `endurance`, а также методы `serve()` и `returnVolley()`. (Обратите внимание, что на рис. 7.4 имена объектов подчеркнуты, чтобы отличать их от имен классов.)

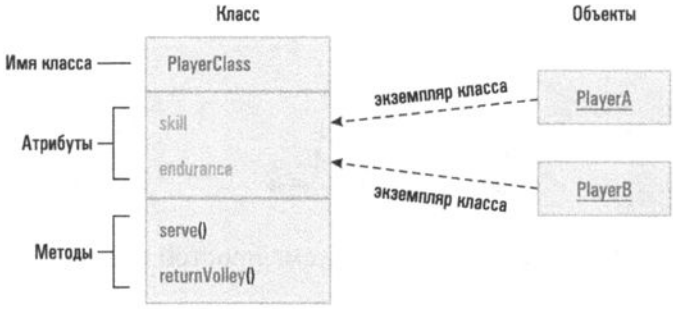


Рис. 7.4. Структура класса `PlayerClass` и объекты, созданные на его основе

Далее нам нужен объект, который будет играть роль судьи, определяющего, являются ли действия, совершенные игроками, допустимыми. Например, задел ли поданный мяч сетку и попал ли он в ту часть корта, в которую разрешается попадание при подаче? Для этой цели мы можем создать объект с именем `Judge` (судья), который будет содержать методы `evaluateServe` (оценка подачи) и `evaluateReturn` (оценка ответного удара). Если объект `Judge` определяет подачу или ответный удар как допустимый, игра продолжается. В противном случае объект `Judge` отправляет сообщение другому объекту с именем `Score` (счет) для фиксации соответствующих результатов.

На данный момент проект игры в теннис включает уже четыре объекта: `PlayerA`, `PlayerB`, `Judge` и `Score`. Чтобы уточнить структуру нашего проекта, рассмотрим последовательность событий, которые могут происходить во время подачи мяча, — она показана на рис. 7.5, на котором объекты нашей программы представлены в виде прямоугольников. Назначение этого рисунка — представление всех взаимодействий, которые имеют место между этими объектами в результате вызова метода `serve()` объекта `PlayerA`. События появляются в хронологическом порядке, соответствующем перемещению вниз по рисунку. Как показано первой горизонтальной стрелкой, объект `PlayerA` сообщает о своей подаче мяча объекту `Judge`, вызывая его метод `valuServe()` (оценить подачу). Далее объект `Judge` определяет, что подача мяча прошла успешно, и предлагает объекту `PlayerB` отбить его, вызвав метод `returnVolley()` этого объекта. Цикл передач мяча завершается, когда объект `Judge` определяет, что объект `PlayerA` допустил ошибку, и просит объект `Score` записать результат.

Как и в случае императивной версии этого примера, объектно-ориентированный вариант программы на этом этапе очень упрощен. Однако мы уже достаточно продвинулись, чтобы увидеть, каким образом выбор объектно-ориентированной парадигмы приводит к модульной структуре, в которой основные компоненты являются объектами.

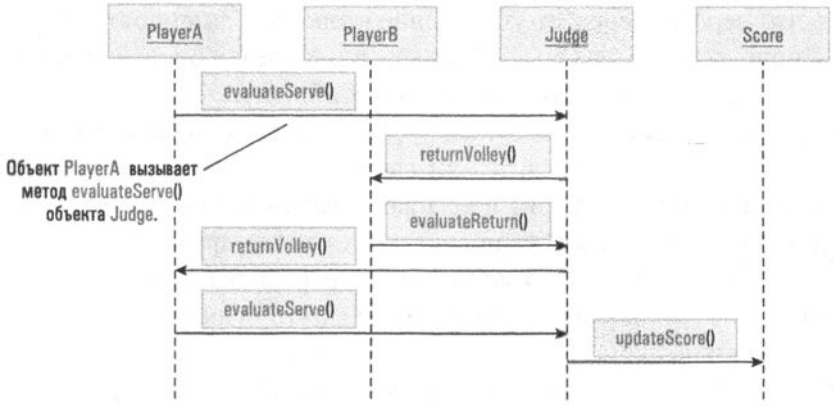


Рис. 7.5. Серия взаимодействий между объектами, вызванная выполнением метода `serve()` объекта `PlayerA`

Связанность модулей

Выше в этой главе модульность была предложена как средство получения управляемого программного обеспечения. Идея состоит в том, что каждая последующая модификация, вероятнее всего, коснется относительно небольшого числа модулей, так что в процессе ее выполнения достаточно будет ограничиться рассмотрением только этой части системы, вместо того чтобы возиться со всем пакетом. Данное утверждение, безусловно, основывается на предположении, что внесение изменений в один из модулей не окажет непредвиденного влияния на работу других модулей системы. Соответственно, при проектировании модульной системы основная задача состоит в обеспечении максимальной независимости отдельных модулей, хотя некоторые взаимосвязи между модулями все-таки необходимы, поскольку данные модули должны образовать согласованно функционирующую систему. Наличие подобных связей между модулями системы называют **связанностью**. Следовательно, задача достижения максимальной независимости модулей соответствует минимизации их связанности. И действительно, одна из метрик, которые используются для измерения сложности программной системы (а следовательно, для получения средства оценки затрат на обслуживание программного обеспечения), заключается в измерении ее межмодульной связанности.

Связанность модулей системы может существовать в нескольких различных формах. Одна из них — это **связанность по управлению**. В этом случае один модуль передает управление другому так, как это происходит при возврате и передаче управления при вызове функции. Структурная схема на рис. 7.3 представляет пример связанности по управлению, которая существует между функциями. В частности, стрелка от модуля `ControlGame()` к модулю `Serve()`

указывает, что первый передает управление второму. Связанность по управлению также присутствует и на рис. 7.5, на котором стрелки определяют последовательность передачи управления от объекта к объекту.

Другая форма связанности модулей — это **связанность по данным**, т.е. совместное использование одних и тех же данных несколькими модулями. Если два модуля манипулируют одним и тем же элементом данных, то изменения, внесенные в один модуль, могут повлиять на работу другого, а изменения в формате самих данных могут оказать влияние на работу обоих модулей.

Связанность по данным между функциями может существовать в двух формах. Одной из них является явная передача данных от одной функции к другой в форме параметров. Такая связь представляется в структурной диаграмме стрелкой между функциями, которая дополнительно помечается с целью определения передаваемых данных. Сама же стрелка задает направление, в котором этот элемент данных передается. Например, на рис. 7.6 представлена расширенная версия рис. 7.3, на котором указано, что функция `ControlGame()` сообщает функции `Serve()`, характеристики какого именно игрока следует имитировать при вызове функции `Serve()`, и что функция `Serve()` при завершении работы передает функции `ControlGame()` информацию о вычисленной ею траектории мяча.

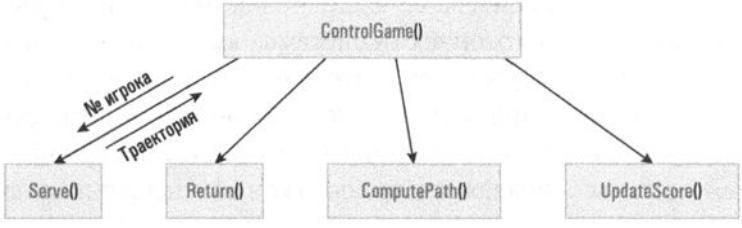


Рис. 7.6. Структурная схема, содержащая информацию о связанности по данным

Похожее связывание по данным имеет место между объектами и в объектно-ориентированном варианте проекта. Например, когда объект `PlayerA` просит объект `Judge` оценить его подачу (см. рис. 7.5), он должен передать последнему сведения о траектории мяча. С другой стороны, одно из преимуществ объектно-ориентированной парадигмы состоит в том, что она по своей сути стремится свести связанность по данным между объектами к минимуму. Это связано с тем, что методы внутри объекта обычно включают в себя все те функции, которые манипулируют внутренними данными объекта. Например, объект `PlayerA` будет содержать информацию о характеристиках этого игрока, а также обо всех методах, которые требуют для своего выполнения эту информацию. Соответственно, нет необходимости передавать эту информацию другим объектам, а

следовательно, межобъектная связанность по данным здесь всегда сводится к минимуму.

В противоположность явной передаче данных в качестве параметров, данные также могут быть неявно распределены между модулями в форме **глобальных данных**. Они представляют собой элементы данных, автоматически доступные для всех модулей в системе, что отличает их от локальных элементов данных, доступных только внутри конкретных модулей, если их явно не передают другому модулю. Большинство языков высокого уровня предоставляют способы реализации как глобальных, так и локальных данных, но к использованию в системе глобальных данных следует относиться с осторожностью. Проблема заключается в том, что человеку, пытающемуся внести изменения в модуль, использующий глобальные данные, может быть сложно установить, как именно этот модуль взаимодействует с другими модулями. Короче говоря, использование глобальных данных может снизить полезность модуля в качестве абстрактного инструмента.

Связность элементов модуля

Почти столь же важной задачей, как минимизация связей между модулями, является достижение максимальной внутренней связности элементов внутри каждого модуля. Термин **связность** (cohesion) относится именно к этим внутренним связям или, другими словами, к степени взаимосвязанности внутренних частей модуля. Чтобы убедиться в важности понятия связности, необходимо выйти за рамки процесса первоначальной разработки системы и обратиться ко всему жизненному циклу программного обеспечения. Если возникает необходимость внести изменения в модуль, то существование множества разнообразных действий внутри него может существенно усложнить процесс, который в противном случае мог бы оказаться совсем простым. Следовательно, помимо поиска любых возможностей уменьшить связывание отдельных модулей, разработчики программного обеспечения должны стремиться к достижению самого высокого уровня связности элементов внутри каждого модуля.

Самая слабая форма связности — **логическая связность**. Этот тип связности внутри модуля строится на том факте, что его внутренние элементы выполняют действия, сходные по своей логической природе. Например, рассмотрим модуль, осуществляющий все связи системы с внешним миром. "Клеем", скрепляющим элементы такого модуля, является то, что все действия внутри этого модуля так или иначе относятся к обмену информацией с внешним миром. Однако назначение такого обмена в каждом конкретном случае может сильно различаться. Например, одни действия могут быть связаны с получением данных, а другие — с выдачей сообщений об ошибках.

Более сильная форма связности — **функциональная связность**, которая означает, что все части модуля фокусируются на выполнении одного действия. При проектировании в рамках императивной парадигмы функциональная связность часто может быть увеличена за счет выделения подзадач в отдельные модули с последующим использованием этих модулей в качестве абстрактных инструментов. Это хорошо демонстрируется в нашем примере игры в теннис (см. рис. 7.3), в котором модуль `ControlGame()` использует другие модули в качестве абстрактных инструментов, в результате чего он может сосредоточиться на наблюдении за игрой, не отвлекаясь на детали осуществления подачи мяча, ответного удара по мячу и ведения счета.

В объектно-ориентированном проектировании объекты в целом обычно являются только логически связными, так как методы внутри объектов зачастую выполняют слабо связанные действия. Единственное, что объединяет все методы объекта, — это то, что они выполняют эти действия с одним и тем же объектом. Например, в нашей игре в теннис каждый представляющий игрока объект будет содержать методы как для подачи мяча, так и для совершения ответного удара, в которых выполняются существенно разные действия. Следовательно, каждый такой объект представляет собой лишь логически связный модуль. Однако разработчики программного обеспечения должны стремиться делать каждый метод внутри объекта функционально связным. Другими словами, даже если объект в целом является всего лишь логически связным, каждый метод внутри объекта должен выполнять всего лишь одну функционально связную задачу, как показано на рис. 7.7.

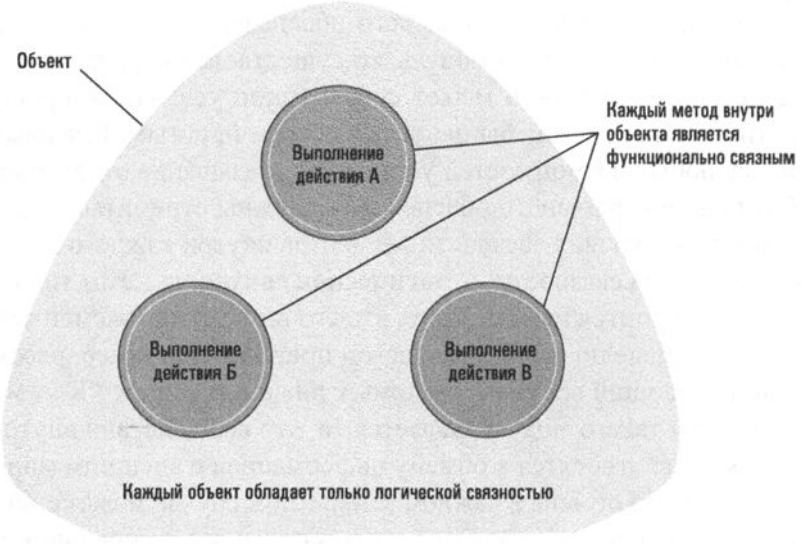


Рис. 7.7. Логическая и функциональная связность внутри объекта

Сокрытие информации

Один из краеугольных камней хорошего модульного дизайна заложен в концепции **сокрытия информации**, которая предполагает ограничение доступности любой информации лишь определенной частью программной системы. Здесь термин *информация* должен интерпретироваться в широком смысле, включая любые сведения о структуре и содержании программного блока. В общем понимании она включает в себя данные, тип используемых структур данных, системы кодирования, внутреннюю композиционную структуру модуля, логическую структуру процедурных единиц и любые другие факторы, касающиеся внутренних свойств модуля.

Смысл сокрытия информации заключается в том, чтобы не допустить в действиях модулей ненужной зависимости или нежелательного влияния на другие модули. В противном случае корректность работы модуля может вызвать сомнения, возможно, из-за ошибок при разработке других модулей или из-за ошибочных действий при модификации программного обеспечения. Например, если модуль не запрещает использование своих внутренних данных со стороны других модулей, то эти данные могут быть ими повреждены. Или, если один модуль спроектирован так, чтобы использовать преимущества внутренней структуры другого, он может начать работать некорректно, если эта внутренняя структура будет изменена.

Важно отметить, что сокрытие информации имеет два аспекта: в первом оно воспринимается как цель проектирования, а во втором — как цель реализации. Модуль должен быть *спроектирован* так, чтобы другие модули не нуждались в доступе к его внутренней информации. С другой стороны, модуль должен быть *реализован* таким способом, который обеспечит неприкосновенность его границ. Примерами первого подхода являются максимизация связности и минимизация связывания. Примеры второго подхода включают использование локальных переменных, применение инкапсуляции и использование четко определенных структур управления.

И в завершение следует отметить, что сокрытие информации занимает центральное место в теме абстракции и использования абстрактных инструментов. Действительно, концепция абстрактного инструмента — это концепция "черного ящика", внутренние особенности которого могут игнорироваться пользователем, что позволяет ему сосредоточиться исключительно на особенностях того более крупного приложении, над которым он работает. В этом смысле сокрытие информации соответствует концепции изоляции абстрактного инструмента во многом так же, как защищенный от взлома корпус может использоваться для защиты сложного и потенциально опасного электронного оборудования. В обоих случаях обеспечиваются защита внешних пользователей от

внутренних опасностей компонента и одновременно защита его внутреннего содержания от нарушений со стороны пользователей.

Компоненты

Мы уже упоминали, что одним из препятствий в области разработки программного обеспечения является отсутствие готовых, поставляемых со стороны строительных блоков, из которых можно было бы успешно строить большие программные системы. Модульный подход к разработке программного обеспечения подает большие надежды в этом отношении. В частности, особенно полезной оказывается парадигма объектно-ориентированного программирования, поскольку объекты представляют собой законченные, автономные модули, имеющие четко определенные интерфейсы с их окружением. Как только объект или, вернее, класс, разработан для выполнения определенной роли, его можно использовать для выполнения этой роли в любой программе, требующей подобных действий. Кроме того, наследование обеспечивает средство уточнения определений заранее созданных классов в тех случаях, когда эти определения должны быть настроены в соответствии с потребностями конкретного приложения. Поэтому неудивительно, что для объектно-ориентированных языков программирования C++, Java и C# разработаны обширные библиотеки готовых "шаблонов", на основании которых программисты легко могут реализовать такие объекты, которые необходимы им для выполнения конкретных ролей. В частности, для языка C++ существует стандартная библиотека шаблонов C++ (STL), в среде программирования языка Java доступен интерфейс Java Application Programmer Interface (API), а программисты на языке C# имеют доступ к библиотеке классов .NET Framework Class Library.

Тот факт, что объекты и классы имеют необходимый потенциал для предоставления сборных строительных блоков при разработке программного обеспечения, вовсе не означает, что они идеальны. Одна из проблем заключается в том, что для сборки они предоставляют относительно небольшие блоки. Поэтому объект на самом деле является частным случаем более общего понятия **компонента**, который по определению является повторно используемой единицей программного обеспечения. На практике большинство компонентов создается с использованием объектно-ориентированной парадигмы и имеет вид набора из одного или нескольких объектов, функционирующих как автономная самодостаточная единица.

Исследования в области разработки и использования компонентов привели к появлению новой области, известной как **компонентная архитектура** (также известная как *разработка программного обеспечения на основе компонентов*), в которой традиционную роль программиста заменяет **ассемблер компонентов**, конструирующий программные системы из готовых компонентов, которые

во многих средах разработки представляются в виде значков в графическом интерфейсе. Вместо того чтобы заниматься внутренним программированием компонентов, методология ассемблера компонентов предполагает простой выбор необходимых компонентов из коллекций предопределенных компонентов с последующим их соединением — с минимальной настройкой — для получения желаемой функциональности. И действительно, основное свойство хорошо разработанного компонента состоит в том, что его можно расширить для охвата функций конкретного приложения без необходимости внесения внутренних изменений.



Основные положения для запоминания

- **Разработка корректно работающих компонентов программ с последующим их комбинированием способствует созданию корректных программ.**

Той областью, в которой компонентная архитектура нашла для себя благодатную почву, являются системы для смартфонов. Из-за ограниченности ресурсов этих устройств отдельные их приложения на самом деле представляют собой всего лишь набор взаимодействующих компонентов, каждый из которых предоставляет в распоряжение приложения некоторую дискретную функцию. Например, каждый отображаемый в приложении экран обычно представляет собой отдельный компонент. За кулисами могут существовать и другие сервисные компоненты, предназначенные, скажем, для сохранения информации на карте памяти и ее последующего считывания, для выполнения некоторой непрерывной функции (например, воспроизведения музыки) или для доступа к информации через Интернет. Каждый из этих компонентов запускается и останавливается индивидуально, по мере необходимости, с целью эффективного обслуживания пользователя. Тем не менее каждое приложение выглядит как непрерывная серия отображаемых экранов и выполняемых действий.

Помимо мотивации к ограничению использования системных ресурсов, компонентная архитектура смартфонов приносит существенные дивиденды и при интеграции между приложениями. Например, приложение Facebook (известная социальная сеть) при запуске на смартфоне может использовать компоненты приложения Контакты для добавления всех друзей пользователя на Facebook в качестве контактов. Более того, приложение Телефон (приложение, реализующее функции телефонной связи) также может получить доступ к компонентам приложения Контакты для поиска абонента входящего вызова. В результате при поступлении входящего звонка от друга в сети Facebook на экране смартфона может быть отображена его фотография (вместе с его последним сообщением в Facebook).

7.4. Вопросы и упражнения

1. Чем роман отличается от энциклопедии в смысле степени связанности, существующей между его элементами, такими как главы, разделы или отдельные записи? Что можно сказать о связности этих элементов?
2. Спортивные мероприятия часто разделены на составляющие единицы. Например, футбольный матч разделен на таймы, а теннисный матч — на сеты. Проанализируйте связь между такими "модулями". В каком смысле такие единицы являются связными?
3. Совместимы ли задачи максимизации связности и минимизации связанности? Другими словами, будет ли естественным образом уменьшаться связанность при возрастании связности?
4. Дайте определения связанности, связности и сокрытию информации.
5. Расширьте структурную диаграмму, представленную на рис. 7.3, включив в нее сообщения, которые должны передаваться между модулями `ControlGame()` и `UpdateScore()`.
6. Нарисуйте диаграмму, подобную приведенной на рис. 7.5 и представляющую последовательность действий в том случае, если объект `PlayerA` выполнит неправильную подачу.
7. Чем отличается работа традиционного программиста от действий ассемблера компонентов?
8. Предполагая, что на большинстве смартфонов установлено несколько приложений класса личных организаторов (Календарь, Контакты, Часы, Почта, Карты, приложения доступа к социальным сетям, мессенджеры и т.д.), какие комбинации их функциональных компонентов вы сочтете полезными и интересными?

7.5. Инструменты и методы проектирования

В этом разделе мы исследуем некоторые методы моделирования и системы обозначений, используемые при создании программного обеспечения на этапах анализа и проектирования. Некоторые из них были разработаны еще в те годы, когда в области создания программного обеспечения доминировала императивная парадигма. Позднее одни из них нашли полезное применение и в контексте объектно-ориентированной парадигмы, тогда как другие, такие как структурная схема (см. рис. 7.3), по-прежнему являются специфическими

лишь для императивной среды. Мы начнем с рассмотрения некоторых методов, которые сохранились еще со времен доминирования императивного подхода, а затем перейдем к изучению новых объектно-ориентированных инструментов и обсуждению все возрастающей роли шаблонов проектирования.

Контроль версий

В настоящее время системы контроля версий являются частью большинства крупных проектов по разработке программного обеспечения. Глобальные и локальные сетевые системы контроля версий позволяют членам команды совместно работать над исходным кодом, предоставляя согласованный механизм для отслеживания отдельных изменений в коде. Инструменты контроля версий, такие как Git, Subversion или Mercurial, позволяют членам группы работать с локальной копией общего проекта, вносить в нее изменения, сделанные другими участниками, работающими параллельно, и направлять им собственные новые изменения, когда они будут к этому готовы. Системы контроля версий могут точно отслеживать, когда и кем каждое конкретное изменение вносится в общий проект, и могут откатывать код до более ранней версии, чтобы отменить изменения, вызвавшие нежелательные последствия. Контроль версий также может использоваться для создаваемой документации, для данных конфигурации и для структуры процесса сборки.

В действительности базовые функции контроля версий теперь включены во многие популярные пакеты программного обеспечения (фактически в любую программу с кнопкой Отменить, позволяющей откатывать множественные изменения), а также в облачные службы, такие как Google Docs.

Традиционные инструменты разработки

Хотя императивная парадигма предполагает создание программного обеспечения в терминах процедур или функций, способ определения этих функций состоит в том, чтобы рассматривать данные, которыми требуется манипулировать, а не сами функции. Теория говорит о том, что изучая, как данные перемещаются по системе, можно определить точки, в которых либо изменяются форматы данных, либо сливаются или разделяются пути их прохождения. В свою очередь, это именно те места, в которых и происходит обработка, а следовательно, анализ потока данных приводит к идентификации функций. Средством представления информации, полученной в результате таких исследований движения данных, является **диаграмма потоков данных**. На диаграмме потоков данных стрелки определяют пути прохождения данных, овалы представляют точки, в которых осуществляется манипулирование данными, а прямоугольники представляют

источники и места хранения данных. В качестве примера на рис. 7.8 показана элементарная диаграмма потоков данных, представляющая систему выставления счетов пациентам больницы. Обратите внимание: на диаграмме показано, что платежи (поступающие от пациентов) и записи пациентов (поступающие из файлов больницы) объединяются в овале Обработка платежей, из которого обновленные записи возвращаются в файлы больницы.



Рис. 7.8. Простая диаграмма потоков данных

Диаграммы потоков данных не только помогают идентифицировать процедуры на этапе проектирования в процессе разработки программного обеспечения, но и будут полезны при попытках разобраться в структуре создаваемой системы еще на этапе анализа. Действительно, построение диаграмм потоков данных может служить удобным средством улучшения взаимодействия между клиентами и разработчиками программного обеспечения, поскольку разработчик программного обеспечения пытается понять, чего хочет клиент, а клиент пытается описать свои ожидания. В результате эти диаграммы все еще находят себе применение, несмотря на то что императивная парадигма давно утратила свою популярность.

Другим инструментом, который годами использовался разработчиками программного обеспечения, является **словарь данных**, представляющий собой центральное хранилище информации об элементах данных, использующихся во всей программной системе. Эта информация включает в себя идентификатор, используемый для ссылки на данный элемент, из чего может состоять допустимое значение этого элемента (будет ли элемент всегда представлен числовым или, наоборот, всегда символьным значением; какой диапазон значений допустим для этого элемента), в котором этот элемент хранится (будет ли элемент храниться в файле или в базе данных и, если да, то в какой именно), и где в программном обеспечении имеются ссылки на этот элемент (каким модулям потребуется информация об этом элементе).

Одной из целей создания словаря данных является углубление взаимопонимания между заказчиками разрабатываемой системы программного обеспе-

чения и инженером-программистом, на которого возложена задача преобразования всех потребностей заказчика в спецификации требований к системе. В этом контексте создание словаря данных дает гарантии, что тот факт, что номер детали на самом деле не является числовым значением, будет выявлен еще на этапе анализа, а не обнаружен на более поздней стадии — проектирования или даже реализации. Другой целью, преследуемой при создании словаря данных, является обеспечение единообразия во всей разрабатываемой системе. Обычно именно при создании словаря в описании исходных данных выявляются факты избыточности и противоречивости. Например, элемент, на который в записях по складским запасам ссылаются как на `PartNumber`, в данных по реализации продукции может быть представлен как `PartId`. Более того, в отделе кадров элемент `Name` может использоваться для хранения данных о фамилиях сотрудников, тогда как в системе складского учета элемент `Name` может содержать название материала или детали, сохраняемой на складах.

UML — унифицированный язык моделирования

Диаграммы потоков данных и словари данных стали важными инструментами в арсенале разработчиков программ задолго до появления объектно-ориентированной парадигмы и до сих пор продолжают находить себе полезное применение, даже несмотря на то что императивная парадигма, для которой они изначально разрабатывались, утратила свою популярность. Теперь мы обратимся к более современному набору инструментов, известному как **UML** (*Unified Modeling Language* — унифицированный язык моделирования), который был разработан уже с учетом особенностей объектно-ориентированной парадигмы. Тем не менее первый инструмент из этого набора, который мы сейчас рассмотрим, будет полезен независимо от выбранной парадигмы, поскольку он предназначен для получения лишь общего представления о создаваемой системе с точки зрения ее пользователя. Этот инструмент — **диаграмма вариантов использования** (use case diagram) или иначе **диаграмма прецедентов**, пример которой представлен на рис. 7.9.

Диаграмма вариантов использования отображает создаваемую систему в виде большого прямоугольника, в котором взаимодействия (называемые **вариантами использования** или **прецедентами**) между системой и ее пользователями представлены в виде овалов, а пользователи системы (называемые **актерами**) представлены в виде стилизованных человечков (даже в тех случаях, когда актер не является человеком). Следовательно, диаграмма на рис. 7.9 показывает, что проектируемая система ведения записей больницы будет использоваться как врачами, так и медсестрами для получения медицинских карточек пациентов.



Рис. 7.9. Простая диаграмма вариантов использования (прецедентов)

Основные положения для запоминания

- **Функциональность программы часто описывается тем, как пользователь будет взаимодействовать с ней.**

Хотя диаграммы вариантов использования представляют взгляд на создаваемую систему программного обеспечения *извне*, язык UML также предлагает множество инструментов для представления *внутренней* организации системы при объектно-ориентированном подходе. Одним из них является **диаграмма классов**, которая представляет собой систему обозначений для представления структуры классов и отношений между классами (называемых в терминологии UML **ассоциациями**). В качестве примера рассмотрим отношения между врачами, пациентами и больничными палатами. Мы предполагаем, что объекты, представляющие эти сущности, создаются на основании классов `Physician`, `Patient` и `Room` соответственно.

На рис. 7.10 показано, как отношения между этими классами могут быть представлены в диаграмме классов языка UML. Здесь классы представлены прямоугольниками, а ассоциации — линиями. Линии ассоциаций могут (или не могут) быть помечены. Если они помечены, жирная стрелка может использоваться для указания направления, в котором должна читаться метка. Например, на рис. 7.10 стрелка после метки заботится о указывает на то, что врач заботится о пациенте, а не пациент заботится о враче. Иногда линиям ассоциации присваиваются две метки с целью обеспечить терминологию для прочтения ассоциации в обоих направлениях. Это положение иллюстрируется на рис. 7.10 на примере ассоциации между классами `Patient` и `Room`.



Рис. 7.10. Простая диаграмма классов

В дополнение к указанию ассоциаций между классами диаграмма классов может передавать *кратности* этих ассоциаций. То есть для каждой ассоциации можно указать, сколько экземпляров одного класса может быть связано через нее с экземплярами другого. Эта информация записывается на концах линий связи. В частности, на рис. 7.10 показано, что каждый пациент может занимать одну палату и в каждой палате может лежать нуль или один пациент. (Здесь предполагается, что каждая палата предназначена только для одного пациента.) Звездочка используется для обозначения произвольного неотрицательного числа. Таким образом, звездочка на рис. 7.10 указывает, что каждый врач может заботиться о многих пациентах, тогда как единица с другой стороны этой ассоциации означает, что о каждом пациенте заботится только один, лечащий, врач. (В нашем проекте принимаются во внимание только лечащие врачи.)

Для полноты картины следует отметить, что кратности ассоциаций встречаются в трех основных формах: в виде отношений "один-к-одному", "один-ко-многим" и "многие-ко-многим", как показано на рис. 7.11. Отношение "**один-к-одному**" иллюстрируется ассоциацией между пациентами и индивидуальными палатами, в которых они лежат. Здесь каждый пациент может занимать только одну палату, а каждая палата предоставляется только одному пациенту. Отношение "**один-ко-многим**" иллюстрируется ассоциацией между врачами и пациентами. В этом случае один врач связан со многими пациентами, тогда как каждый пациент связан только с одним (лечащим) врачом. Отношения "**многие-ко-многим**" появились бы в этом проекте в том случае, если бы в него дополнительно была включена концепция консультирующих врачей. Тогда каждый врач мог бы быть связан со многими пациентами, которых он консультирует, а каждый пациент мог бы быть связан со многими врачами, предоставлявшими ему консультации.

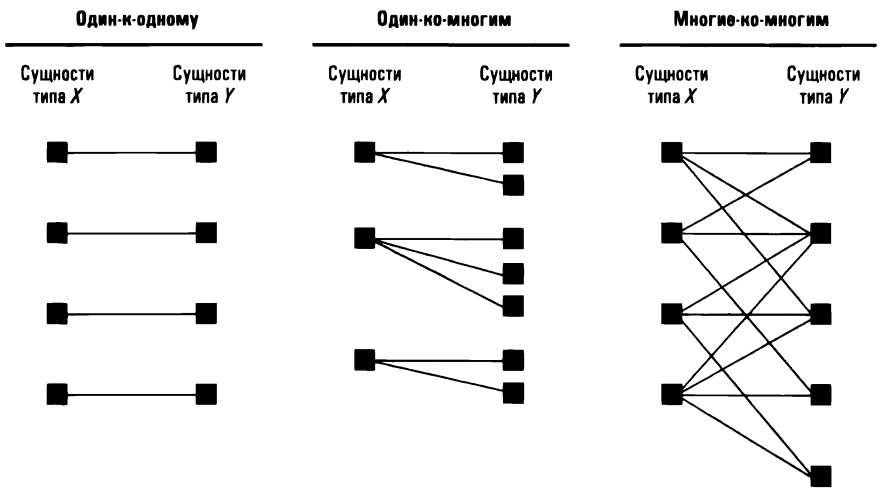


Рис. 7.11. Отношения "один-к-одному", "один-ко-многим" и "многие-ко-многим" между сущностями типа *X* и *Y*

В объектно-ориентированном проектировании часто бывает так, что один класс представляет собой более специализированную версию другого. В этих ситуациях говорят, что последний класс является обобщением первого. Язык UML предоставляет специальные обозначения для представления таких обобщений. Пример приведен на рис. 7.12, на котором представлены отношения обобщения между классами `MedicalRecord`, `SurgicalRecord` и `OfficeVisitRecord`. Здесь ассоциации между классами представлены стрелками с наконечниками без заливки, что в языке UML является стандартной нотацией для ассоциаций, отражающих отношение обобщения. Обратите внимание, что на диаграмме каждый класс представлен прямоугольником, содержащим имя, атрибуты и методы класса в формате, ранее использовавшемся на рис. 7.4. Это стандартный способ представления внутренних характеристик класса на диаграмме классов, принятый в языке UML. Информация, представленная на рис. 7.12, состоит в том, что класс `MedicalRecord` является обобщением как для класса `SurgicalRecord`, так и для класса `OfficeVisitRecord`. Фактически это означает, что классы `SurgicalRecord` и `OfficeVisitRecord` содержат все методы и свойства класса `MedicalRecord` плюс собственные компоненты, которые явно указаны в соответствующих прямоугольниках. Таким образом, классы `SurgicalRecord` и `OfficeVisitRecord` содержат свойства `patient`, `doctor` и `dateOfRecord`, но класс `SurgicalRecord` также содержит свойства `surgicalProcedure`, `hospital` и `dateOfDischarge` плюс метод `dischargePatient()`, тогда как класс `OfficeVisitRecord` содержит собственные свойства `symptoms` и `diagnosis`. Кроме того, все три класса включают метод `printRecord()`, позволяющий распечатать медицинскую карту. При этом

методы `printRecord()` в классах `SurgicalRecord` и `OfficeVisitRecord` являются специализациями метода `printRecord()` из класса `MedicalRecord`, поскольку каждый из них дополнительно будет печатать информацию, специфическую для его класса.

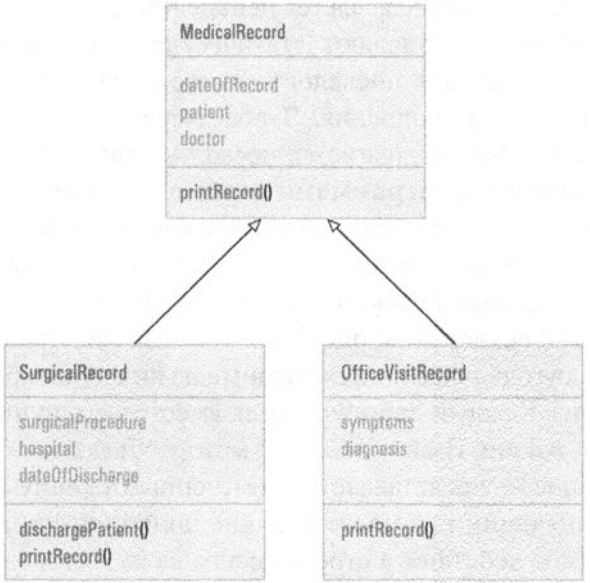


Рис. 7.12. Диаграмма классов с указанием отношений обобщения

Вспомните из главы 6 (раздел 6.5), что естественным способом реализации обобщений в среде объектно-ориентированного программирования является использование механизма наследования. Однако многие разработчики программного обеспечения предупреждают, что наследование не может использоваться для всех случаев обобщения. Причина в том, что наследование существенно повышает уровень связанности между классами — связанности, которая может оказаться нежелательной на более позднем этапе жизненного цикла программного обеспечения. Например, поскольку изменения внутри класса автоматически отражаются во всех унаследованных от него классах, кажущееся незначительным изменение, внесенное на этапе сопровождения программы, может привести к непредвиденным последствиям. В качестве примера предположим, что компания для своих сотрудников открыла базу отдыха, а это означает, что все люди, имеющие право пользоваться базой отдыха, являются сотрудниками компании. Чтобы получить список всех тех, кто имеет право пользоваться базой отдыха, программист мог бы воспользоваться механизмом наследования для создания класса `RecreationMember` на основе уже определенного ранее класса `Employee`. Однако если впоследствии компания преуспеет и решит сделать базу отдыха доступной и для членов семей сотрудников или, допустим, для

пенсионеров компании, то встроенную связь между классом `Employee` и классом `RecreationMember` потребуется разорвать. Таким образом, использовать механизм наследования просто для удобства недопустимо. В действительности применение этого механизма должно быть ограничено только теми случаями, в которых реализуемое обобщение является неизменным.

Диаграммы классов представляют статические свойства проектируемой программы. Они не отражают последовательности событий, которые будут происходить во время ее выполнения. Чтобы выразить такие динамические особенности, в языке UML предлагается несколько типов диаграмм, которые в совокупности называют **диаграммами взаимодействия**. Одним из типов диаграмм взаимодействия является **диаграмма последовательности**, отображающая связь между объектами (такими, как актеры, полные программные компоненты или отдельные объекты), которые участвуют в выполнении задачи. Эти диаграммы аналогичны рис. 7.5 в том смысле, что объекты на них представлены прямоугольниками с исходящими из них пунктирными линиями, направленными вниз. Каждый прямоугольник вместе с его пунктирной линией называется **линией жизни**. Взаимодействие между объектами — например, запрос на обслуживание — представляется помеченными линиями со стрелками (сигналами), соединяющими соответствующие линии жизни, где метка определяет запрашиваемое действие, а стрелка направлена в сторону того объекта, которому направлен запрос. Эти линии размещаются на диаграмме в хронологическом порядке, предполагающем чтение диаграммы сверху вниз. Взаимодействие, которое имеет место, когда объект завершает запрошенную задачу и возвращает управление запросившему ее выполнение объекту, как и в традиционном возврате из процедуры, представляется немаркированной стрелкой, направленной в сторону линии жизни источника запроса.

Таким образом, рис. 7.5, по сути, является диаграммой последовательности. Однако синтаксису, использованному на рис. 7.5, свойственно несколько недостатков. Во-первых, он не позволяет уловить симметрию между двумя игроками. Потребуется нарисовать отдельную диаграмму, чтобы представить на ней розыгрыш мяча, начинающийся с подачи игрока B, даже несмотря на то что эта последовательность взаимодействий будет очень похожа на последовательность, когда подает игрок A. Более того, в то время как на рис. 7.5 показан один конкретный розыгрыш мяча, в общем случае этот розыгрыш может продолжаться бесконечно. Формальные диаграммы последовательности предлагают способы представления подобных нюансов на одной диаграмме, и хотя нам нет необходимости подробно их изучать, будет полезно просто взглянуть на формальную диаграмму последовательности, показанную на рис. 7.13, которая представляет обобщенный розыгрыш мяча в терминах нашего проекта игры в теннис.

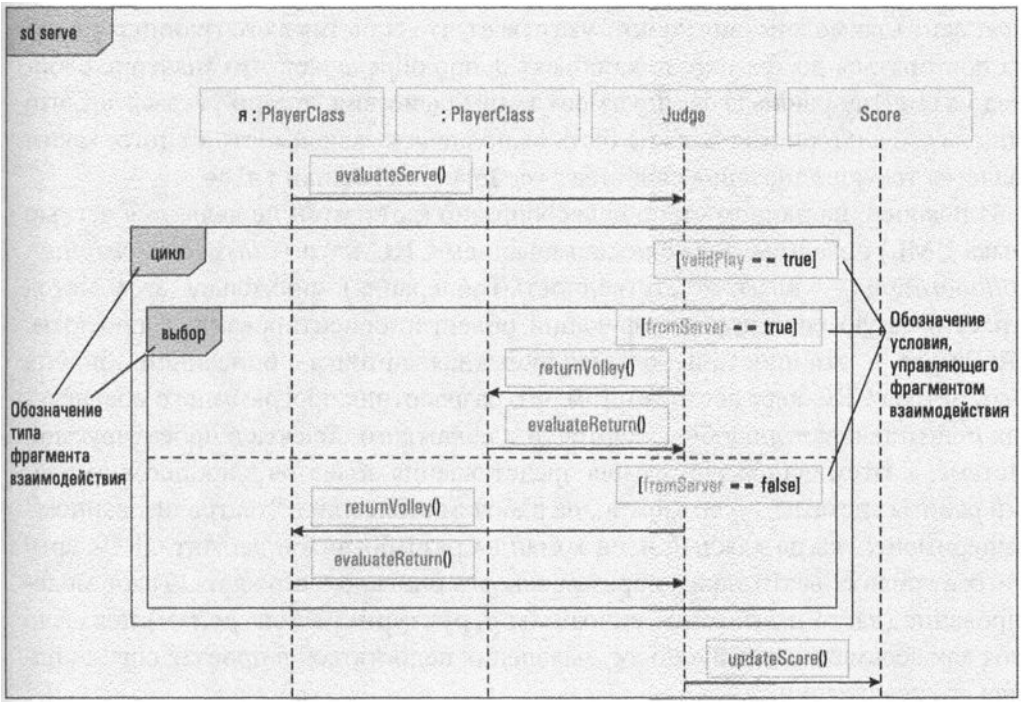


Рис. 7.13. Диаграмма последовательности, представляющая общую схему розыгрыша мяча

Обратите внимание: на рис. 7.13 видно, что вся диаграмма последовательности заключена в прямоугольник (называемый **фреймом**). В верхнем левом углу этой рамки находится пятиугольник, содержащий символы **sd** (означающие "диаграмма последовательности" — *sequence diagram*), за которыми следует идентификатор. Этот идентификатор может быть именем, идентифицирующим общую последовательность, или, как на рис. 7.13, именем метода, который вызывается для инициирования последовательности. Обратите внимание, что в отличие от рис. 7.5, прямоугольники, представляющие на рис. 7.13 игроков, не относятся к конкретным игрокам, а просто указывают, что они представляют объекты "типа" `PlayerClass`. Один из них обозначен как "**я**": это означает, что это тот объект, метод подачи которого был активирован для инициирования данной последовательности.

Другое важное замечание относительно рис. 7.13 касается двух внутренних прямоугольников. Это **фрагменты взаимодействия**, которые используются для представления альтернативных последовательностей на одной диаграмме. На рис. 7.13 присутствуют два фрагмента взаимодействия. Один из них помечен как "**цикл**", другой — как "**выбор**". По сути, это структуры `while` и `if-else`, с которыми вы уже познакомились при обсуждении языка Python в разделе 5.2.

Фрагмент взаимодействия "**цикл**" указывает, что события в его границах должны повторяться до тех пор, пока объект `Judge` определяет, что значение свойства `validPlay` равно `true`. Фрагмент взаимодействия "**выбор**" указывает, что одна из его альтернатив должна быть выполнена в зависимости от того, каким является текущее значение свойства `fromServer`: `true` или `false`.

И наконец, на данном этапе целесообразно (хотя это и не являются частью языка UML) познакомиться с использованием **CRC-карт** (*Class-Responsibility-Collaboration* — класс-ответственность-кооперация), поскольку этот метод играет важную роль при верификации объектно-ориентированных проектов. CRC-карта — это просто небольшая бумажная карточка с описанием объекта. Суть метода CRC-карт состоит в том, что разработчик программного обеспечения подготавливает подобные карточки для каждого объекта в проектируемой системе, а затем использует их для представления объектов в процессе имитации работы системы, — возможно, на рабочем столе или в "театрализованном" эксперименте, когда каждый член команды разработчиков держит CRC-карту и играет роль объекта таким образом, как это описано в его карте. Такое моделирование (часто называемое **сквозным структурным контролем**) показало себя как весьма полезный способ выявления недостатков в проекте еще до начала его реализации.

Шаблоны проектирования

Для разработчиков программного обеспечения важным инструментом, возможности которого со временем только увеличиваются, является постоянно расширяющаяся коллекция шаблонов проектирования. **Шаблон проектирования** — это заранее разработанная архитектурная структура, предназначенная для решения регулярно возникающей проблемы при разработке программного обеспечения. Например, шаблон проектирования **Адаптер** предоставляет способ решения проблемы, которая часто возникает при создании программного обеспечения из готовых модулей. В частности, готовый модуль может иметь функциональные возможности, необходимые для решения рассматриваемой задачи, но при этом может не иметь интерфейса, совместимого с разрабатываемым приложением. В таких случаях шаблон **Адаптер** обеспечивает стандартный подход в виде "обертывания" требуемого модуля другим модулем, обеспечивающим трансляцию сигналов между интерфейсом требуемого модуля и внешним миром, что и позволяет использовать готовый модуль в приложении.

Другим известным шаблоном проектирования является шаблон **Декоратор**. Он предлагает способ разработки системы, которая выполняет различные комбинации одних и тех же действий в зависимости от текущей ситуации. Подобные системы могут требовать реализации бесчисленного множества вариантов

действий, что без тщательного проектирования часто приводит к созданию чрезвычайно сложного программного обеспечения. Однако использование шаблона **Декоратор** позволяет реализовать такую систему стандартным способом и получить полностью контролируемое решение.

Выявление повторяющихся проблем, а также создание и каталогизация шаблонов проектирования, предназначенных для их решения, является непрекращающимся процессом в области разработки программного обеспечения. Однако задача здесь состоит не в том, чтобы просто найти решения определенных проблем в проектировании, а в том, чтобы найти *высококачественные* решения, обеспечивающие необходимую гибкость на более поздних этапах жизненного цикла программного обеспечения. По этой причине соглашения о хороших принципах проектирования, таких как минимизация связывания и максимизация связности, играют важную роль в разработке шаблонов проектирования.

Результаты прогресса в разработке шаблонов проектирования находят свое отражение в библиотеках инструментов, доступных в современных пакетах разработки программного обеспечения, таких как среда программирования языка Java, предоставляемая корпорацией Oracle, или среда программирования .NET Framework, предоставляемая корпорацией Microsoft. В действительности многие из "шаблонов", которые можно найти в подобных "наборах инструментов", по сути являются каркасами шаблонов проектирования, обеспечивающих получение готовых высококачественных решений для различных задач проектирования.

7.5. Вопросы и упражнения

1. Нарисуйте диаграмму потоков данных, представляющую тот поток данных, который возникает, когда читатель берет книгу в библиотеке.
2. Нарисуйте диаграмму вариантов использования в системе ведения записей библиотеки.
3. Нарисуйте диаграмму классов, представляющую отношения между путешественниками и гостиницами, в которых они проживают.
4. Нарисуйте диаграмму классов, отражающую тот факт, что человек является обобщением работника. Включите некоторые атрибуты, которые могут принадлежать каждому.
5. Преобразуйте рис. 7.5 в полную диаграмму последовательности.
6. Какую роль в процессе разработки программного обеспечения играют шаблоны проектирования?

В заключение следует отметить, что появление шаблонов проектирования в области разработки программного обеспечения является примером того, как различные области человеческой деятельности могут оказывать плодотворное влияние друг на друга. Истоки идеи применения шаблонов проектирования лежат в исследованиях Кристофера Александера в области традиционной архитектуры. Его целью было выявление элементов, которые способствуют созданию высококачественных архитектурных проектов для зданий или комплексов зданий, с последующей разработкой шаблонов проектирования, включающих эти элементы. Сегодня многие из его идей приняты и в области разработки программного обеспечения, а его работа продолжает вдохновлять многих разработчиков в этой области.

7.6. Обеспечение качества программ

Частые случаи неправильного функционирования программного обеспечения, перерасхода средств и несоблюдения сроков реализации проектов требуют совершенствования методов контроля качества программного обеспечения. В этом разделе мы рассмотрим некоторые направления, активно изучаемые в этой области.

Область применения средств обеспечения качества ПО

В начальный период развития вычислительной техники проблема создания качественного программного обеспечения воспринималась как устранение в программах ошибок, возникавших в процессе их реализации. Далее в этом разделе мы обсудим прогресс, достигнутый в этом направлении. Однако сегодня сфера обеспечения качества программного обеспечения вышла далеко за рамки процесса отладки и включает такие направления, как улучшение процедур разработки программного обеспечения, разработку учебных программ, которые во многих случаях приводят к сертификации, и установление стандартов, на которых строится разработка качественного программного обеспечения. В этой связи выше уже отмечалась важная роль таких организаций, как ISO, IEEE и ACM в повышении профессионализма и внедрении стандартов оценки контроля качества в компаниях, занимающихся разработкой программного обеспечения. Конкретным примером являются стандарты серии ISO 9000, касающиеся многих видов промышленной деятельности, таких как проектирование, производство, монтаж и обслуживание. Другим примером являются стандарты ISO/IEC 33001, представляющие собой серию стандартов, совместно разработанных ISO и Международной электротехнической комиссией (IEC — *International Electrotechnical Commission*).

Многие крупные поставщики программного обеспечения в настоящее время требуют, чтобы организации, которые они нанимают для разработки программного обеспечения, соответствовали таким стандартам. В результате компании-разработчики программного обеспечения создают **группы обеспечения качества** (SQA — *Software Quality Assurance*) программного обеспечения, в обязанности которых входит контроль и обеспечение соблюдения требований систем контроля качества, принятых в организации. Таким образом, в случае традиционной модели водопада группе SQA будет поручено утвердить спецификацию требований к программному обеспечению до начала этапа проектирования или утвердить разработанный проект и связанные с ним документы до начала реализации.



Основные положения для запоминания

- **Сотрудничество позволяет упростить поиск и исправление ошибок при разработке программ.**

Сегодня в основу прилагаемых усилий по контролю качества положено несколько направлений. Одним из них является учет. Крайне важно, чтобы каждый шаг в процессе разработки был точно задокументирован для использования в будущем. Однако эта цель не соответствует природе человека: всегда имеет место большой соблазн принимать или изменять решения без обновления соответствующих документов. В результате есть определенная вероятность того, что записи будут неправильными, а следовательно, их использование на будущих этапах может вводить в заблуждение. В этом заключается важное преимущество CASE-инструментов. Они значительно упрощают выполнение таких заданий, как перерисовка диаграмм или обновление словаря данных, в сравнении с ручными методами. Следовательно, обновления, скорее всего, будут сделаны, и окончательная документация, скорее всего, будет точной. (Этот пример является лишь одним из многих случаев, когда технология разработки программного обеспечения должна обеспечивать преодоление недостатков, свойственных природе человека. Другие такие недостатки включают личностные конфликты, ревность и столкновение интересов исполнителей, которые неизбежно возникают, когда люди работают вместе.)

Еще одна тема, связанная с обеспечением качества ПО, — это проведение аудитов, во время которых различные стороны, участвующие в проекте разработки программного обеспечения, собираются для обсуждения конкретной темы. Такие аудиты проводятся на протяжении всего процесса разработки программного обеспечения с целью анализа требований, анализа проектного решения и

анализа его реализации. Они могут иметь форму демонстрации и обсуждения прототипа на ранних этапах анализа требований, сквозного структурного контроля, осуществляемого силами членов команды разработчиков программного обеспечения, или координационных встреч между программистами, реализующими взаимосвязанные части проекта. Такие встречи и просмотры, проводимые на регулярной основе, обеспечивают каналы коммуникации, благодаря которым удается избежать недопонимания и исправить ошибки еще до того, как они приведут к катастрофическим последствиям. Значимость таких встреч подтверждается тем фактом, что они специально рассматриваются в стандарте IEEE по вопросу аудита программного обеспечения, известном как IEEE 1028.

Некоторые аудиты и просмотры имеют ключевое значение. Примером является встреча представителей заказчиков проекта и команды разработчиков программного обеспечения, на которой утверждается окончательная спецификация требований к программному обеспечению. И действительно, утверждение подобного документа знаменует собой конец фазы формального анализа требований и закладывает основу, на которой будет строиться вся дальнейшая работа над проектом. Тем не менее все проверки, аудиты и просмотры важны и для контроля качества обязательно должны быть задокументированы как часть постоянного процесса ведения документации.

Тестирование программного обеспечения

Хотя обеспечение качества программного обеспечения в настоящее время признано в качестве предмета, пронизывающего весь процесс разработки ПО, тестирование и проверка самих программ также продолжает оставаться постоянным предметом исследований. В разделе 5.6 рассматривались методы верификации алгоритмов в строгом математическом смысле. Однако было сделано заключение, что бо́льшая часть создаваемого сегодня программного обеспечения все еще "верифицируется" посредством тестирования. К сожалению, тестирование в лучшем случае можно расценивать как неточную науку. Проведя тестирование, мы не можем утверждать, что некоторая часть программы правильна, если не было выполнено достаточно проверок, чтобы исчерпать все возможные сценарии. Но даже для простых программ могут существовать миллиарды возможных путей ее выполнения. А это означает, что тестирование всех возможных путей выполнения достаточно сложной программы — задача просто невыполнимая.

С другой стороны, создатели программного обеспечения разработали методы тестирования программ, повышающие вероятность обнаружения существующих в них ошибок с использованием ограниченного количества тестов. Один из таких методов основан на наблюдении, что ошибки в программном

обеспечении имеют тенденцию собираться в группы. Иначе говоря, опыт показывает, что в крупной системе программного обеспечения всегда существует небольшое число модулей, являющихся более проблематичными, чем все остальные. Следовательно, обнаружив эти модули и проверив их более тщательно, можно выявить бо́льшую часть существующих в системе ошибок, даже если все остальные модули будут протестированы обычным, менее тщательным образом. Это предположение часто называют **принципом Парето**, в честь экономиста и социолога Вильфредо Парето (Vilfredo Pareto, 1848–1923), который заметил, что малая часть населения Италии контролирует бо́льшую часть богатств этой страны. В области разработки программного обеспечения принцип Парето утверждает, что желаемого результата часто можно достичь быстрее, если приложить больше усилий в областях концентрации ошибок.

Еще один метод тестирования ПО, называемый **тестированием основных путей**, состоит в разработке набора контрольных данных, гарантирующего, что каждая инструкция в программе будет выполнена хотя бы один раз. Для подготовки таких наборов были разработаны методы, построенные на основе математической теории графов. Поэтому, хотя и нельзя будет утверждать, что все возможные пути выполнения в системе программного обеспечения будут проверены, можно гарантировать, что в процессе тестирования каждая инструкция в программном коде системы будет выполнена как минимум один раз.

Методы, основанные на принципе Парето, и способ тестирования основных путей предполагают знание внутренней структуры тестируемого программного обеспечения. Следовательно, они относятся к категории тестирования по принципу "**прозрачного ящика**", при котором подразумевается, что внутреннее устройство программного обеспечения известно тестирующему и он использует это знание при разработке тестов. В противоположность этому при тестировании по принципу "**черного ящика**" выполняемая проверка не может основываться на знании внутренней структуры тестируемого программного обеспечения. Короче говоря, тестирование по принципу черного ящика выполняется с точки зрения пользователя системы. В этом случае анализируется не то, как именно программа функционирует при решении задачи, а исключительно то, насколько правильно она работает в смысле точности достигнутых результатов и скорости ее выполнения.

Один из методов, которые обычно относятся к концепции тестирования по принципу черного ящика, именуется **анализом граничных условий**. Он предполагает определение диапазонов данных, называемых **классами эквивалентности**, в которых программное обеспечение должно работать единообразно, и тестирование программного обеспечения на данных, близких к границам этих диапазонов. Например, если предполагается, что в программе допускается введение исходных значений только из конкретно заданного диапазона, работу

программы следует проверить при вводе наименьшего и наибольшего значений из допустимого диапазона. Если же программное обеспечение должно координировать множество различных действий, то его работу следует проверять с использованием множества действий, имеющих максимальные требования к системе. Положенная в основу этого метода теория заключается в том, что путем идентификации классов эквивалентности количество тестовых случаев можно минимизировать, поскольку правильная работа для нескольких примеров в классе эквивалентности может рассматриваться как успешная проверка программного обеспечения для всего класса. Более того, наилучший шанс идентифицировать ошибку в классе — использовать данные на границах класса.

Еще одним методом тестирования по принципу черного ящика является **бета-тестирование**, при котором предварительная версия программного обеспечения предоставляется сегменту целевой аудитории с целью изучения работы этого ПО в реальных условиях, прежде чем окончательная версия данного программного продукта будет утверждена и выпущена на рынок. (Подобное тестирование, проводимое на сайте разработчика, называется **альфа-тестированием**.) Преимущества бета-тестирования выходят далеко за рамки традиционного обнаружения ошибок. Получение общих отзывов от потенциальных клиентов (как положительных, так и отрицательных) может оказать существенную помощь в уточнении рыночных стратегий. Более того, раннее распространение

7.6. Вопросы и упражнения

1. Какова роль группы обеспечения качества (SQA) в организации по разработке программного обеспечения?
2. Как человеческая природа работает против обеспечения качества ПО?
3. Определите два типа мероприятий, которые применяются в процессе разработки для повышения качества.
4. Какая проверка при тестировании программного обеспечения является успешной, нашедшая или не нашедшая ошибки?
5. Какие методы можно предложить для обнаружения в системе модулей, которые необходимо тестировать более тщательно, чем все остальные?
6. Что можно считать хорошим тестом для проверки работы пакета программного обеспечения, разработанного для сортировки списка, содержащего не более 100 элементов?

бета-версии программного обеспечения помогает сторонним разработчикам ПО в разработке своих совместимых продуктов. Например, в случае новой операционной системы для рынка настольных ПК распространение бета-версии стимулирует разработку разнообразного совместимого программного обеспечения, так что окончательная версия операционной системы появится на полках магазинов уже в окружении сопутствующих программных продуктов. Более того, наличие бета-тестирования помогает создать на рынке атмосферу предвкушения, которая повышает популярность продукта, а значит, увеличивает объем его продаж.

7.7. Документирование программного обеспечения

Программная система будет бесполезна, пока люди не научатся ее использовать и обслуживать. Следовательно, документация является важной частью конечного программного продукта, а ее разработка — важная тема в технологии разработки программного обеспечения.

Документация по программному обеспечению служит трем целям, что ведет к трем категориям создаваемых документов: пользовательская документация, системная документация и техническая документация. Назначение **пользовательской документации** состоит в объяснении возможностей программного обеспечения и описании того, как их использовать. Она предназначена для чтения пользователем программного обеспечения и поэтому выражается в терминологии приложения, т.е. должна носить нетехнический характер.

Сегодня документация пользователя считается важным инструментом маркетинга. Хорошая пользовательская документация в сочетании с хорошо разработанным интерфейсом пользователя делает программный пакет более доступным и, следовательно, способствует увеличению объема его продаж. Осознавая это, многие разработчики программного обеспечения для выполнения данной части проекта нанимают известных авторов технической литературы или представляют предварительные версии своих продуктов независимым авторам в надежде, что соответствующие книги уже поступят в продажу к тому моменту, когда само программное обеспечение появится на рынке.

Документация пользователя традиционно имеет форму учебного руководства в виде обычной книги или буклета, но все чаще доступна в электронном виде в Сети или как часть самого программного обеспечения. Это дает пользователю возможность обращаться к документации непосредственно при использовании программного обеспечения. В этом случае информация может быть разбита на небольшие блоки, иногда называемые страницами справки, которые могут автоматически отображаться на экране, если пользователь слишком долго раздумывает при переходе от одной команды к другой.

Назначение **системной документации** состоит в описании внутренней структуры программного обеспечения, чтобы обеспечить возможность сопровождения системы на более позднем этапе ее жизненного цикла. Основным компонентом системной документации является исходная версия всех программ системы. Очень важно, чтобы эти программы были представлены в читабельном виде, поэтому разработчики программного обеспечения используют хорошо разработанные языки программирования высокого уровня, сопровождают текст программы комментариями, а также применяют технологию модульного проектирования, что позволяет представить каждый модуль как отдельный, согласованно работающий элемент. На практике многие компании, выпускающие программное обеспечение, приняли ряд правил, которым должны следовать их работники при написании программ. Сюда входят соглашения об использовании отступов в исходных текстах программ; соглашения о присвоении имен, устанавливающие различия между именами переменных, констант, объектов, классов и т.д.; и соглашения по документированию, гарантирующие, что все написанные программы будут достаточно документированы. Подобные соглашения обеспечивают единообразие создаваемого программного обеспечения в рамках всей компании, что существенно упрощает процесс его сопровождения.



Основные положения для запоминания

- **Программная документация помогает программистам создавать программы и поддерживать их корректность, обеспечивая эффективное решение возникающих проблем.**

Другим компонентом системной документации является проектная документация, включая спецификацию требований к программному обеспечению и записи о том, как эти спецификации были получены во время проектирования. Эта информация будет полезна при модификации программного обеспечения, поскольку указывает, почему программное обеспечение было реализовано именно таким, каким оно есть, — эта информация позволяет уменьшить вероятность того, что изменения, внесенные во время модификации программы, нарушат целостность системы.

Назначением **технической документации** является описание того, как данную систему программного обеспечения следует устанавливать и обслуживать (например, настраивать параметры ее функционирования, устанавливать обновления и сообщать о проблемах разработчику этого ПО). Техническая документация программного обеспечения аналогична документации, предоставляемой механикам в автомобильной промышленности. В этой документации

не сообщается, как автомобиль был спроектирован и построен (подобно системной документации), и не объясняется, как управлять автомобилем и пользоваться его системой обогрева/охлаждения (подобно пользовательской документации). Вместо этого в ней описывается, как обслуживать компоненты автомобиля, например как заменить коробку передач или отследить возникающую время от времени проблему в электропроводке.

В сфере персональных компьютеров различия между технической документацией и пользовательской документацией довольно размыты, поскольку их пользователь часто является одновременно и тем человеком, который устанавливает и обслуживает программное обеспечение. Однако в многопользовательской среде это различие выражено заметнее. Здесь техническая документация предназначена прежде всего для системного администратора, который отвечает за обслуживание всего ПО, установленного в сфере его обслуживания, что обеспечивает пользователям возможность доступа к программным пакетам как к абстрактным инструментам.

7.7. Вопросы и упражнения

1. Какие существуют формы документации на программное обеспечение?
2. На какой фазе (или фазах) жизненного цикла программного обеспечения создают его документацию?
3. Что важнее, программа или ее документация?

7.8. Интерфейс "человек–машина"

Вспомним из материала раздела 7.2, что одной из задач при анализе требований является определение того, как создаваемая система программного обеспечения будет взаимодействовать со своей средой. В этом разделе мы рассмотрим темы, связанные с этим взаимодействием, когда оно включает в себя общение с людьми — вопрос, имеющий очень большое значение. В конце концов, люди должны получить возможность использовать программную систему как абстрактный инструмент. И этот инструмент должен быть прост в обращении и предназначен для минимизации (в идеале — для исключения) ошибок коммуникации между системой и ее пользователями-людьми. Это означает, что интерфейс системы должен быть спроектирован для удобства *людей*, а не просто для удобства разработки программной системы.

Важность хорошего дизайна интерфейса дополнительно подчеркивается тем фактом, что системный интерфейс может произвести на пользователя более сильное впечатление, чем любая другая характеристика системы. В конце концов, человек склонен рассматривать систему с точки зрения удобства ее использования, а не с точки зрения того, насколько умно она решает свои внутренние задачи. Для человека выбор между двумя конкурирующими системами, вероятнее всего, будет основан на особенностях интерфейсов этих систем. Следовательно, дизайн интерфейса системы может в конечном итоге стать определяющим фактором успеха или неудачи всего проекта разработки программного обеспечения.

По этим причинам интерфейс "человек–машина" стал важной проблемой на этапе разработки требований к проектам создания ПО и является постоянно расширяющейся областью в сфере разработки программного обеспечения. Фактически некоторые даже утверждают, что изучение интерфейсов "человек–машина" — это вообще отдельная область компьютерных наук.

Наибольшую выгоду в результате исследований, проведенных в этой области, получил интерфейс смартфона. Чтобы достичь поставленных целей и получить удобное устройство карманного размера, все элементы традиционного интерфейса "человек–машина" (полноразмерная клавиатура, мышь, полосы прокрутки, меню) были заменены новыми решениями, такими как жесты, выполняемые на сенсорном экране, голосовые команды и виртуальные клавиатуры с расширенным автозаполнением слов и фраз. Хотя все это в целом уже представляет собой значительный прогресс, большинство пользователей смартфонов утверждают, что есть еще много возможностей для дальнейших инноваций.

Исследования в области дизайна интерфейса "человек–машина" в значительной степени опираются на области инженерии, называемые **эргономикой**, которая отвечает за проектирование систем, соответствующих физическим способностям человека, и **когнетикой**, которая отвечает за проектирование систем, которые гармонируют с интеллектуальными способностями людей. Из этих двух эргономика понимается лучше, в основном потому, что люди веками физически взаимодействовали с машинами. Соответствующие примеры можно найти в древних инструментах, оружии и транспортных системах. Большая часть этой истории самоочевидна, однако иногда применение эргономики бывало и нелогичным. Часто цитируемым примером является конструкция клавиатуры пишущей машинки (которая теперь перевоплотилась в клавиатуру компьютера), в которой клавиши преднамеренно были расположены таким образом, чтобы уменьшить скорость работы машинистки, поскольку механическая система рычагов, использовавшаяся в ранних машинах, при увеличении скорости могла заедать.

Интеллектуальное (или ментальное) взаимодействие с машинами, напротив, является относительно новым явлением, и следовательно, именно когнетика предлагает более высокий потенциал для плодотворных исследований и проливающих свет идей. Зачастую находки в этой области оказываются интересны своей тонкостью. Например, люди формируют привычки — на первый взгляд, это хорошая черта, потому что она может способствовать повышению эффективности. Но привычки могут приводить к ошибкам, даже если дизайн интерфейса, безусловно, позволяет решить проблему. Рассмотрим процесс, когда человек просит типичную операционную систему удалить файл. Чтобы избежать непреднамеренного удаления, в большинстве системных интерфейсов в ответ на такой запрос пользователя просят подтвердить его намерения, например, с помощью сообщения "Вы действительно хотите удалить этот файл?" На первый взгляд, такое обязательное подтверждение требования пользователя, казалось бы, должно решить любую проблему непреднамеренного удаления файла. Однако после достаточно продолжительного периода использования системы у человека вырабатывается привычка автоматически отвечать на этот вопрос утвердительно. В результате задача удаления файла перестает быть двухэтапным процессом, состоящим из ввода команды удаления и предоставления обдуманного ответа на вопрос, заданный системой. Вместо этого он превращается в одноэтапный процесс "удалить–да", означающий, что к тому времени, когда человек поймет, что он выдал ошибочную команду на удаление, запрос от системы уже будет подтвержден, а удаление выполнено.

Формирование привычек может также вызвать проблемы, если человек постоянно использует несколько пакетов прикладного программного обеспечения. Интерфейсы таких пакетов могут быть похожими, но все же разными. В результате одни и те же действия пользователя могут привести к различной реакции со стороны каждой из программ или, наоборот, одинаковые запросы от разных программ могут требовать различных действий пользователя. В подобных случаях привычки, выработанные в одном приложении, могут приводить к ошибочным результатам в других приложениях.

Другой человеческой характеристикой, которая привлекает внимание исследователей в области проектирования интерфейса "человек–машина", является узость человеческого внимания, которое становиться все более сфокусированным при повышении сосредоточенности. По мере того, как человек оказывается все более поглощенным решаемой задачей, разрушить это фокусирование становится все труднее. В 1972 году коммерческий самолет потерпел крушение из-за того, что пилоты настолько погрузились в проблему с шасси (фактически речь идет о замене лампочки индикатора шасси), что позволили самолету врезаться в землю, даже несмотря на то, что в кабине громко звучал сигнал, предупреждающий их об опасности.

Менее критичные примеры часто имеют место в интерфейсах настольных компьютеров. Например, на большинстве клавиатур предусмотрен индикатор "Caps Lock", указывающий, что клавиатура находится в режиме ввода прописными буквами (т.е. была нажата клавиша <Caps Lock>). Однако, случайно нажав эту клавишу, человек редко замечает изменение состояния соответствующего индикатора: он не реагирует на него, пока на экране не начнут появляться совсем не те символы, которые он ожидает. И даже в этом случае пользователь часто приходит в недоумение, не сразу осознав причину возникновения проблемы. В некотором смысле это не удивительно — подсвеченный индикатор клавиатуры, как правило, оказывается вне основного поля зрения пользователя. Тем не менее пользователи часто не замечают и те индикаторы, которые расположены у них непосредственно на виду. Так, пользователь может настолько увлечься выполнением текущей задачи, что просто не обратит внимания на изменение внешнего вида курсора на экране дисплея, даже если в этой задаче предполагается наблюдение за видом курсора.

Еще одна человеческая черта, которую следует учитывать при проектировании интерфейса, — это ограниченная способность сознания работать с несколькими фактами одновременно. В статье, опубликованной в журнале "Психологический обзор" в 1956 году, Джордж А. Миллер сообщил о проведенных им исследованиях, показавших, что человеческий разум способен удерживать внимание только примерно на семи элементах одновременно. Следовательно, когда требуется принятие решения, очень важно, чтобы интерфейс был спроектирован так, чтобы представлять пользователю всю необходимую информацию, не полагаясь на его память. В частности, следует считать очень плохим решением требовать от пользователя запоминания точных деталей из информации, выведенной на предыдущих экранах. Более того, когда интерфейс требует обширной навигации между многочисленными изображениями, человек может просто потеряться в лабиринте. А это означает, что назначение и расположение изображений на экране является важной проблемой дизайна.

Хотя требования эргономики и когнетики придают некоторые уникальные черты дизайну интерфейса человек-машина, эта область также охватывает многие более традиционные аспекты разработки программного обеспечения. В частности, поиск метрик в области дизайна интерфейса так же важен, как и в более традиционных областях разработки программного обеспечения. Характеристики интерфейса, для которых проводились измерения, включают время, необходимое для изучения интерфейса; время, необходимое для выполнения задачи через интерфейс; уровень ошибок интерфейса пользователя; степень, в которой пользователь сохраняет навыки работы с интерфейсом после

перерывов в работе с ним; и даже такие субъективные черты, как степень, в которой интерфейс нравится пользователям.

Модель GOMS (*Goals, Operators, Methods and Selection* — цели, операторы, методы, правила выбора), впервые представленная в 1954 году, является результатом исследований по отысканию метрик в области проектирования интерфейса "человек–машина". Основная методология модели заключается в анализе задач с точки зрения целей пользователя (таких, как удаление слова из текста), операторов (таких, как нажатие кнопки мыши), методов (таких, как двойной щелчок кнопкой мыши или нажатие клавиши удаления) и правил выбора (например, выбора между двумя способами достижения одной и той же цели). Все упомянутое, по сути, и является источником аббревиатуры GOMS — цели, операторы, методы и правила выбора. Коротко говоря, GOMS — это методология, позволяющая анализировать действия использующего интерфейс человека, представленные в виде последовательности элементарных этапов (нажать клавишу, переместить мышь, принять решение). Выполнению каждого элементарного этапа назначается точный период времени, и, таким образом, суммируя время, назначенное всем этапам выполнения задачи, методология GOMS предоставляет средство сравнения различных предлагаемых интерфейсов с точки зрения времени, которое потребуется в каждом из них для выполнения одной и той же задачи.

Однако изучение технических деталей таких систем, как GOMS, вовсе не является целью нашего текущего обсуждения. Эта методология упоминается здесь потому, что она основана на особенностях человеческого поведения — движение рук, принятие решений и т.д. На самом деле методология GOMS изначально вообще рассматривалась как тема из области психологии. И это еще раз подчеркивает ту роль, которую особенности человеческого поведения и восприятия играют в области проектирования интерфейса "человек–машина", даже в таких темах, которые были перенесены из области традиционной разработки программного обеспечения.

Дизайн интерфейсов "человек–машина" обещает оставаться активной областью исследований и в обозримом будущем. Многие проблемы, связанные с современными графическими интерфейсами, все еще не решены, и множество новых сложнейших проблем возникнет при переходе к трехмерным интерфейсам, разработка и использование которых сейчас уже не является далекой перспективой. И действительно, поскольку в этих интерфейсах предполагается объединение аудио- и тактильного взаимодействия с трехмерным изображением, объем потенциальных проблем здесь просто огромен.

7.8. Вопросы и упражнения

1. **а.** Приведите примеры возможности применения эргономики в области проектирования интерфейса "человек-машина".

   б. Приведите примеры возможности применения когнетики в области проектирования интерфейса "человек–машина".

2. Заметным отличием интерфейса "человек–машина" смартфона от интерфейса настольного компьютера являются методы, используемые для прокрутки изображения на экране. В случае настольных компьютеров прокрутка обычно осуществляется перетаскиванием мышью ползунков специальных полос прокрутки, отображаемых справа и внизу в прокручиваем окне, либо с помощью колесика прокрутки, встроенного непосредственно в мышь. С другой стороны, на смартфонах полосы прокрутки чаще всего не используются. (А если они используются, то обычно отображаются в виде тонких линий, предназначенных лишь для указания, какая часть просматриваемого окна видна на экране в данный момент.) Прокрутка в данном случае осуществляется особым жестом "смахивания" — скользящего касания вверх или вниз по экрану дисплея.

   а. Какие аргументы могут быть приведены в поддержку этого различия из соображений эргономики?

   б. Какие аргументы могут быть приведены в поддержку этого различия из соображений когнетики?

3. Что отличает область проектирования интерфейса "человек–машина" от более традиционной области разработки программного обеспечения?

4. Укажите три характеристики человека, которые следует учитывать при разработке интерфейса "человек–машина".

7.9. Право собственности и ответственность за создаваемое программное обеспечение

Не вызывает сомнения, что компания или отдельный человек должен иметь возможность возмещать затраты и получать доходы от тех инвестиций, которые потребовались для разработки качественного программного обеспечения. В противном случае маловероятно, что многие захотят взять на себя решение

задачи создания программного обеспечения, необходимого нашему обществу. Короче говоря, разработчикам программного обеспечения необходим определенный уровень прав владения на то программное обеспечение, которое они производят.

Юридические усилия по обеспечению такого права собственности подпадают под категорию прав **интеллектуальной собственности**, большая часть которого основана на устоявшихся принципах авторского права и патентного права. Действительно, цель авторского права или патента состоит в том, чтобы позволить разработчику "продукта" реализовать этот продукт (или его части) заинтересованным сторонам с необходимой защитой своих прав собственности. По существу, разработчик продукта (будь то физическое или юридическое лицо) будет отстаивать свое право собственности путем включения заявления об авторских правах во все выполненные им работы, включая спецификации требований, проектную документацию, исходный код, план тестирования и конечный продукт — в каком-либо видимом его месте. В уведомлении об авторских правах четко указываются право собственности, лица, которым разрешено использовать произведение, и другие ограничения. Кроме того, права разработчика формально выражены в юридических терминах в **лицензии на программное обеспечение**.

Лицензия на программное обеспечение — это юридическое соглашение между владельцем и пользователем программного продукта, предоставляющее пользователю определенные разрешения на использование продукта без передачи прав собственности на интеллектуальную собственность. В этих соглашениях подробно изложены права и обязанности обеих сторон. Следовательно, важно внимательно прочитать и понять условия лицензии на программное обеспечение — до того, как оно будет установлено и начнется его использование.

Хотя авторские права и лицензионные соглашения на программное обеспечение предоставляют юридические возможности для запрета прямого копирования и несанкционированного использования программного обеспечения, их, как правило, недостаточно, чтобы помешать другой стороне самостоятельно разработать продукт с почти идентичной функцией. Печально, но за прошедшие годы было много случаев, когда разработчик действительно революционного программного продукта так и не смог полностью воспользоваться своим изобретением (два примечательных примера — электронные таблицы и веб-браузеры). В большинстве случаев другой компании удавалось разработать конкурентоспособный продукт, обеспечивавший ей доминирующую долю на рынке. Правовой путь предотвращения такого вторжения со стороны конкурента находится в патентном праве.

Патентные законы были установлены, чтобы позволить изобретателю получить коммерческую выгоду от изобретения. Чтобы получить патент,

изобретатель должен раскрыть подробности изобретения и продемонстрировать, что оно является новым, полезным и неочевидным для других с аналогичным опытом (требование, которое может оказаться достаточно сложным в случае программного обеспечения). Если патент выдан, изобретателю предоставляется право запретить другим лицам создавать, использовать, продавать или импортировать изобретение в течение ограниченного периода времени, который обычно составляет двадцать лет с даты подачи заявки на патент.

Одним из недостатков использования патентов является то, что процесс получения патента является дорогим и трудоемким (часто он растягивается на несколько лет). В течение этого времени программный продукт может устареть, а до выдачи патента заявитель имеет только сомнительные полномочия для предотвращения присвоения продукта другими лицами.

Важность признания авторских прав, лицензий на программное обеспечение и патентов имеет первостепенное значение в процессе разработки программного обеспечения. При разработке программного продукта инженеры-программисты часто решают включить в него программное обеспечение из других продуктов, будь то весь продукт, подмножество его компонентов или даже части исходного кода, загруженные через Интернет. Однако несоблюдение прав интеллектуальной собственности в таких случаях может привести к огромным денежным штрафам и другим последствиям. Например, в 2004 году малоизвестная компания NPT, Inc. успешно выиграла иск против компании Research In Motion (компания RIM — создатель смартфонов марки BlackBerry), выдвинув ей обвинение в нарушении патента на несколько ключевых технологий, встроенных в системы электронной почты RIM. В судебное постановление было включено решение о приостановке работы служб электронной почты для всех пользователей смартфонов BlackBerry в Соединенных Штатах! В конечном итоге компания RIM достигла компромиссного соглашения о выплате компании NPT компенсации на сумму 612,5 млн. долларов США с целью отменить это решение о приостановке.

И наконец, необходимо также рассмотреть вопрос об ответственности. Чтобы защитить себя от ответственности, разработчики программного обеспечения часто включают в лицензии на программное обеспечение отказы от ответственности, в которых указаны те или иные ограничения в отношении их ответственности за созданный продукт. Такие заявления, как "Компания *X* ни в коем случае не будет нести ответственность за любые убытки, возникшие в результате использования этого программного обеспечения", встречаются очень часто. Однако суды редко признают подобный отказ от ответственности, если истец сможет доказать проявление небрежности со стороны ответчика. Таким образом, дела об ответственности обычно фокусируются на том, проявил ли ответчик уровень тщательности, соответствующий выпущенному им продукту.

Понятно, что уровень тщательности, который будет считаться приемлемым в случае разработки системы обработки текста, может оказаться недостаточным и будет воспринят как недопустимая небрежность в случае разработки программного обеспечения для управления ядерным реактором. Следовательно, одним из лучших способов защиты от претензий по поводу ответственности за программное обеспечение является применение строгих принципов разработки программного обеспечения в процессе разработки создаваемого ПО и обеспечение уровня тщательности разработки, соответствующего области применения создаваемого приложения, а также создание и ведение записей, подтверждающих соблюдение всех этих условий.

7.9. Вопросы и упражнения

1. В чем заключается значение уведомления об авторских правах в технических требованиях, проектной документации, исходном коде и конечном продукте?
2. Каким образом закон об авторском праве и патентное право служат на благо обществу?
3. В каких случаях оговорки об отказе от ответственности не принимаются во внимание судами?

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

1. Приведите пример, каким образом усилия, затраченные при разработке программного обеспечения, могут окупиться позднее, при сопровождении программы.
2. Что такое пошаговая модель?
3. Объясните, как на технологию разработки программного обеспечения влияет отсутствие метрик для измерения точных характеристик программного обеспечения.
4. Ожидаете ли вы, что метрика измерения сложности программной системы будет кумулятивной в том смысле, что сложность полной системы будет суммой сложностей ее частей? Поясните свой ответ.
5. Ожидаете ли вы, что метрика измерения сложности программной системы будет коммутативной в том смысле, что сложность полной системы

будет одной и той же, если она изначально была разработана с функцией *X*, после чего к ней добавили функцию *Y*, либо если она изначально была разработана с функцией *Y*, после чего к ней добавили функцию *X*? Поясните свой ответ.

6. Чем технологии разработки программного обеспечения отличаются от технологий традиционных технических областей, таких как электротехника и машиностроение?

7. **а.** В чем заключаются недостатки использования традиционной модели водопада при разработке программного обеспечения?

   б. В чем заключаются преимущества использования традиционной модели водопада при разработке программного обеспечения?

8. Является ли разработка с открытым исходным кодом методологией типа "сверху вниз" или "снизу вверх"? Поясните свой ответ.

9. Опишите, как использование констант вместо литералов может упростить процесс модификации программного обеспечения.

10. В чем заключается различие между связанностью и связностью модулей? Что следует минимизировать, а что максимизировать и почему?

11. Выберите объект из повседневной жизни и проанализируйте его компоненты с точки зрения функциональной или логической связности.

12. Сравните связанность между двумя программными единицами, достигаемую с помощью команды `goto`, со связанностью, получаемой при использовании механизма вызова процедур.

13. В главе 6 объяснялось, что параметры могут быть переданы в функции по значению или по ссылке. Какой вариант обеспечивает более сложную форму связанности по данным? Поясните свой ответ.

14. Какие проблемы могут возникнуть при модификации ПО, если большая программная система была спроектирована таким образом, что все ее элементы данных являются глобальными?

15. Что в объектно-ориентированной программе означает объявление переменной экземпляра как открытой или закрытой в отношении связанности по данным? Что может служить основанием для предпочтения объявления переменной экземпляра как закрытой?

16. Укажите проблему в отношении связанности по данным, которая может возникнуть в контексте параллельной обработки.

17. Ответьте на следующие вопросы, пользуясь приведенной ниже структурной схемой.

    а. Какому модулю возвращает управление модуль *Y*?

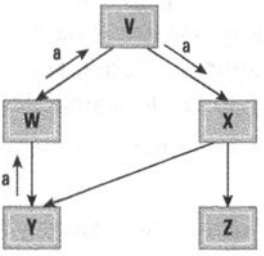
б. Какому модулю возвращает управление модуль *Z*?

в. Являются ли модули *W* и *X* связанными по управлению?

г. Обладают ли модули *W* и *X* связанностью по данным?

д. Какие данные совместно используются модулями *W* и *Y*?

е. В каких отношениях находятся модули *W* и *Z*?



18. Воспользовавшись структурной схемой, представьте процедурную структуру простой системы инвентаризации/учета для небольшого магазина (например, частного магазина сувениров в курортном поселке). Какие модули в вашей системе потребуется модифицировать вследствие внесения изменений в законы о налогах с продаж? Какие модули потребуется изменить, если будет принято решение вести учет клиентов таким образом, чтобы впоследствии им можно было отправлять рекламные материалы по почте?

19. Используя диаграмму классов, разработайте объектно-ориентированное решение для предыдущей задачи.

20. Начертите простую диаграмму классов, представляющую взаимоотношения между издателями, журналами и подписчиками. Достаточно указать только имя класса в каждом поле, представляющем класс.

21. Что такое UML и для чего он используется? Уточните слово, соответствующее в этой аббревиатуре букве "M".

22. Нарисуйте простую диаграмму вариантов использования, представляющую способы, которыми читатель может использовать библиотеку.

23. Нарисуйте диаграмму последовательности, отражающую ту серию взаимодействий, которая возникает, когда коммунальная компания отправляет клиенту счет на оплату услуг.

24. Нарисуйте простую диаграмму потока данных, отображающую поток данных, который происходит в автоматизированной системе складского учета при продаже изделий.

25. Сравните информацию, представленную в диаграмме классов, с информацией, представленной в диаграмме последовательности.

26. В чем разница между отношением "один-ко-многим" и отношением "многие-ко-многим"?

27. Приведите пример отношения "один-ко-многим", который не упомянут в этой главе. Приведите пример отношения "многие-ко-многим", который не упомянут в этой главе.

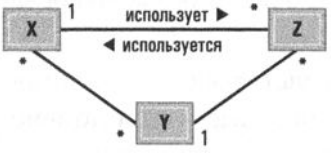
28. Исходя из информации, представленной на рис. 7.10, представьте серию взаимодействий, которая может происходить между врачом и пациентом во время их встречи при обращении пациента за консультацией. Нарисуйте диаграмму последовательности, представляющую эти взаимодействия.

29. Нарисуйте диаграмму классов, представляющую отношения между посетителями и официантами в ресторане.

30. Нарисуйте диаграмму классов, представляющую отношения между журналами, издателями журналов и подписчиками журналов. Для каждого класса укажите необходимый набор переменных экземпляра и методов.

31. Расширьте диаграмму последовательности, представленную на рис. 7.5, чтобы показать последовательность взаимодействий, которая произойдет, если класс `PlayerA` успешно вернет мяч классу `PlayerB`, а класс `PlayerB` не сможет успешно отбить этот мяч.

32. Ответьте на следующие вопросы, основываясь на прилагаемой диаграмме классов, которая представляет связи между инструментами, их пользователями и их производителями.



а. Какие классы (`X`, `Y` и `Z`) представляют на диаграмме инструменты, пользователей и производителей? Обоснуйте свой ответ.

б. Может ли инструмент использоваться более чем одним пользователем?

в. Может ли инструмент быть изготовлен несколькими производителями?

г. Каждый пользователь использует инструменты, изготовленные многими производителями?

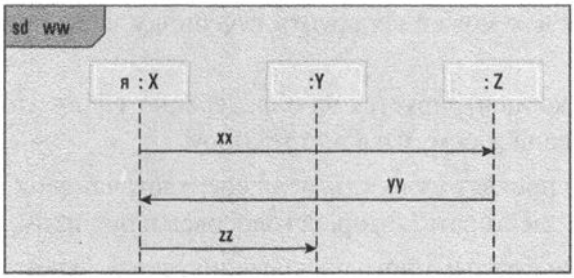
33. В каждом из следующих случаев определите, относится ли действие к диаграмме последовательности, диаграмме вариантов использования или диаграмме классов.

а. Представляет способ взаимодействия пользователей с системой.

б. Представляет отношения между классами в системе.

в. Представляет способ взаимодействия объектов для выполнения некоторой задачи.

34. Ответьте на следующие вопросы, основываясь на прилагаемой диаграмме последовательности.



а. Какой класс содержит метод с именем `ww()`?

б. Какой класс содержит метод с именем `xx()`?

в. В этой последовательности объект "типа" `Z` когда-либо непосредственно связывается с объектом "типа" `Y`?

35. Нарисуйте диаграмму последовательности, показывающую, что объект `A` вызывает метод `bb()` в объекте `B`, объект `B` выполняет запрошенное действие и возвращает управление объекту `A`, а затем объект `A` вызывает метод `cc()` в объекте `B`.

36. Расширьте решение предыдущей задачи таким образом, чтобы указать, что объект `A` вызывает метод `bb()` только в том случае, если переменная `continue` имеет значение `true` и продолжает вызывать метод `bb()` до тех пор, пока в переменной `continue` сохраняется значение `true`, и после того, как объект `B` возвращает управление.

37. Нарисуйте диаграмму классов, отображающую тот факт, что классы `Truck` и `Automobile` являются обобщениями класса `Vehicle`.

38. Исходя из информации, представленной на рис. 7.12, укажите, какие дополнительные переменные экземпляра будут содержаться в объекте "типа" `SurgicalRecord`? А в объекте "типа" `OfficeVisitRecord`?

39. Объясните, почему наследование — не всегда лучшее средство реализации обобщений классов.

40. Укажите какие-либо шаблоны проектирования в других областях, помимо разработки программного обеспечения.

41. Обобщите роль, которую шаблоны проектирования играют в разработке программного обеспечения.

42. В какой степени управляющие структуры, представленные в типичном высокоуровневом языке программирования (`if-else`, `while` и т.д.), можно рассматривать как шаблоны проектирования небольшого масштаба?

43. Какие из приведенных ниже высказываний соответствуют принципу Парето? Объясните свой ответ.

а. Один неприятный человек может испортить вечеринку всем присутствующим.

б. Каждая радиостанция концентрируется на определенном формате, таком как хард-рок, классическая музыка или ток-шоу.

в. На выборах кандидаты проявят мудрость, если сосредоточат свои кампании на том сегменте электората, который голосовал в прошлом.

44. Ожидают ли разработчики программного обеспечения, что большие программные системы будут однородными (либо неоднородными) относительно содержащихся в них ошибок? Поясните свой ответ.

45. В чем различие между тестированиями по принципам "черного ящика" и "прозрачного ящика"?

46. Приведите некоторые аналогии тестирования по принципам "черного ящика" и "прозрачного ящика" в других областях, отличных от разработки программного обеспечения.

47. Чем разработка с открытым исходным кодом отличается от бета-тестирования? (Сравните вариант тестирования по принципу "прозрачного ящика" с тестированием по принципу "черного ящика".)

48. Предположим, что перед окончательным тестированием крупной системы программного обеспечения в нее было намеренно внесено 100 ошибок. Далее допустим, что во время этого тестирования было обнаружено и исправлено 200 ошибок, из которых 50 оказались из группы намеренно внесенных в систему. Если исправить оставшиеся 50 известных ошибок, сколько невыявленных ошибок, по-вашему, еще останется в системе? Объясните, почему.

49. Что такое GOMS?

50. Что такое эргономика? Что такое когнетика?

51. Одно из различий между интерфейсами "человек–компьютер" смартфона и настольного компьютера заключается в методике, используемой для изменения масштаба изображения на экране с целью получения более или менее детального представления данных (процесс, называемый масштабированием). На рабочем столе масштабирование обычно достигается путем перетаскивания ползунка, отдельного от отображаемой области, или с помощью элемента меню или панели инструментов. На смартфоне масштабирование выполняется путем одновременного касания экрана дисплея большим и указательным пальцами, а затем изменения расстояния

между двумя точками касания (процесс, называемый растягиванием для увеличения масштаба или стягиванием — для его уменьшения).

а. Исходя из положений эргономики, какие аргументы могут быть сделаны в поддержку этого различия?

б. Исходя из положений когнетики, какие аргументы могут быть сделаны в поддержку этого различия?

52. В каких случаях существующие законы о защите авторских прав не смогут защитить инвестиции разработчиков программного обеспечения?

53. В каких случаях разработчику программного обеспечения может быть отказано в получении патента?

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также разобраться, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. а. Исполнителю была поставлена задача разработать систему для занесения медицинских записей в машину, соединенную с большой сетью. Его предложения относительно требований безопасности были отклонены по финансовым соображениям, и исполнителю было приказано продолжить работу над проектом, применяя систему безопасности, которую он считал недостаточной. Что ему следует делать и почему?

б. Предположим, что упомянутый выше исполнитель разработал систему так, как ему было приказано, и теперь опасается, что медицинские записи могут подвергаться несанкционированному просмотру. Что ему следует делать? В какой степени он несет ответственность за возможные нарушения безопасности?

в. Допустим, вместо того чтобы подчиниться руководству, исполнитель отказывается от работы с системой и поднимает шум, разглашая недостатки проекта, что приводит к финансовым затруднениям в компании и потере работы многими ни в чем не повинными сотрудниками. Корректны ли действия этого исполнителя? Что если, будучи всего лишь одним из рядовых членов группы разработчиков, он не знал о том, что

в другом подразделении компании предпринимались усилия по разработке действенной системы безопасности, которую предполагалось применить и к его системе. Как это изменит ваше отношение к действиям данного исполнителя? (Помните, что взгляд исполнителя на ситуацию остается таким же, как и раньше.)

2. Как должна распределяться ответственность, если крупная система программного обеспечения разрабатывается многими людьми? Существует ли иерархия ответственности? Существуют ли степени юридической ответственности?
3. Как мы знаем, крупные и сложные системы программного обеспечения часто разрабатываются многими людьми, однако лишь некоторые из них имеют полное представление о существе проекта. Допустимо ли работнику принимать участие в проекте, не имея полных знаний о его назначении и свойствах?
4. До какой степени каждый несет ответственность за то, как его достижения в конечном счете применяются другими людьми?
5. С точки зрения отношений между специалистом в области компьютеров и его клиентом должен ли первый просто реализовывать желания клиента или же скорее направлять его желания? Что если специалист предвидит, что требования клиента могут привести к нежелательным последствиям? Например, клиент может пожелать внести в систему некоторые упрощения для повышения ее эффективности, но специалист может предвидеть, что такие упрощения могут стать потенциальным источником появления ошибочных результатов или злоупотреблений в системе. Если клиент все же настаивает на своем решении, может ли специалист считать себя свободным от ответственности?
6. Что произойдет, если технологии начнут развиваться так быстро, что новые изобретения будут вытеснены новейшими еще до того, как изобретатели успеют извлечь выгоду из своего изобретения? Нужна ли прибыль для мотивации изобретателей? Как с вашим ответом соотнести возможный успех разработки с открытым исходным кодом? Является ли качественное бесплатное программное обеспечение устойчивым элементом реальности?
7. Внесла ли компьютерная революция заметный вклад или хотя бы оказала определенную помощь в решении энергетических проблем в мире? А как насчет других масштабных проблем, таких как голод или бедность?
8. Будут ли достижения в технологии продолжаться бесконечно? В любом случае что могло бы изменить зависимость общества от технологий? Что

может оказаться конечным итогом развития общества, бесконечно продолжающего продвижение технологии?

9. Если бы у вас была машина времени, в каком историческом периоде вы хотели бы жить? Есть ли современные технологии, которые вы хотели бы взять с собой? Можно ли отделить одну технологию от другой? Реально ли протестовать против глобального потепления, но принимать современный уровень медицины и применяемые ею методы лечения?
10. Многие приложения на смартфоне автоматически интегрируются с услугами, предоставляемыми другими приложениями. Эта интеграция может делиться информацией, введенной в одно приложение, с другим. Каковы преимущества этой интеграции? Возможны ли какие-либо проблемы в связи со "слишком большой" интеграцией?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Alexander C., Ishikawa S., Silverstein M. *A Pattern Language*. — New York: Oxford University Press, 1977.
2. Beck K. *Extreme Programming Explained: Embrace Change*, 2nd ed. — Boston, MA: Addison-Wesley, 2004.
3. Bowman D.A., Kruijff E., LaViola J.J., Jr., Poupyrev I. *3D User Interfaces Theory and Practice*. — Boston, MA: Addison-Wesley, 2005.
4. Braude E. *Software Design: From Programming to Architecture*. — New York: Wiley, 2004.
5. Bruegge B., Dutoit A. *Object-Oriented Software Engineering Using UML, Patterns, and Java*, 3rd ed. — Boston, MA: Addison-Wesley, 2010.
6. Cockburn A. *Agile Software Development: The Cooperative Game*, 2nd ed. — Boston, MA: Addison-Wesley, 2006.
7. Fox C. *Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2*. — Boston, MA: Addison-Wesley, 2007.
8. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. — Boston, MA: Addison-Wesley, 1995.
9. Maurer P.M. *Component-Level Programming*. — Upper Saddle River, NJ: Prentice-Hall, 2003.
10. Pfleeger S.L., Atlee J.M. *Software Engineering: Theory and Practice*, 4th ed. — Upper Saddle River, NJ: Prentice-Hall, 2010.
11. Pilone D., Pitman N. *UML 2.0 in a Nutshell*. — Cambridge, MA: O'Reilly Media, 2005.
12. Pressman R.S., Maxim B. *Software Engineering: A Practitioner's Approach*, 8th ed. — New York: McGraw-Hill, 2014.

13. Schach S.R. *Classical and Object-Oriented Software Engineering*, 8th ed. — New York: McGraw-Hill, 2010.
14. Shalloway A., Trott J.R. *Design Patterns Explained*, 2nd ed. — Boston, MA: Addison-Wesley, 2005. (Имеется русский перевод первого издания этой книги: Шаллоуей А., Тротт Дж.Р. *Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию.* — М.: Издательский дом "Вильямс", 2002.)
15. Shneiderman B., Plaisant C., Cohen M., Jacobs S., Elmqvist N., Diakopoulos N. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5th ed. — Boston, MA: Addison-Wesley, 2009.
16. Sommerville I. *Software Engineering*, 9th ed. — Boston, MA: Addison-Wesley, 2010. (Имеется русский перевод 6-го издания этой книги: Соммервил Я. *Инженерия программного обеспечения*. 6-е изд. — М.: Издательский дом "Вильямс", 2002.)

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Элиенс А. *Принципы разработки объектно-ориентированных программ*, 2-е изд. — М.: Издательский дом "Вильямс", 2001.
2. Астелс Д., Миллер Г., Новак М. *Практическое руководство по экстремальному программированию.* — М.: Издательский дом "Вильямс", 2002.
3. Скотт К. *UML. Основные концепции.* — М.: Издательский дом "Вильямс", 2002.
4. Леффингуэлл Д., Уидриг Д. *Принципы работы с требованиями к программному обеспечению. Унифицированный подход.* — М.: Издательский дом "Вильямс", 2002.
5. Тамре Л. *Введение в тестирование программного обеспечения.* — М.: Издательский дом "Вильямс", 2003.
6. Грехэм Я. *Объектно-ориентированные методы.* Принципы и практика, 3-е изд. — М.: Издательский дом "Вильямс", 2004.
7. Ларман К. *Применение UML 2.0 и шаблонов проектирования*, 3-е изд. — М.: Издательский дом "Вильямс", 2006.
8. Майерс Г., Баджетт Т., Сандлер К. *Искусство тестирования программ*, 3-е изд. — М.: ООО "Вильямс", 2012.
9. Смит Дж. *Элементарные шаблоны проектирования.* — М.: ООО "Вильямс", 2012.
10. Мартин Р.С. *Гибкая разработка программ на Java и C++: принципы, паттерны и методики.* — М.: ООО "Вильямс", 2017.

Глава 8

Из этой главы вы узнаете, как можно смоделировать различные структуры данных в основной памяти компьютера, организованной в виде отдельных ячеек с последовательно увеличивающимися адресами — предмет, известный как *реализация структур данных*. Цель здесь состоит в том, чтобы предоставить пользователю данных возможность получать доступ к ним с помощью абстрактных инструментов вместо того, чтобы вынуждать его мыслить в терминах реальной организации основной памяти компьютера. Наше обсуждение покажет, как желание конструировать такие абстрактные инструменты в конечном счете приводит к концепции объектов и объектно-ориентированному программированию.

Структуры данных

8.1. БАЗОВЫЕ СТРУКТУРЫ ДАННЫХ
- Массивы и записи
- Списки, стеки и очереди
- Деревья

8.2. СВЯЗАННЫЕ КОНЦЕПЦИИ
- Еще раз об абстракции
- Статические или динамические структуры
- Указатели

8.3. РЕАЛИЗАЦИЯ СТРУКТУР ДАННЫХ
- Представление в памяти массивов
- Представление в памяти записей
- Представление в памяти списков
- Представление в памяти стеков и очередей
- Представление в памяти двоичных деревьев
- Манипулирование структурами данных

8.4. НЕБОЛЬШОЙ ПРАКТИЧЕСКИЙ ПРИМЕР: РЕАЛИЗАЦИЯ БИНАРНОГО ДЕРЕВА

8.5. СПЕЦИАЛИЗИРОВАННЫЕ ТИПЫ ДАННЫХ
- Типы данных, определяемые пользователем
- Абстрактные типы данных

8.6. КЛАССЫ И ОБЪЕКТЫ

*8.7. УКАЗАТЕЛИ В МАШИННОМ ЯЗЫКЕ

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

В главе 6 была введена концепция структур данных и вы узнали, что языки программирования высокого уровня предоставляют методы, с помощью которых программисты могут выражать алгоритмы так, как если бы данные, которыми они манипулируют, хранились различными способами, отличными от размещения в последовательно расположенных ячейках основной памяти компьютера. Вы также узнали, что основные типы данных, поддерживаемые традиционными языками программирования, обычно называют примитивными типами данных. В этой главе будут рассмотрены основные методы, с помощью которых можно создавать, а затем манипулировать структурами данных, отличными от примитивных данных языков программирования — и это обсуждение в конечном счете уведет нас от традиционных структур данных и приведет к объектно-ориентированной парадигме. При этом связующей нитью, пронизывающей и направляющей всю эту прогрессию, является создание абстрактных инструментов.

8.1. Базовые структуры данных

Наше рассмотрение мы начнем с введения некоторых базовых структур данных, которые будут служить примерами в последующих разделах. Абстракции объединения данных в списки и другие структуры являются типичными инструментами решения различных задач.



Основные положения для запоминания

- **В методах вычислений при решении различных задач могут использоваться списки и наборы элементов других типов.**

Массивы и записи

В разделе 6.2 речь шла о структурах данных, известных как однородные массивы и неоднородные массивы, — компоненты последних называют записью или структурой. Напомним, что **массив** представляет собой "прямоугольный" блок данных, все элементы которого имеют одинаковый тип данных. Простейшей формой массива является одномерный массив — одна строка элементов, в которой позиция каждого из них определяется индексом. Например, одномерный массив с 32 элементами можно использовать для хранения информации о том, сколько раз, каждая буква алфавита встречается на определенной странице текста. Двухмерный массив состоит из нескольких строк и столбцов, в которых

позиции идентифицируются парой индексов — первый индекс определяет строку, в которой находится данный элемент, а второй индекс определяет соответствующий столбец. Примером может служить прямоугольный массив чисел, представляющих ежемесячное количество продаж, совершенных отдельными работниками отдела сбыта — элементы каждой строки представляют количество продаж, совершенных в определенный месяц отдельными работниками отдела, а элементы в каждом столбце представляют количество продаж, совершенных определенным сотрудником в разные месяцы года. В результате элемент в первом столбце третьей строки будет представлять количество продаж, совершенных третьим работником в январе.

Напомним, что в отличие от массива, **агрегатный тип** (запись или структура) — это блок элементов данных, которые могут иметь разный тип представления и размер. Элементы в таком блоке (записи или структуре) обычно называют полями. Примером данных такого типа может служить блок данных, относящихся к одному сотруднику, полями которого могут быть имя сотрудника (поле символьного типа), возраст (поле целочисленного типа) и рейтинг профессиональных навыков (поле типа числа с плавающей запятой). Поля в записях или структурах обычно доступны по имени поля, а не по числовому индексу.

Списки, стеки и очереди

Другая базовая структура данных — это **список**, представляющий собой набор элементов, расположенных последовательно, один за другим, как показано на рис. 8.1, *а*. Начало списка называется **головой** списка, а другой его конец — **хвостом**.



Рис. 8.1. Список, стек и очередь

Почти любой набор данных может быть представлен в виде списка. Например, текст можно представить в виде списка символов, двухмерный массив — в виде списка строк, а музыку, записанную на компакт-диске, — в виде списка звуков. Более традиционные примеры включают списки гостей, списки покупок, списки зачисленных на курсы или инвентарные списки. Действия, связанные со списком, варьируются в зависимости от ситуации. В некоторых случаях

может потребоваться удалять элементы из списка, добавлять в него новые элементы, "обрабатывать" элементы в списке по одному, изменять расположение элементов в списке или, возможно, выполнять поиск с целью удостовериться, что определенный элемент присутствует в списке. Все эти операции будут рассматриваться ниже в этой главе, а пока лишь отметим, что фундаментальное понятие списка и различных его вариантов — это, пожалуй, самая распространенная абстракция данных, встречающаяся в алгоритмах.



Основные положения для запоминания

- **Списки и операции со списками, такие как добавление, удаление и поиск, часто встречаются во многих программах.**
- **Основные операции с совокупностями элементов (коллекциями) включают добавление элементов, удаление элементов, итерацию по всем элементам и определение наличия элемента в коллекции.**

Ограничивая способ доступа к элементам списка, мы получаем два специализированных типа списков, известных как стеки и очереди. **Стек** — это список, в который элементы вставляются и из которого удаляются только в его голове. Примером является стопка книг, физические ограничения которой требуют, чтобы все добавления и удаления происходили только на ее вершине (рис. 8.1, *б*). Следуя обычной разговорной речи, голову стека называют **вершиной**, а хвост — **дном** или **основанием**. Операцию добавления нового элемента на вершине стека называют **вставкой** (push), а операцию удаления элемента с вершины стека называют **извлечением** (pop). Обратите внимание, что последний помещенный в стек элемент всегда будет и первым удаленным из него элементом — этот факт служит основанием для того, что стек называют структурой типа **LIFO** (*Last In, First Out* — последним вошел, первым вышел).

Данная характеристика (LIFO) означает, что стек идеально подходит для хранения элементов, которые должны извлекаться в порядке, обратном по отношению к тому, в котором они в него были вставлены. Именно поэтому стек часто используется как средство реализации механизма возврата. (Термин **механизм возврата** применяется к процессу, связанному с организацией выхода из системы в порядке, обратном порядку входа в эту систему. Классическим примером является пошаговый процесс возвращения по собственному следу с целью найти выход из леса.) В качестве примера рассмотрим базовую структуру, которой можно воспользоваться для реализации рекурсивного процесса. При запуске каждой новой активации рекурсивного процесса его предыдущая

Списки в языке Python

В языке Python встроенная структура данных `list` (список) является своего рода "универсальным складным ножом" среди абстракций языков программирования, которые можно найти в более современных языках. Список в языке Python имеет такие свойства, которые в более традиционных языках свойственны массивам: доступ к его элементам можно получить посредством индексации, начинающейся с нуля. Однако, подобно агрегатным типам (структуры или записи), элементы списка могут иметь разные размеры и типы данных, как в следующем примере:

```
listOfStuff = [42, 'синий', 3.14592]
```

Здесь создается новый объект списка с именем `listOfStuff`, который будет содержать целое число, строку символов и число с плавающей запятой. (В отличие от настоящих агрегатных типов, списки Python с разнородными данными по-прежнему доступны по индексу, тогда как поля объектов в языке Python доступны по имени.)

В языке Python списки имеют встроенные механизмы, позволяющие добавлять и удалять элементы, выполнять поиск и обработку каждого элемента в наборе.

```
# Добавление нового элемента в конец набора.
listOfStuff.append('прозрачный')
# Построчная печать каждого элемента набора, по одному в строке.
for item in listOfStuff:
      print(item)
# Проверка наличия определенного элемента в наборе.
if 'прозрачный' in listOfStuff:
      print('прозрачный присутствует в списке')
```

Списки в языке Python могут работать как структура данных типа стека (LIFO) с использованием только встроенных функции `pop()` и `insert()` с индексом `0`.

```
# Вставка нового элемента в вершину (индекс 0) набора.
listOfStuff.insert(0, 'гофрированный')
# Извлечение этого элемента из набора.
front = listOfStuff.pop(0)
```

Списки в языке Python также могут работать как структура данных типа очереди (FIFO) с использованием только встроенных функций `pop()` и `append()`.

активация должна быть приостановлена. Более того, после завершения каждой очередной активации потребуется возобновить выполнение предыдущей активации — той, которая была приостановлена при запуске только что завершившейся. Следовательно, если все активации рекурсивного процесса в момент приостановки будут помещаться в стек, то нужная активация будет находиться на вершине этого стека каждый раз, когда потребуется возобновить работу очередной активации.

Очередь — это список, в котором элементы удаляются только в его голове, а новые элементы вставляются только в хвост. Типичным примером является очередь людей, желающих купить билеты в театр (рис. 8.1, *в*), — человек в начале (голове) очереди обслуживается в кассе, тогда как вновь прибывшие пристраиваются в конец (или хвост) этой очереди. Вы уже познакомились со структурой очереди в главе 3, где говорилось о том, что операционная система с пакетной обработкой помещает задания, ожидающие выполнения, в очередь, называемую очередью заданий. Там также говорилось, что очередь представляет собой структуру, действующую по принципу **FIFO** (*First In, First Out* — первым пришел, первым вышел), а это означает, что элементы удаляются из очереди в том порядке, в котором они в нее поступали.

Очереди часто используются в качестве базовой структуры буфера, который, как объяснялось в главе 1, является областью хранения, выделенной для временного размещения данных, передаваемых из одного места в другое. Когда элементы данных поступают в буфер, они помещаются в конец очереди. Позднее, когда появляется возможность переслать очередной элемент в его конечный пункт назначения, он извлекается из буфера, и это будет тот из элементов, который в данный момент находится в голове очереди. В результате все элементы пересылаются в том порядке, в котором они поступали в буфер.

Деревья

Дерево — это совокупность элементов, имеющая иерархическую организацию, аналогичную организационной структуре типичной компании (рис. 8.2). Президент компании представлен верхним элементом с линиями, идущими вниз к его заместителям, за которыми следуют региональные управляющие, и т.д. На это интуитивное определение древовидной структуры накладывается одно дополнительное ограничение, которое (в терминах организационной структуры компании) состоит в том, что никто в компании не подчиняется одновременно двум разными руководителями. Это означает, что разные ветви организации не пересекаются на более низких уровнях. (В главе 6 вы уже встречались с примерами древовидных структур, в которых они были приведены в форме деревьев синтаксического анализа.)

Каждый элемент дерева называется **вершиной** или **узлом** (рис. 8.3). Самая верхняя точка именуется **корневой вершиной** (если перевернуть рисунок вверх ногами, этот узел будет представлять собой основание, или корень, дерева). Вершины на противоположной стороне дерева называются **конечными вершинами** (или **листами**). Количество узлов на самом длинном пути от корня к листу называют **глубиной** дерева. Другими словами, глубина дерева — это число горизонтальных уровней в нем.



Рис. 8.2. Пример организационной структуры компании

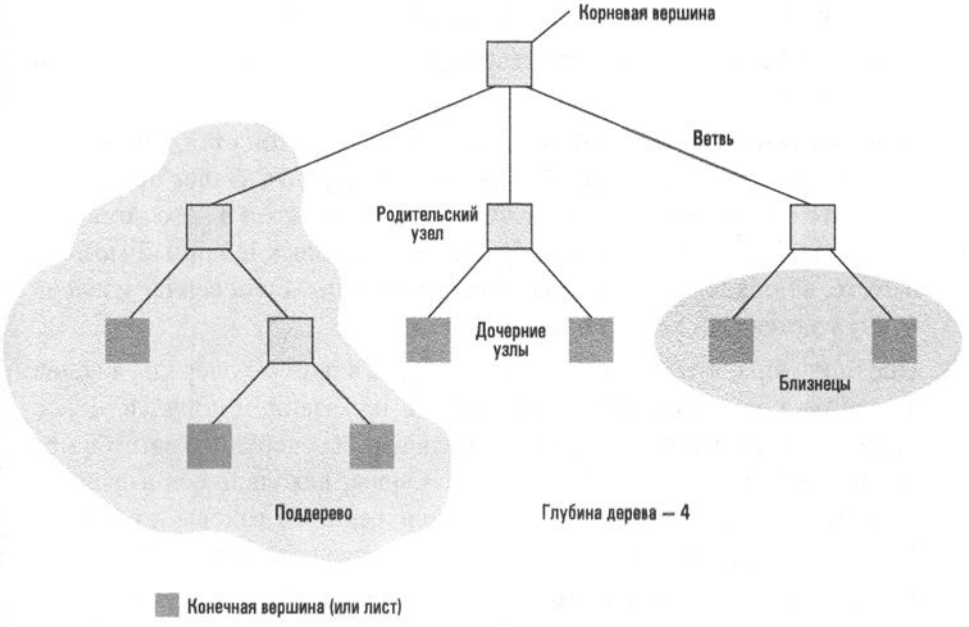


Рис. 8.3. Терминология деревьев

Иногда на древовидные структуры ссылаются так, как будто каждый узел порождает те узлы, которые расположены непосредственно под ним. В этом смысле могут употребляться ссылки на **предки** и **потомки** данной вершины. Непосредственные потомки узла называются его **дочерними** узлами, а непосредственный предок — его **родительским** узлом. Кроме того, узлы, имеющие общую родительскую вершину, называются **близнецами**. Дерево, в котором каждый родительский узел имеет не более двух дочерних, называется **бинарным** деревом.

Выбрав некоторую вершину дерева, можно заметить, что эта вершина вместе с расположенными ниже ее вершинами также образует некоторую древовидную структуру. Эту меньшую структуру принято называть **поддеревом**. Таким образом, каждый дочерний узел является корнем поддерева ниже родительского узла. Каждое такое поддерево называется **ветвью**, исходящей из родительского узла. В двоичном дереве часто говорят о левой и правой ветвях узла в соответствии с выбранным способом отображения этого дерева.

8.1. Вопросы и упражнения

1. Приведите примеры (вне области компьютерных наук) каждой из следующих структур: список, стек, очередь и дерево.
2. Назовите все различия, существующие между списками, стеками и очередями.
3. Предположим, что буква ***A*** была помещена в пустой стек, после чего в него поместили буквы ***B*** и ***C*** в указанном порядке. Далее предположим, что одна буква была извлечена из стека, а затем в него поместили буквы ***D*** и ***E***. Перечислите буквы, находящиеся в данный момент в стеке, в порядке их расположения сверху вниз. Если сейчас извлечь букву из стека, то какая это будет буква?
4. Предположим, что буква ***A*** была помещена в пустую очередь, а вслед за ней туда были помещены буквы ***B*** и ***C*** в указанном порядке. Далее предположим, что одна буква была удалена из очереди, а затем в нее добавили буквы ***D*** и ***E***. Перечислите буквы, находящиеся в данный момент в очереди, в порядке их расположения от головы до хвоста. Если сейчас удалить букву из очереди, то какая это будет буква?
5. Предположим, что дерево имеет четыре узла: ***A***, ***B***, ***C*** и ***D***. Если узлы ***A*** и ***C*** являются близнецами, а родительской вершиной для узла ***D*** является узел ***A***, то какие узлы в этом дереве являются листами? Какой узел в этом дереве является корневым?

8.2. Связанные концепции

В этом разделе мы выделяем три отдельные темы, которые тесно связаны с предметом структур данных: абстракция, различие между статическими и динамическими структурами и концепция указателя.

Еще раз об абстракции

Структуры, представленные в предыдущем разделе, чаще всего ассоциируются с данными. Однако основная память компьютера вовсе не организована в виде массивов, списков, стеков, очередей или древовидных структур. На самом деле она организована как простая последовательность адресуемых ячеек памяти. Следовательно, все другие структуры необходимо смоделировать в ней тем или иным образом. Как именно выполняется это моделирование, собственно, и является предметом данной главы. На данный момент мы просто выяснили, что такие структуры, как массивы, списки, стеки, очереди и деревья, являются некими абстрактными инструментами, создаваемыми таким образом, чтобы пользователи данных были полностью изолированы от деталей фактического хранения данных в компьютерах и могли получать доступ к информации таким образом, как если бы она хранилась в памяти компьютера в наиболее удобной для них форме.

Термин *пользователь* в этом контексте не обязательно относится к человеку. В действительности значение этого слова зависит от того, с какой точки зрения рассматривается данная ситуация. Если мы говорим о пользователе, использующем ПК для ведения записей о результатах розыгрышей в лотерее, то это человек. В этом случае прикладное программное обеспечение (вероятнее всего, приложение электронных таблиц) будет отвечать за представление данных в абстрактной форме, удобной для человека — скорее всего, в виде массива. Если же речь идет о сервере в Интернете, то для него пользователем может быть клиентский компьютер, который он обслуживает, и в этом случае сервер будет отвечать за представление данных в абстрактной форме, удобной для этого клиента. Если ситуация рассматривается с точки зрения программы с модульной структурой, то для нее пользователем будет любой модуль, требующий доступа к данным. В этом случае модуль, содержащий данные, будет отвечать за представление данных в такой абстрактной форме, которая будет удобной для других модулей. В каждом из этих сценариев связующая мысль заключается в том, что пользователю привилегия доступа к данным предоставляется как некий абстрактный инструмент.

Статические или динамические структуры

Важным моментом при построении абстрактных структур данных является то, какой должна быть эта моделируемая структура: статической или динамической, т.е. будет ли форма или размер структуры изменяться с течением времени. Например, если абстрактный инструмент представляет собой список имен, важно учитывать, будет ли этот список иметь фиксированный размер в течение

всего времени его существования или он будет расширяться или сокращаться по мере добавления в него новых имен или удаления уже существующих.

Как правило, статическими структурами управлять проще, чем динамическими. Если структура является статической, достаточно будет просто предоставить пользователю средства доступа к различным элементам данных в структуре и, возможно, средства изменения значений в указанных им местах. Если же структура является динамической, дополнительно потребуется решить задачи добавления и удаления элементов, а также нахождения дополнительного места в памяти, необходимого для растущей структуры данных. В случае плохо спроектированной структуры добавление одного нового элемента может потребовать масштабной перестройки всей структуры, а чрезмерный ее рост может привести к необходимости переноса всей структуры в другую область памяти, где доступно больше места.

Указатели

Вспомним, что различные ячейки в основной памяти машины идентифицируются по числовым адресам. Будучи просто числовыми величинами, эти адреса сами по себе легко могут быть закодированы и сохранены в ячейках памяти. **Указатель** — это область памяти, которая содержит такой закодированный адрес. В случае структур данных указатели используются для записи сведений о месте хранения отдельных элементов данных. Например, если потребуется многократно перемещать элемент данных из одного местоположения в другое, можно определить фиксированное местоположение в памяти компьютера как соответствующий указатель. Тогда каждый раз, когда элемент данных перемещается, достаточно будет просто обновить значение указателя так, чтобы он отражал новый адрес размещения элемента. Позже, когда потребуется получить доступ к этому элементу данных, его легко можно будет найти, воспользовавшись содержимым указателя. И действительно, при такой схеме указатель всегда будет "указывать" на текущее местоположение данных.

Мы уже сталкивались с концепцией указателя при обсуждении работы ЦП компьютеров в главе 2. Там сообщалось, что регистр, называемый счетчиком адреса, используется в ЦП для хранения адреса следующей команды, которая должна быть выполнена. Следовательно, счетчик адреса играет в этом случае роль указателя.

В качестве примера применения указателей предположим, что у нас есть список книг, хранящихся в памяти компьютера в алфавитном порядке по заголовкам. Хотя этот вариант будет удобен для многих приложений, такое расположение затрудняет поиск всех книг определенного автора — они будут разбросаны по всему списку. Чтобы решить эту проблему, можно зарезервировать

дополнительную ячейку памяти в каждом блоке ячеек, отведенных для представления книги, и использовать эту ячейку в качестве указателя на другой блок, представляющий книгу того же автора. В результате книги, написанные одним и тем же автором, можно будет связать в замкнутую цепочку (рис. 8.4). Как только будет найдена одна книга заданного автора, все остальные можно будет найти, просто следуя указателям от одной книги к другой.



Рис. 8.4. Сведения о книгах, упорядоченные по заглавию и дополнительно связанные по авторству

Сборка мусора

При расширении и сжатии динамических структур данных происходит выделение дополнительного и освобождение уже не используемого пространства памяти. Процесс поиска и восстановления доступности неиспользуемого пространства памяти с целью его дальнейшего использования известен как **сборка мусора**. Сборка мусора требуется во многих ситуациях. Программа управления памятью, имеющаяся в составе операционной системы, должна осуществлять сборку мусора при распределении и освобождении памяти. Программа управления файлами выполняет сборку мусора при размещении файлов в массовой памяти компьютера и удалении их оттуда. Кроме того, каждому процессу, работающему под управлением диспетчера, может понадобиться осуществить сборку мусора в пределах выделенного ему пространства памяти.

Процедура сборки мусора содержит некоторые скрытые проблемы. В случае связанных структур при каждом изменении указателя на элемент данных сборщик мусора должен решить, освобождается ли та область в памяти, на которую исходно указывал указатель. Проблема становится особенно сложной в переплетающихся структурах данных, содержащих множество ветвей указателей. Неточности в процедурах сборки мусора могут привести к потере данных или, наоборот, к неэффективному использованию пространства памяти. Например, если сборка мусора не в состоянии восстанавливать пространство памяти, объем доступного в системе пространства будет медленно сокращаться — явление, известное как **утечка памяти**.

Многие современные языки программирования включают указатели в качестве примитивного типа данных. Иначе говоря, они позволяют объявлять, размещать и манипулировать указателями способами, напоминающими целые числа и строки символов. Используя такой язык, программист может проектировать сложные сети данных в памяти машины, где указатели используются для соединения связанных элементов между собой.

8.2. Вопросы и упражнения

1. В каком смысле структуры данных, такие как массивы, списки, стеки, очереди и деревья, представляют собой абстракции?
2. Опишите приложение, в котором предполагается использовать статическую структуру данных. Затем опишите приложение, в котором предполагается задействовать динамическую структуру данных.
3. Приведите примеры контекстов (вне компьютерных наук), в которых имеет место понятие указателя.

8.3. Реализация структур данных

А теперь давайте рассмотрим способы, которыми структуры данных, обсуждавшиеся в предыдущих разделах, могут быть реализованы в основной памяти компьютера. Как было показано в главе 6, эти структуры часто бывают представлены в качестве примитивных структур данных в языках программирования высокого уровня. В данном случае наша цель — понять, как программы, работающие с такими структурами, переводятся в программы на машинном языке, манипулирующие данными, хранящимися в основной памяти.

Представление в памяти массивов

Наше обсуждение мы начнем с методов хранения массивов.

Предположим, что требуется сохранить последовательность из 24 почасовых значений температуры за сутки, для сохранения каждого из которых нужна одна ячейка памяти. Кроме того, предположим, что идентифицировать эти показания следует по их позиции в общей последовательности, т.е. необходимо иметь возможность доступа к отдельным значениям: первому, пятому и т.п. Короче говоря, требуется манипулировать этой последовательностью так, как если бы это был одномерный массив.

Этой цели можно достичь, просто сохранив показания в группе из 24 ячеек памяти с последовательными адресами. Затем, если адрес первой ячейки в последовательности равен x, местоположение в памяти любого конкретного показания температуры может быть вычислено путем вычитания единицы из индекса желаемого показания и последующего добавления результата к значению x. В частности, четвертое показание будет находиться по адресу $x + (4 - 1)$, как показано на рис. 8.5.

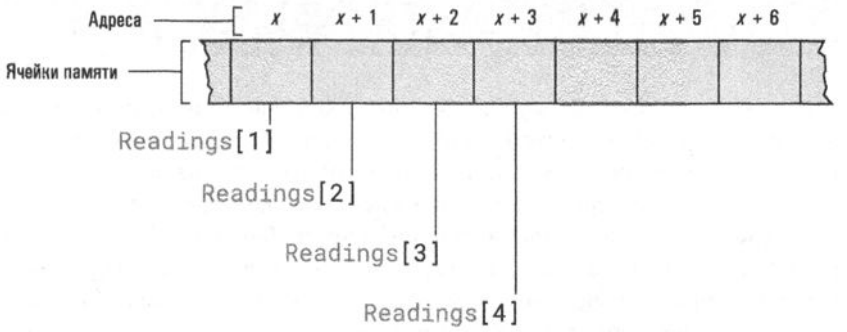


Рис. 8.5. Массив показаний температуры `Readings`, сохраненный в основной памяти, начиная с адреса x

Этот метод используется большинством трансляторов с языков программирования высокого уровня для реализации одномерных массивов. Когда транслятор встречает оператор объявления, такой как

```
int Readings[24];
```

он понимает, что терм `Readings` (Показания) будет использоваться для ссылок на одномерный массив из 24 целочисленных значений и организует выделение 24 последовательных ячеек памяти для размещения этого массива. Допустим, далее в программе транслятор встретит оператор присваивания

```
Readings[4] = 67;
```

Данный оператор указывает, что в четвертый элемент массива `Readings` следует поместить значение 67, поэтому транслятор создаст последовательность машинных команд, необходимых для помещения значения 67 в ячейку памяти по адресу $x + (4 - 1)$, где x — это адрес первой ячейки в блоке, связанном с массивом `Readings`. В результате программист получает возможность писать программу так, как будто показания температуры были действительно сохранены в одномерном массиве. (*Внимание*: в языках Python, C, C++, C# и Java отсчет индексов массивов начинается с `0`, а не с `1`, поэтому ссылка на четвертый элемент будет выглядеть как `Readings[3]`. См. вопрос 3 в конце этого раздела.)

Теперь предположим, что мы хотим записать информацию об объеме продаж, осуществленных отделом сбыта компании за одну неделю. В этом случае мы могли бы представить, что данные сохраняются в двухмерном массиве, значения в каждой строке которого представляют объем продаж, совершенных отдельным сотрудником, а значения в столбцах представляют объем продаж, совершенных всеми сотрудниками отдела в определенный день недели.

Библиотека стандартных шаблонов

Структуры данных, обсуждаемые в этой главе, уже стали стандартными структурами программирования, причем настолько стандартными, что во многих средах программирования они трактуются как примитивы. Один из примеров этому можно найти в типичной среде разработки языка программирования C++, которая часто дополняется **библиотекой стандартных шаблонов** (STL — *Standard Template Library*). Библиотека STL представляет набор предварительно определенных классов, описывающих популярные структуры данных. Следовательно, в результате включения библиотеки STL в программу на языке C++ программист избавляется от необходимости подробно описывать эти структуры. Вместо этого он просто объявляет, что те или иные идентификаторы принадлежат к тем или иным типам, объявляемым в библиотеке STL, подобно тому как в разделе 8.6 ниже в этой главе декларируется, что переменная `StackOne` принадлежат к типу `StackOfIntegers`.

Чтобы удовлетворить эту потребность, сначала примем к сведению, что этот массив является статическим в том смысле, что его размер не будет изменяться при обновлении. Следовательно, можно рассчитать размер области основной памяти, необходимый для всего массива, и зарезервировать блок смежных ячеек памяти требуемого размера. Далее, примем, что данные в массиве будут сохраняться строка за строкой. Начиная с первой ячейки зарезервированного блока в последовательных ячейках памяти будут сохранены значения из первой строки массива. После этого будут сохранены данные второй строки, затем — третьей и т.д. (рис. 8.6). Говорят, что такая система хранения использует **развертку по строкам** в отличие от **развертки по столбцам**, когда массив сохраняется столбец за столбцом.

Давайте рассмотрим, как при таком методе хранения данных можно найти ячейку памяти, содержащую значение из третьей строки четвертого столбца нашего массива. Представим, что мы находимся в первой ячейке зарезервированного блока памяти. Начиная с этой ячейки расположены данные первой строки массива, затем — второй, третьей и т.д. Чтобы добраться до данных

третьей строки, необходимо пропустить данные первой и второй строк. Поскольку каждая строка содержит пять элементов данных (по одному на каждый рабочий день — с понедельника по пятницу), следует пропустить десять элементов данных, чтобы попасть к первому элементу третьей строки. Чтобы от начала третьей строки добраться до значения четвертого столбца массива, следует продвинуться еще на три элемента. Таким образом, чтобы попасть к элементу третьей строки четвертого столбца, в сумме потребуется продвинуться на 13 элементов от начала блока.

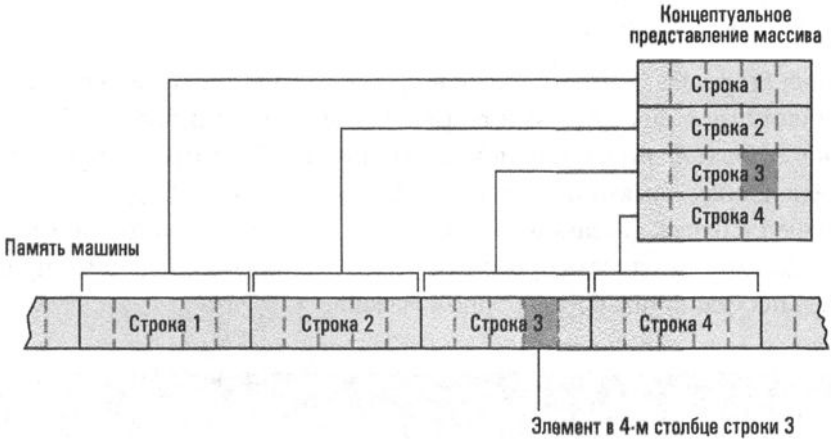


Рис. 8.6. Двухмерный массив из четырех строк и пяти столбцов, сохраненный в памяти с разверткой по строкам

Предыдущие вычисления можно обобщить, что позволит получить общий алгоритм, который может быть использован транслятором для преобразования ссылок в формате с указанием номеров строк и столбцов в реальные адреса в машинной памяти. В частности, если c является числом столбцов в массиве (т.е. числом элементов в каждой строке), то адрес элемента i-й строки j-го столбца будет выглядеть следующим образом:

$$x + (c \times (i - 1)) + (j - 1)$$

Здесь x — это адрес ячейки, содержащей элемент из первой строки первого столбца. Другими словами, чтобы дойти до i-й строки, необходимо продвинуться на $i - 1$ строк, каждая из которых содержит c элементов, а затем, чтобы дойти до j-го элемента этой строки, необходимо продвинуться еще на $j - 1$ элемент. В нашем примере $c = 5$, $i = 3$ и $j = 4$, так что если массив записан в памяти, начиная с адреса x, то элемент третьей строки четвертого столбца будет находиться по адресу $x + (5 \times (3 - 1)) + (4 - 1) = x + 13$. (Выражение $(c \times (i - 1)) + (j - 1)$ иногда называют **адресным полиномом**.)

И вновь, это метод, который используется большинством трансляторов языков программирования высокого уровня. Когда такой транслятор сталкивается с оператором объявления вида

```
int Sales[8, 5];
```

он понимает, что терм `Sales` будет использоваться для ссылок на двухмерный массив из 8 строк и 5 столбцов, и организует выделение 40 последовательных ячеек памяти для размещения этого массива. Допустим, далее в программе транслятор встретит оператор присваивания

```
Sales[3, 4] = 5;
```

Данный оператор указывает, что значение 5 должно быть помещено в элемент на пересечении третьей строки и четвертого столбца массива `Sales`. Транслятор создает последовательность машинных команд, необходимых для помещения значения 5 в ячейку памяти по адресу $x + 5 \times (3 - 1) + (4 - 1)$, где x — это адрес первой ячейки в блоке, связанном с массивом `Sales`. В результате программист получает возможность писать программу так, как если бы сведения о продажах действительно хранились в двухмерном массиве.

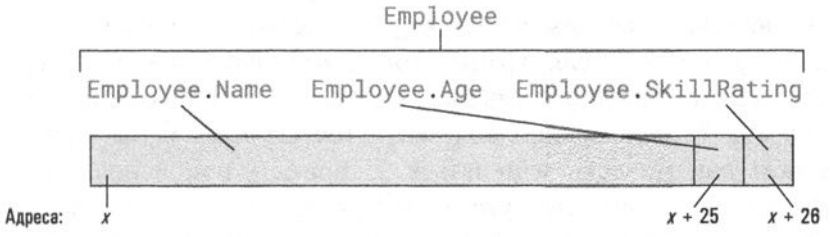
Реализация непрерывных списков

Примитивы для создания и манипулирования массивами, которые предоставляются в большинстве языков программирования высокого уровня, являются удобными инструментами для создания и манипулирования непрерывными списками. Если все элементы списка имеют одинаковый примитивный тип данных, то список представляет собой не что иное, как одномерный массив. Несколько более сложный пример — список из десяти имен, каждое из которых не длиннее восьми символов, как обсуждается в тексте. В этом случае программист может построить непрерывный список в виде двухмерного массива символов с десятью строками и восемью столбцами, что даст структуру, подобную представленной на рис. 8.6 (при условии, что массив хранится с разверткой по стокам).

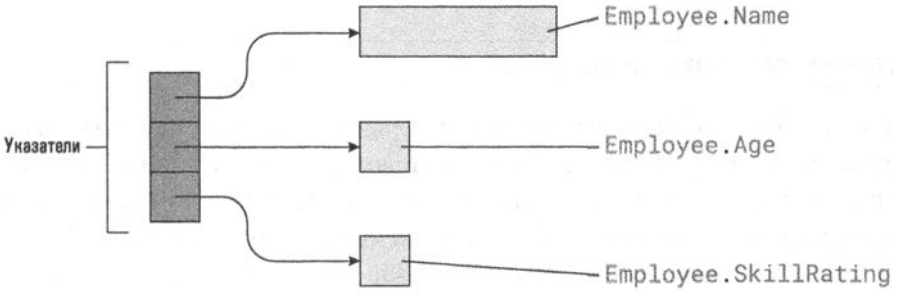
Многие языки высокого уровня включают функции, которые поощряют такие реализации списков. Например, предположим, что упомянутый выше двухмерный массив символов называется `MemberList`. Тогда, в дополнение к традиционной записи, в которой выражение `MemberList[3,5]` ссылается на один символ в третьей строке и пятом столбце, некоторые языки принимают выражение `MemberList[3]` для ссылки на всю третью строку, которая будет третьим элементом в списке.

Представление в памяти записей

Теперь предположим, что необходимо сохранить в памяти структуру (запись) с именем `Employee` (Работник), состоящую из трех полей: `Name` (Имя) с типом `character array` (строка символов), `Age` (Возраст) с типом `integer` (целое число) и `SkillRating` (Квалификация) с типом `float` (число с плавающей точкой). Если количество ячеек памяти, требуемых для каждого поля, является фиксированным, то эту структуру можно будет хранить в блоке смежных ячеек памяти. Например, предположим, что для поля `Name` требуется не более 25 ячеек, для поля `Age` требуется только одна ячейка, а для поля `SkillRating` также требуется только одна ячейка. В результате для хранения каждой записи можно было бы выделить блок из 27 смежных ячеек памяти, сохранив имя работника в первых 25 ячейках, его возраст — в 26-й ячейке и показатель квалификации — в последней ячейке (рис. 8.7, *а*).



а) Сохранение записи в непрерывном блоке ячеек памяти



б) Поля записи хранятся в памяти по отдельности

Рис. 8.7. Представление в памяти записи `Employee`

При таком расположении было бы очень просто получить доступ к различным полям одной записи. Ссылка на поле может быть преобразована в адрес ячейки памяти, если известен адрес начала записи в памяти машины и

смещение к требуемому полю в этой записи. Если принять, что адрес первой ячейки записи равен x, то любая ссылка на поле `Employee.Name` (что означает поле `Name` в записи `Employee`) будет указывать на 25 ячеек, расположенных в памяти начиная с адреса x, а ссылки на поле `Employee.Age` (поле `Age` в записи `Employee`) будут указывать на ячейку по адресу $x + 25$. В частности, если транслятор в программе на языке высокого уровня обнаружит оператор вида

```
Employee.Age = 22;
```

он просто создаст последовательность инструкций машинного языка, необходимую для помещения значения 22 в ячейку памяти с адресом $x + 25$.

Альтернативой хранению записи в блоке смежных ячеек памяти является сохранение каждого ее поля в отдельном месте, а затем связывание их вместе в одну структуру с помощью указателей. Говоря точнее, если запись содержит три поля, то в памяти машины выделяется место для хранения трех указателей, каждый из которых указывает на одно из ее полей (рис. 8.7, *б*). Если эти указатели хранятся в блоке, начинающемся по адресу x, то первое поле можно будет найти, используя значение указателя, хранящегося в местоположении x, второе поле можно будет найти, используя значение указателя в местоположении $x + 1$, и т.д.

Этот подход будет особенно полезен в тех случаях, когда размер полей записи может динамически изменяться. Например, при использовании для представления записи системы указателей размер любого поля легко можно увеличить, просто найдя в памяти область такого размера, который позволит сохранить увеличенное значение поля, а затем переопределив соответствующий указатель так, чтобы он указывал на новое местоположение. В подобном случае, если бы запись хранилась в непрерывном блоке ячеек памяти, потребовалось бы изменить всю ее структуру.

Представление в памяти списков

Теперь давайте рассмотрим методы, которые можно использовать для хранения списка имен в основной памяти компьютера. Одна из стратегий заключается в том, чтобы хранить содержимое списка в едином блоке ячеек памяти с последовательными адресами. Если предположить, что каждое имя будет состоять не более чем из восьми букв, то можно разделить этот большой блок ячеек на группу подблоков, каждый из которых будет состоять из восьми ячеек. В каждом таком подблоке можно хранить отдельное имя, записанное с помощью символов кода ASCII, — по одной ячейке памяти на каждую букву. Если имя не заполняет все ячейки выделенного для него подблока, можно просто заполнить оставшиеся ячейки ASCII-кодом символа "пробел". При использовании такой системы для хранения списка из десяти имен понадобится блок из восьмидесяти последовательно расположенных ячеек памяти.

Только что описанная система хранения представлена на рис. 8.8. Важным моментом здесь является то, что весь список хранится в одном большом блоке памяти, а последовательные записи следуют одна за другой в смежных ячейках памяти. Такую организацию данных называют **непрерывным списком**.



Рис. 8.8. Список имен, сохраняемый в памяти в виде непрерывного списка

Непрерывный список — это удобная структура хранения для реализации статических списков, но ему свойственны существенные недостатки в случае динамических списков, когда удаление и вставка имен могут потребовать выполнения трудоемкой перестановки записей. В худшем случае добавление новых записей может потребовать перемещения всего списка в иное место, где будет доступен блок ячеек, достаточно большой для размещения увеличившегося в размерах списка.

Всех этих проблем можно избежать, если позволить отдельным записям в списке храниться в разных областях памяти, а не всем вместе в одном большом непрерывном блоке. Чтобы объяснить суть такого подхода подробнее, давайте еще раз обратимся к примеру хранения списка имен (в котором каждое имя имеет длину не более восьми символов). На этот раз каждое имя будет сохранено в блоке из *девяти* непрерывных ячеек памяти. Из них первые восемь ячеек предназначены для хранения самого имени, а последняя ячейка будет использоваться в качестве указателя на следующее имя в списке. При таком подходе элементы списка могут быть разбросаны по отдельным небольшим блокам из девяти ячеек, связанных между собой указателями. Из-за наличия связей между отдельными элементами такая организация называется **связанным списком**.

Чтобы знать, где расположено начало связанного списка, в отдельной ячейке хранится еще один указатель, определяющий адрес первой записи в списке. Поскольку этот указатель указывает на начало (или голову) списка, он называется **указателем головного элемента**.

Чтобы отметить конец связанного списка, используется специальный **нулевой указатель** — `null` (в некоторых языках программирования называемый **NIL**, или объектом `None` в языке Python), который, по сути, является просто специальным битовым шаблоном, помещаемым в ячейку указателя последней

записи для указания, что в списке больше нет записей. Например, если мы договоримся никогда не сохранять какую-либо запись списка по адресу `0`, то нулевое значение никогда не будет использоваться в этом списке как допустимое значение указателя и, следовательно, может использоваться как нулевой указатель.

Окончательная структура связанного списка показана на рис. 8.9, на котором разбросанные блоки памяти, предназначенные для хранения отдельных элементов списка, представлены прямоугольниками. Каждый такой прямоугольник помечен для определения его назначения и составных элементов. Каждый указатель представлен стрелкой, которая ведет от самого указателя к объекту, на который он указывает. Обход списка начинается с перехода по указателю головного элемента, чтобы найти первую запись. Далее достаточно будет следовать указателям, сохраняемым в записях, чтобы последовательно переходить от очередной записи списка к следующей, пока не будет найден нулевой указатель.

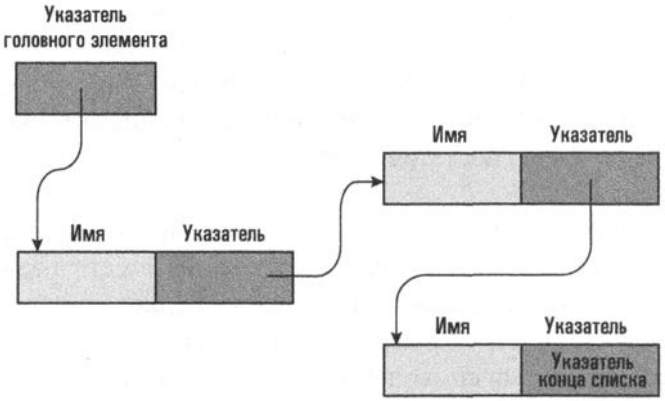


Рис. 8.9. Структура связанного списка

Чтобы оценить преимущества связанного списка в сравнении с непрерывным, рассмотрим задачу удаления элемента. В непрерывном списке это действие создает дыру, пустое место, а это означает, что все элементы списка, следующие за удаленным, необходимо переместить вперед для восстановления непрерывности всего списка. В случае же связанного списка элемент можно удалить путем изменения значения всего лишь одного указателя. Делается это очень просто: указатель, который ранее указывал на удаляемый элемент, изменяется так, чтобы он указывал на элемент, следующий в списке за удаляемым (рис. 8.10). Теперь при обходе списка удаленный элемент будет игнорироваться, поскольку он больше не является частью общей последовательности элементов списка.

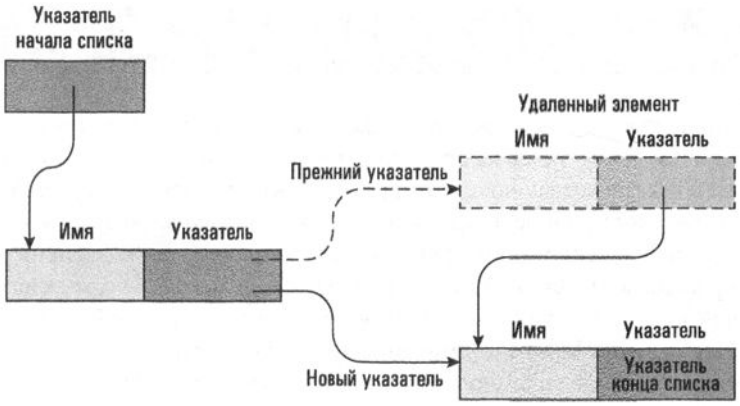


Рис. 8.10. Удаление элемента из связанного списка

Вставка нового элемента в связанный список лишь немногим сложнее. Сначала отыскивается неиспользуемый блок ячеек памяти, достаточно большой для размещения нового элемента и указателя. В нем сохраняется новый элемент, а в указатель помещается адрес того элемента в списке, который должен следовать за новым. Для завершения операции достаточно изменить указатель, связанный с элементом, предшествующим новому элементу в списке, таким образом, чтобы он указывал на новый элемент, как показано на рис. 8.11. После внесения всех этих изменений при каждом обходе списка новый элемент будет обнаружен в отведенном для него месте.

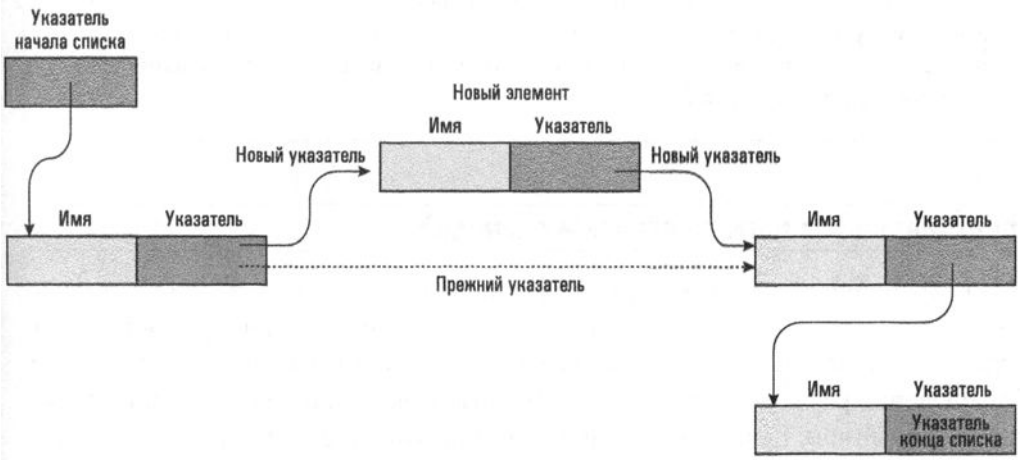


Рис. 8.11. Вставка элемента в связанный список

Проблема указателей

Известно, что использование блок-схем может привести к путанице при разработке алгоритмов (см. главу 5), а беспорядочное использование команд безусловного перехода `goto` ведет к созданию плохо спроектированных программ (см. главу 6). Точно так же бессистемное использование указателей, как оказалось, может привести к созданию необоснованно сложных и потенциально приводящих к ошибкам структур данных. Чтобы внести некоторый порядок в этот хаос, многие языки программирования ограничивают допустимую гибкость использования указателей. Например, в языке Java не разрешается использовать указатели общего вида. Допускается применение только ограниченных типов указателей — так называемых *ссылок*. Одно из отличий между ссылками и указателями состоит в том, что значение ссылки нельзя модифицировать с помощью арифметических операций. Например, если программист, работающий на языке Java, хочет переместить ссылку `Next` к следующему элементу непрерывного списка, он должен использовать инструкцию, эквивалентную следующему выражению:

```
redirect Next to the next list entry
```

Тогда как программист, работающий на языке C, может использовать инструкцию, эквивалентную следующей:

```
assign Next the value Next + 1
```

Заметим, что инструкция на языке Java лучше отражает назначение производимого действия. Более того, для выполнения инструкции языка Java необходимо, чтобы существовал еще один элемент списка. Однако если ссылка `Next` уже указывает на последний элемент списка, то выполнение инструкции языка C приведет к тому, что она будет указывать на нечто, находящееся вне списка, — распространенная ошибка начинающих (и не только начинающих) программистов, пишущих программы на языке C.

Представление в памяти стеков и очередей

Для хранения стеков и очередей часто используется организация их элементов, похожая на непрерывный список. В случае стека резервируется блок памяти, достаточно большой, чтобы вместить стек в его максимальном размере. (Определение размера этого блока часто может быть критически важным проектным решением. Если зарезервировано слишком мало места, стек в конечном итоге может выйти за пределы выделенного для его хранения пространства; а если зарезервировано слишком много места, определенное пространство памяти будет потрачено впустую.) Один конец этого блока отмечается как основание стека. Именно здесь сохраняется первый элемент, помещаемый в стек. Затем каждый дополнительный элемент помещается рядом со своим

предшественником по мере роста стека в направлении к другому концу зарезервированного блока ячеек памяти.

Обратите внимание, что по мере того, как элементы вставляются и извлекаются, расположение вершины стека будет перемещаться вперед и назад в пределах зарезервированного блока ячеек памяти. Чтобы отслеживать это местоположение, соответствующий адрес хранится в дополнительной ячейке памяти, называемой **указателем вершины стека**. Таким образом, указатель вершины стека всегда определяет текущее положение вершины стека.

Вся система в целом представлена на рис. 8.12 и работает следующим образом. Чтобы вставить элемент в стек, прежде всего следует так изменить текущее значение указателя вершины стека, чтобы оно указывало на свободную позицию, следующую за тем элементом, который является текущей вершиной стека, а затем поместить в эту позицию новый элемент. Чтобы извлечь элемент из стека, считываются те данные, на которые указывает содержимое указателя вершины стека, а затем текущее значение указателя вершины стека корректируется таким образом, чтобы оно указывало на следующий (ниже) расположенный элемент стека.

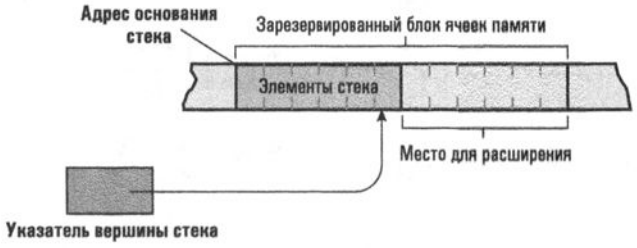


Рис. 8.12. Организация стека в памяти компьютера

Традиционная реализация очереди аналогична реализации стека. Опять же, резервируется блок смежных ячеек в основной памяти, достаточно большой, чтобы вместить всю очередь в ее предполагаемом максимальном размере. Однако в случае очереди операции потребуется выполнять на обоих концах структуры, поэтому для использования в качестве указателей выделяется *две* ячейки памяти, а не одна, как это было в случае стека. Один из этих указателей называется **указателем начала**, — в нем находится адрес начала очереди, а другой, называемый **указателем конца**, предназначен для хранения адреса ее конца. Когда очередь пуста, оба этих указателя указывают на одно и то же местоположение (рис. 8.13). Каждый раз при включении в очередь новый элемент помещается в позицию, определяемую указателем конца очереди, после чего значение этого указателя корректируется так, чтобы он указывал на следующую свободную позицию. Таким образом, указатель конца очереди всегда указывает на первую свободную позицию в конце очереди. Удаление элемента из очереди

означает извлечение элемента, на который указывает указатель начала очереди, с последующим изменением значения этого указателя таким образом, чтобы он указывал на элемент очереди, следующий за только что удаленным.

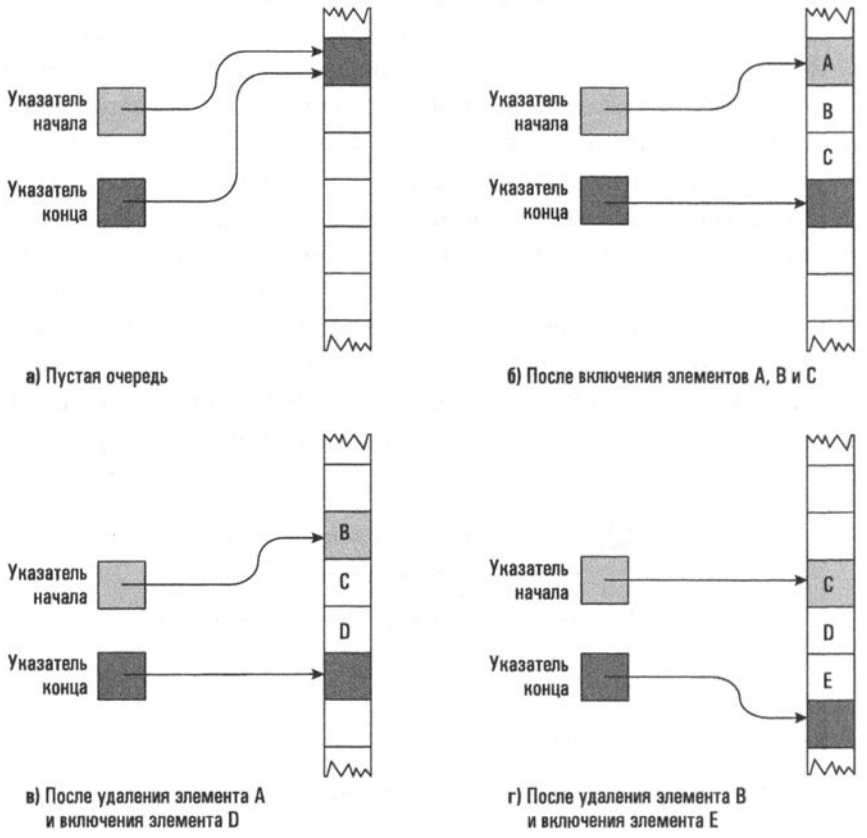


Рис. 8.13. Реализация очереди с использованием указателей ее начала и конца. Обратите внимание, как очередь перемещается в памяти при добавлении и удалении ее элементов

Однако описанный выше механизм хранения очереди связан с определенной проблемой. Если оставить его работу без контроля, то очередь будет ползти по памяти, как ледник, разрушая все данные на своем пути (см. рис. 8.13). Следовательно, необходим механизм ограничения перемещения очереди пределами зарезервированного для нее блока памяти. Решение простое. Очереди позволено свободно мигрировать в пределах выделенного блока, но когда конец очереди достигнет его конца, следующие вставляемые элементы будут вновь размещаться в начальном конце блока, который к этому времени освободится. Аналогичным образом, когда последняя позиция в блоке, в конце концов, уже будет содержать начальный элемент очереди, то после его удаления указатель

начала очереди вновь будет установлен на начало блока, где к этому времени уже будут находиться все оставшиеся элементы очереди. Таким образом, очередь как бы преследует саму себя по кругу внутри выделенного ей блока памяти, как показано на рис. 8.14. Такой метод реализации приводит к созданию структуры, называемой **циклической очередью**.

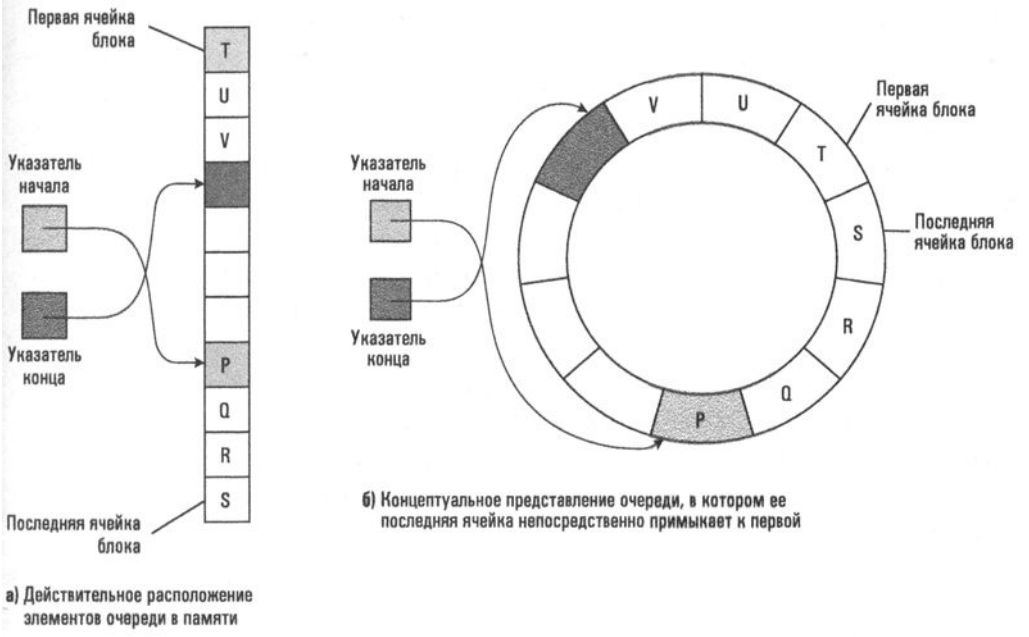


Рис. 8.14. Циклическая очередь, содержащая буквы от P до V

Представление в памяти двоичных деревьев

При обсуждении методов представления в памяти деревьев мы ограничимся только бинарными деревьями, т.е. деревья, в которых каждая вершина имеет не более двух дочерних вершин. Такие деревья обычно хранятся в памяти с использованием связанных структур, аналогичных связанным спискам. Однако вместо двух компонентов (значение данных, за которым следует указатель) каждый элемент (или вершина) бинарного дерева состоит из трех компонентов: значения данных, указателя первой дочерней вершины и указателя второй дочерней вершины. Хотя в пределах памяти машины не существует понятий "левый" и "правый", исходя из способа изображения дерева на бумаге, удобно считать первый указатель указывающим на **левую дочернюю вершину**, а второй — **на правую дочернюю**. Таким образом, каждая вершина дерева представляется коротким непрерывным блоком ячеек памяти, формат которого представлен на рис. 8.15.

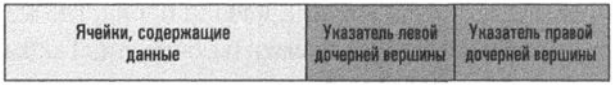


Рис. 8.15. Представление вершины бинарного дерева в памяти машины

Для сохранения дерева в памяти необходимо найти доступные блоки ячеек, достаточные для сохранения отдельных вершин, и связать эти вершины в соответствии с желаемой структурой данного дерева. Это означает, что каждый указатель надо установить так, чтобы он указывал на левую или правую дочернюю вершину для данной вершины, или присвоить ему значение `null`, если в этом направлении у дерева больше нет вершин. (Отсюда следует, что у листовых вершин дерева оба указателя имеют значение `null`.) Наконец, отдельно от дерева необходимо выделить специальную ячейку памяти, называемую **указателем корневой вершины**, в которой будет храниться адрес корневой вершины дерева. Наличие указателя корневой вершины обеспечивает первоначальный доступ к дереву.

Пример такой связанной системы хранения представлен на рис. 8.16. Здесь изображена концептуальная структура бинарного дерева вместе с реальным представлением этого дерева в памяти машины. Обратите внимание, что действительное расположение элементов в основной памяти может сильно отличаться от концептуального их расположения. Тем не менее с помощью указателя корневой вершины всегда можно получить доступ к корневой вершине дерева, а затем осуществить обход вниз по дереву, следуя от вершины к вершине по соответствующим указателям.

Альтернативой связанной системе хранения бинарных деревьев является метод, предусматривающий выделение одного непрерывного блока ячеек памяти для всех элементов дерева. При таком подходе в первой ячейке блока записывается корневая вершина (для простоты предположим, что каждая вершина дерева занимает одну ячейку), во второй — левый дочерний элемент корневой вершины, в третьей — правый дочерний элемент и т.д. В общем случае левый и правый дочерние элементы вершины n находятся в ячейках $2n$ и $2n + 1$ соответственно. В пределах блока ячейки, не используемые в данной древовидной структуре, отмечаются специальным двоичным кодом, означающим отсутствие данных. При использовании такого подхода концептуальная древовидная структура, изображенная на рис. 8.16, будет сохранена в памяти в виде, представленном на рис. 8.17. Отметим, что, в сущности, при этой системе хранения вершины, находящиеся на последовательно понижающихся уровнях дерева, сохраняются в виде непрерывных сегментов, следующих друг за другом. Иначе говоря, первый элемент блока является корневой вершиной дерева, за ним следуют дочерние вершины корневой вершины, затем — вершины-"внуки" корневой вершины и т.д.

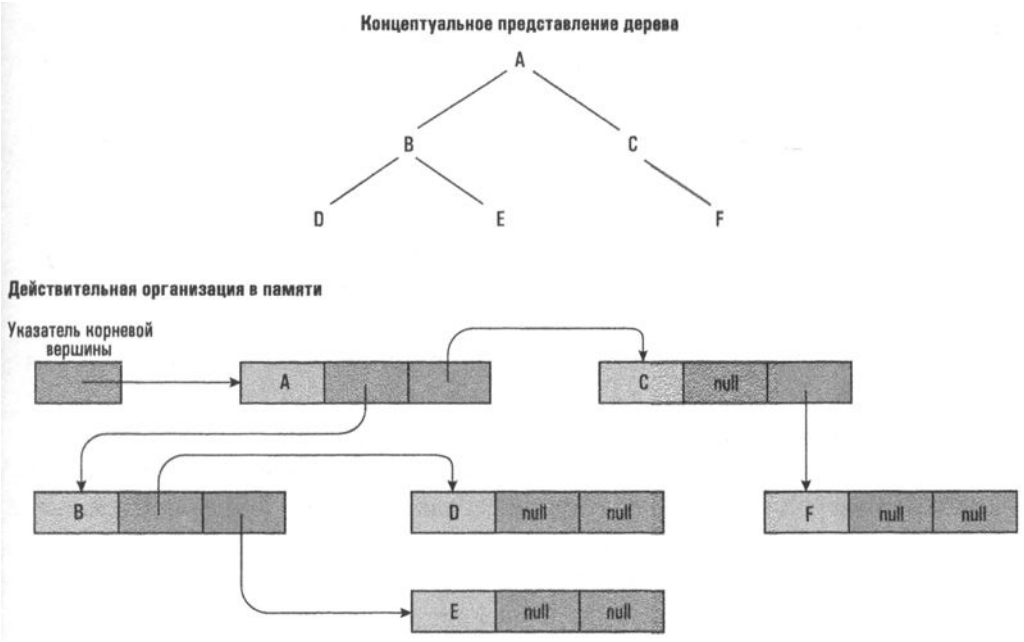


Рис. 8.16. Концептуальная и реальная организации бинарного дерева с использованием связанной системы хранения

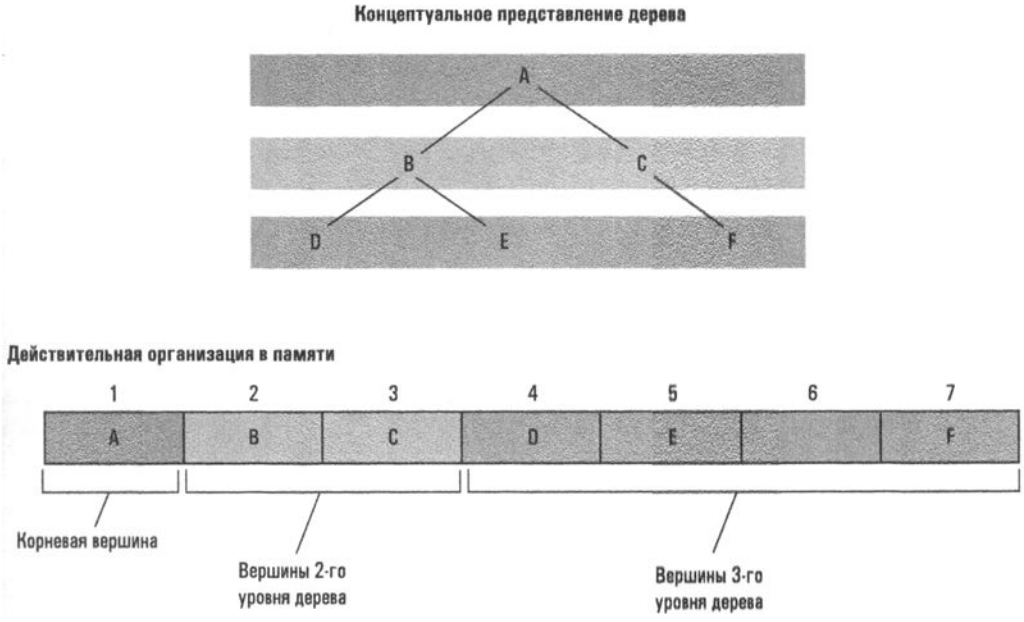


Рис. 8.17. Древовидная структура, сохраненная в памяти без использования указателей

В отличие от связанных структур, описанных выше, эта альтернативная система хранения обеспечивает для любой вершины дерева удобный метод нахождения ее родительской вершины или вершины, имеющей с данной общую родительскую вершину. Местонахождение родительской вершины для данной вершины можно найти, разделив позицию данной вершины в блоке на 2 и отбросив остаток (например, родительской вершиной для вершины, находящейся на 7-й позиции в блоке, будет вершина в позиции 3). Местоположение вершины, имеющей общего родителя с данной, можно найти, прибавив 1 к местоположению данной вершины, если оно четно, и отняв 1, если данная вершина находится на нечетной позиции в блоке (например, вершина, имеющая общего родителя с вершиной в позиции 4, находится в позиции 5, а вершина, имеющая общего родителя с вершиной в позиции 3, — в позиции 2). Кроме того, эта система хранения эффективно использует область памяти в случае бинарных деревьев, структура которых практически сбалансирована (оба поддерева, находящиеся ниже корневой вершины, имеют одинаковую глубину) и является полной (дерево не имеет длинных тонких ветвей). Для деревьев, не обладающих указанными характеристиками, данная система хранения может оказаться неэффективной, как показано на рис. 8.18.

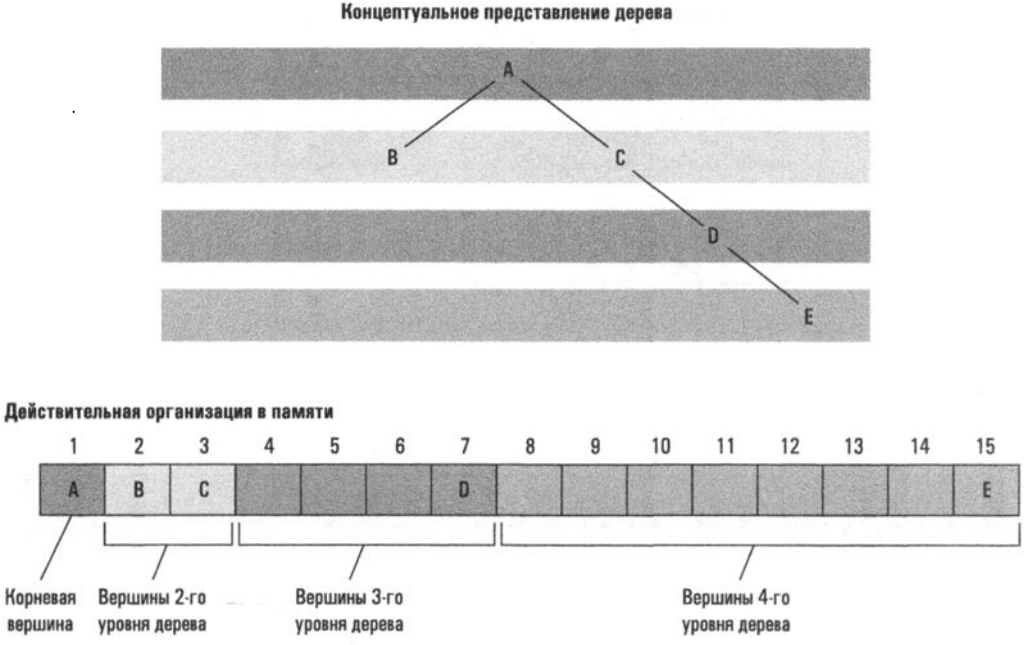


Рис. 8.18. Пример неполного и несбалансированного дерева в концептуальном представлении и схема его размещения в памяти в варианте без использования указателей

Манипулирование структурами данных

Вы уже видели, что способы, которым различные структуры данных фактически сохраняются в памяти компьютера, как правило, не совпадают с их концептуальной структурой, как ее представляет себе пользователь. Двухмерный массив в действительности не хранится в памяти как двухмерный прямоугольный блок ячеек, а список или дерево на самом деле может быть представлено в памяти множеством небольших элементов, произвольным образом размещенных в разных местах большой области памяти.

Следовательно, чтобы обеспечить пользователю доступ к структуре как к абстрактному инструменту, необходимо оградить его от излишних подробностей реального способа ее представления. Это означает, что инструкции, выданные пользователем (и представленные в терминах абстрактного инструмента), должны быть преобразованы в этапы выполнения, соответствующие реальной системе хранения. В случае массивов выше уже было показано, как это можно сделать, используя адресный полином для преобразования индексов строк и столбцов в адреса ячеек памяти. В частности, мы видели, как оператор присваивания

```
Sales[3, 4] = 5;,
```

написанный программистом, мыслящим в терминах абстрактного представления массива, может быть преобразован в конкретные действия, обеспечивающие внесение требуемых изменений в основной памяти компьютера.

Аналогичным образом было показано, как инструкция вида

```
Employee.Age = 22;,
```

включающая ссылку на абстрактный тип данных "запись", может быть преобразована в соответствующие действия в зависимости от того, как именно этот абстрактный тип данных представлен в памяти машины.

В случае списков, стеков, очередей и деревьев инструкции, сформулированные в терминах абстрактных структур, обычно преобразуются в соответствующие действия с помощью функций, обеспечивающих выполнение требуемых задач и защищающих пользователя от необходимости разбираться в деталях реальной системы представления и хранения данных. Например, если для вставки новых записей в связанный список предлагается использовать функцию `insert()`, то сведения о некоем С.И. Петрове могут быть добавлены в список слушателей курса "Физика-208" просто выдачей следующей инструкции, предусматривающей вызов этой функции:

```
insert("Петров С.И.", Physics208)
```

Обратите внимание, что вызов функции выполнен исключительно в терминах используемой абстрактной структуры — способ, которым фактически реализован этот список, скрыт.

В качестве более подробного примера на рис. 8.19 представлена функция с именем `printList()`, предназначенная для вывода на печать элементов связанного списка. В этой функции предполагается, что на первую запись списка указывает поле с именем `Head`, входящее в состав объекта с именем `List`, и что каждый элемент в списке состоит из двух частей: значения (`Value`) и указателя на следующую запись (`Next`). Также на рисунке в качестве указателя конца списка используется специальное значение языка Python `None`. Как только эта функция будет создана, ее можно будет использовать как абстрактный инструмент, предназначенный для вывода на печать связанных списков, не заботясь о тех действиях, которые действительно потребуется выполнить при печати. Например, чтобы вывести на печать список слушателей курса "Экономика-301", достаточно будет просто написать следующую инструкцию, осуществляющую вызов этой функции:

```
printList(Economics301ClassList)
```

Более того, если позже будет принято решение изменить способ фактического хранения списка, то потребуется лишь соответствующим образом изменить действия, выполняемые внутри функции `printList()`, а пользователь сможет выводить списки на печать посредством тех же вызовов этой функции, которые он использовал прежде.

```
def PrintList(List):
  CurrentPointer = List.Head
  while (CurrentPointer != None):
    print(CurrentPointer.Value)
    CurrentPointer = CurrentPointer.Next
```

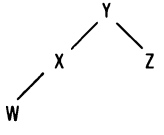
Рис. 8.19. Функция распечатки связанного списка

8.3. Вопросы и упражнения

1. Покажите, как приведенный ниже массив может быть записан в основной памяти с разверткой по строкам.

5	3	7
4	2	8
1	9	6

2. Приведите формулу для нахождения элемента *i*-й строки *j*-го столбца двухмерного массива, записанного в основной памяти с разверткой по столбцам, а не по строкам.
3. В языках программирования Python, C, C++, Java и C# отсчет индексов массивов начинается с 0, а не с 1, поэтому элемент, содержащийся в первой строке четвертого столбца массива с именем `Array`, обозначается как `Array[0][3]`. Какой адресный полином используется в этом случае для преобразования ссылок из формы `Array[i][j]` в адреса памяти?
4. Какое условие показывает, что связанный список пуст?
5. Преобразуйте процедуру на рис. 8.19 так, чтобы она прекращала печать, как только будет напечатано определенное имя.
6. Какое условие будет означать, что стек пуст, если для его представления использовать описанный в этом разделе метод реализации стека в непрерывном блоке ячеек памяти?
7. Опишите, как стек может быть реализован на языке высокого уровня в терминах одномерного массива.
8. Пусть очередь реализована в виде циклической структуры, как описано в данном разделе. Каково взаимоотношение указателей начала и конца очереди, когда очередь пуста и когда очередь полностью заполнена? Как можно определить, является очередь пустой или полностью заполненной?
9. Нарисуйте схему, представляющую, как выглядит приведенное ниже дерево в машинной памяти, если оно хранится по схеме с использованием левых и правых дочерних указателей, описанной в этом разделе. Затем нарисуйте другую схему, показывающую, как это дерево будет выглядеть в схеме с использованием непрерывного блока памяти, также обсуждавшейся в данном разделе.



8.4. Небольшой практический пример: реализация бинарного дерева

Давайте вернемся к задаче хранения списка имен в алфавитном порядке. Предположим, что с этим списком будут выполняться следующие операции.

Поиск определенного элемента.
Распечатка списка в алфавитном порядке.
Включение нового элемента.

Наша цель — разработать систему хранения данных вместе с набором процедур, выполняющих указанные операции и, следовательно, представляющих собой законченные абстрактные инструменты.

Начнем с рассмотрения вариантов возможных способов хранения списка. Если список будет храниться в соответствии с моделью связанного списка, поиск придется осуществлять методом последовательного перебора, что, как уже указывалось в главе 5, может оказаться весьма неэффективным, если список достаточно длинный. Следовательно, для списка необходимо выбрать реализацию, позволяющую применить в процедуре поиска алгоритм двоичного поиска (см. раздел 5.5). Чтобы можно было применить указанный алгоритм, система хранения должна позволять находить средний элемент последовательно уменьшающихся частей списка. Оптимальным выбором будет хранение списка в виде бинарного дерева. При этом средний элемент списка будет его корневой вершиной, средний элемент оставшейся первой части списка — ее левой дочерней вершиной, а средний элемент второй части списка — ее правой дочерней вершиной. Средние элементы оставшихся четвертей списка будут дочерними вершинами дочерних вершин корневой вершины и т.д. Например, дерево, показанное на рис. 8.20, представляет собой список, состоящий из букв A, B, C, D, E, F, G, H, I, J, K, L и M. (Если рассматриваемая часть списка содержит четное число элементов, будем считать средним больший из двух элементов.)

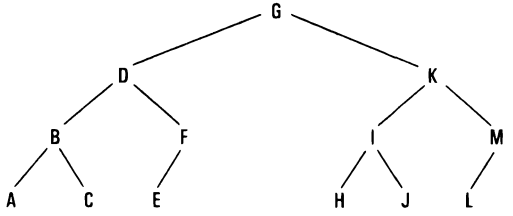


Рис. 8.20. Латинские буквы от A до M, организованные в виде упорядоченного дерева

Для поиска в списке, хранящемся в таком виде, следует начать со сравнения искомого значения с корневой вершиной. Если они эквивалентны, поиск успешно завершается. Если они не равны, выполняется переход к левой или правой дочерней вершине в зависимости от того, является ли искомое значение меньшим или большим корневого соответственно, — именно там находится средний элемент оставшейся части списка, необходимый для продолжения поиска. Этот процесс сравнения и перемещения к дочерним элементам продолжается до тех пор, пока не будет найдено искомое значение (это означает, что поиск был успешным) или не будет обнаружен указатель со значением `null` (`None`) без нахождения нужного нам значения (это означает, что поиск завершился неудачей).

На рис. 8.21 показано, как этот процесс поиска можно сформулировать для связанной древовидной структуры. В языке Python ключевое слово `elif` является сокращением для выражения "`else: if ...`" Обратите внимание, что эта функция является лишь уточнением функции, показанной на рис. 5.14, и представляющей нашу первоначальную формулировку двоичного поиска. Различие во многом чисто косметическое. Вместо того чтобы строить алгоритм с точки зрения поиска последовательно уменьшающихся сегментов списка, теперь алгоритм формулируется с точки зрения поиска последовательно уменьшающихся поддеревьев (рис. 8.22).

```
def Search(Tree, TargetValue):
  if (Tree is None):
    return None              # Поиск завершился неудачей
  elif (TargetValue == Tree.Value):
    return Tree              # Поиск был успешен
  elif (TargetValue < Tree.Value):
    return Search(Tree.Left, TargetValue)
    # Применить функцию Search для определения, находится ли
    значение TargetValue в поддереве, определяемом левым
    дочерним указателем корня, и сообщить полученный результат
  elif (TargetValue > Tree.Value):
    return Search(Tree.Right, TargetValue)
    # Применить функцию Search для определения, находится ли
    значение TargetValue в поддереве, определяемом правым
    дочерним указателем корня, и сообщить полученный результат
```

Рис. 8.21. Процедура двоичного поиска в списке, реализованном в виде связанного бинарного дерева

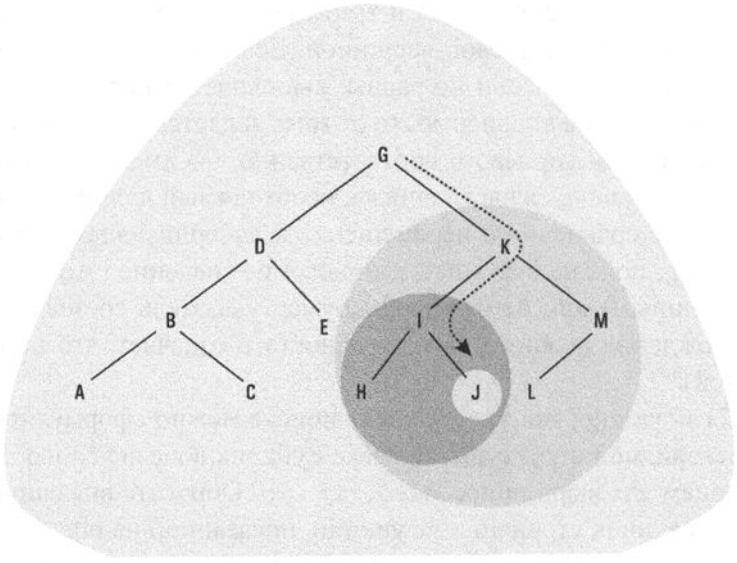


Рис. 8.22. Последовательно уменьшающиеся поддеревья, рассматриваемые функцией `Search`, представленной на рис. 8.21, при поиске буквы J

Можно предположить, что изменение естественного последовательного порядка хранения списка с целью повышения эффективности поиска приведет к возникновению трудностей при распечатке списка в алфавитном порядке. Однако это не так: чтобы распечатать список в алфавитном порядке, достаточно просто распечатать левое поддерево в алфавитном порядке, корневую вершину, а затем распечатать в алфавитном порядке правое поддерево (рис. 8.23). Все просто, ведь левое поддерево содержит все элементы, меньшие корневой вершины, а правое — все элементы, большие ее. Предварительный вариант процедуры, осуществляющей печать списка, выглядит следующим образом.

```
if (дерево не пусто):
  print левое поддерево в алфавитном порядке
  print корневая вершина
  print правое поддерево в алфавитном порядке
```

Этот предварительный вариант процедуры включает задачи печати левого и правого поддеревьев в алфавитном порядке, обе из которых, по существу, являются уменьшенными версиями исходной задачи. Следовательно, для задачи печати дерева необходимо решить меньшую задачу печати поддеревьев, что предполагает рекурсивный подход к решению всей проблемы.

Следуя этому подходу, можно привести наш предварительный вариант к полному тексту процедуры печати дерева, написанной на псевдокоде и представленной на рис. 8.24. Этой процедуре присвоено имя `PrintTree()` и она

обращается к самой себе для печати левого и правого поддеревьев. Обратите внимание, что условие окончания рекурсивного процесса (процесс доходит до пустого дерева, представленного значением None) обязательно будет достигнуто, так как каждая из активаций процедуры работает с меньшим деревом, чем вызвавшая ее процедура.

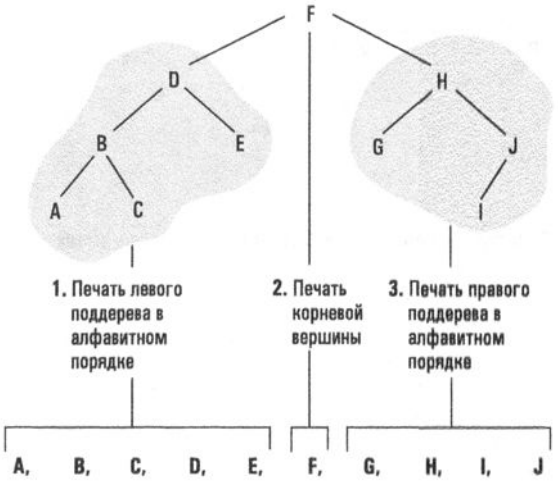


Рис. 8.23. Печать связанного бинарного дерева в алфавитном порядке

```
def PrintTree(Tree):
  if (Tree is None):
    PrintTree(Tree.Left)
    print(Tree.Value)
    PrintTree(Tree.Right)
```

Рис. 8.24. Процедура печати связанного бинарного дерева в алфавитном порядке

Задача включения в дерево нового элемента намного проще, чем может показаться на первый взгляд. Можно подумать, что такое включение потребует полного разрезания связей в дереве с целью нахождения подходящего места для нового элемента, но на самом деле добавляемую вершину всегда можно присоединить к основанию дерева в качестве листа, независимо от ее значения. Чтобы найти надлежащее место для нового элемента, следует перемещаться по дереву точно так, как при поиске данного элемента. Поскольку такого элемента в дереве нет, поиск обязательно приведет нас к основанию дерева. В данной точке и будет размещаться новая вершина (рис. 8.25), поскольку была найдена именно та позиция, к которой привел бы поиск этих новых данных.

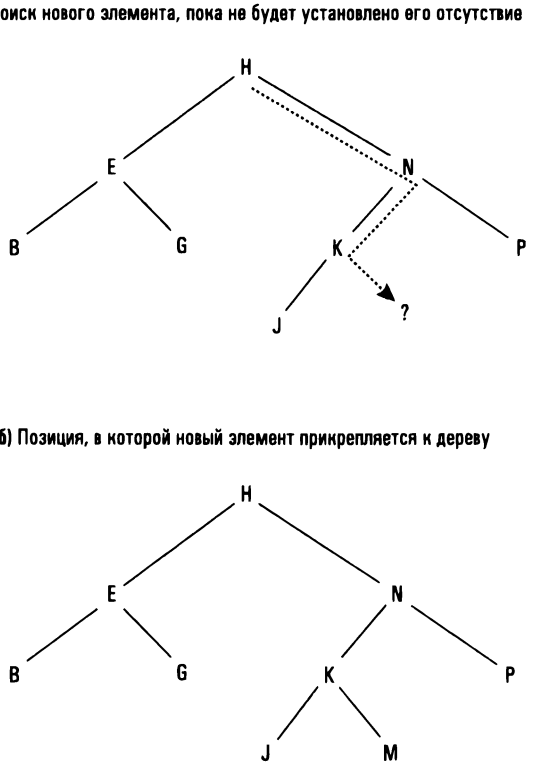


Рис. 8.25. Включение элемента `M` в список `B, E, G, H, J, K, N, P`, хранящийся в виде бинарного дерева

Функция, осуществляющая этот процесс при использовании связанной древовидной структуры, представлена на рис. 8.26. Сначала осуществляется поиск в дереве включаемого значения (переменная `NewValue`), а затем в соответствующую позицию помещается новая вершина, содержащая значение из переменной `NewValue`. Обратите внимание, что если включаемое значение при выполнении поиска будет *найдено*, то его вставка не выполняется. В программном коде на языке Python, представленном на рис. 8.26, для включения элемента, который будет служить новым листом в структуре связанного дерева, используется вызов функции `TreeNode()`. Для этого потребуется дополнительный код, не представленный на рисунке, который идентифицирует `TreeNode` как пользовательский тип, как это будет показано в следующем разделе.

Итак, можно сделать заключение, что пакет программного обеспечения, состоящий из связанной древовидной структуры и процедур поиска, печати и включения в нее данных, образует полный пакет, который может использоваться в качестве абстрактного инструмента в других создаваемых приложениях.

И действительно, при правильной реализации этот пакет может использоваться без учета фактической базовой структуры хранения. Используя процедуры в пакете, пользователь может представить список имен, хранящихся в алфавитном порядке, тогда как реальность такова, что записи "списка" фактически разбросаны по разным блокам ячеек памяти, которые связаны между собой в виде двоичного дерева.

```
def Insert(Tree, NewValue):
  if (Tree is None):
    # Создание нового листа, используя значение NewValue
    Tree = TreeNode()
    Tree.Value = NewValue
  elif (NewValue < Tree.Value):
    # Вставка NewValue в левое поддерево
    Tree.Left = Insert(Tree.Left, NewValue)
  elif (NewValue > Tree.Value):
    # Вставка NewValue в правое поддерево
    Tree.Right = Insert(Tree.Right, NewValue)
    # Если значение NewValue не меньше, чем, и не больше, чем
    # значение в корневой вершине, вставляемое значение
    # уже присутствует в дереве и вставка отменяется
  return Tree
```

Рис. 8.26. Функция включения элемента в связанное упорядоченное бинарное дерево

8.4. Вопросы и упражнения

1. Нарисуйте бинарное дерево, которое можно использовать для хранения и поиска в списке следующих элементов: R, S, T, U, V, W, X, Y, Z.
2. Укажите, по какому пути будет следовать алгоритм двоичного поиска, текст которого приведен на рис. 8.21, если его использовать для поиска элемента J в бинарном дереве, изображенном на рис. 8.20. Укажите путь следования этого алгоритма при поиске элемента P.
3. Нарисуйте схему, представляющую состояние активаций рекурсивного алгоритма печати бинарного дерева (см. рис. 8.24) во время печати элемента K упорядоченного бинарного дерева, представленного на рис. 8.20.
4. Опишите, как древовидную структуру, в которой каждый узел имеет до 26 дочерних узлов, можно было бы использовать для кодирования правильного написания слов в английском языке.

8.5. Специализированные типы данных

В главе 6 было введено понятие типа данных и рассмотрены такие элементарные типы данных, как целый, действительный (с плавающей точкой), символьный и логический, являющиеся базовыми в большинстве языков программирования. В этом разделе мы рассмотрим методы, с помощью которых пользователь сможет определять собственные типы данных, более точно отвечающие нуждам конкретного приложения.

Типы данных, определяемые пользователем

Часто удобнее описывать алгоритм, если используемые в нем типы данных отличаются от базовых типов данного языка программирования. Поэтому большинство современных языков программирования предоставляет пользователю возможность определять дополнительные типы данных, используя базовые типы в качестве строительных блоков. Такие "самодельные" типы данных принято называть **типами, определяемыми пользователем.**

В качестве примера предположим, что необходимо разработать программу, в которой будет использоваться множество переменных, имеющих одинаковую смешанную структуру, состоящую из имени, возраста и показателя квалификации некоторого работника. Один из подходов может состоять в повторном объявлении состава этой структуры при каждом обращении к ней (см. раздел 6.2). Однако лучшим подходом будет определить структуру лишь однажды как новый (определяемый пользователем) тип данных, а затем использовать его так, как будто он является одним из базовых типов языка программирования.

Вспомним пример оператора на языке C, приведенный в разделе 6.2:

```
struct
{
  char  Name[25];
  int   Age;
  float SkillRating;
} Employee;
```

Он определяет новую структуру (или запись) с именем `Employee`, содержащую поля `Name` (имя) — символьного типа, `Age` (возраст) — целое число и `SkillRating` (квалификация) — действительного типа (с плавающей точкой).

В противоположность этому очень похожий оператор на том же языке C определяет уже не новую структуру, а новый тип данных `EmployeeType`, представляющий собой структуру.

```
struct EmployeeType
{
  char  Name[25];
  int   Age;
  float SkillRating;
};
```

Теперь этот новый тип данных EmployeeType можно будет использовать при объявлении переменных наряду с базовыми типами языка. Это означает, что тем же самым способом, как в языке C с помощью следующего оператора можно объявить переменную x, имеющую тип целого числа:

```
int x;,
```

переменную Employee1 можно объявить как имеющую тип EmployeeType с помощью оператора

```
struct EmployeeType Employee1;
```

В результате далее в программе на переменную Employee1 можно будет ссылаться как на весь блок ячеек памяти, содержащих имя, возраст и показатель квалификации некоего работника. К отдельным элементам в этом блоке можно будет обращаться с помощью выражений типа Employee1.Name или Employee1.Age. Следовательно, оператор

```
Employee1.Age = 26;
```

можно использовать для присвоения значения 26 полю Age в блоке ячеек, доступном по имени Employee1. Более того, оператор

```
struct EmployeeType DistManager, SalesRep1, SalesRep2;
```

можно использовать для объявления сразу трех переменных с именами DistManager, SalesRep1 и SalesRep2, каждая из которых будет иметь тип EmployeeType, — аналогично тому, как оператор вида

```
float Sleeve, Waist, Neck;
```

обычно используется для объявления переменных с именами Sleeve, Waist и Neck, имеющих базовый тип языка float.

Очень важно различать определенные пользователем *типы* данных и сами *элементы* данных этих типов. Последние рассматриваются как **реализации** данного типа. Тип, определенный пользователем, по сути, является шаблоном, используемым при создании экземпляров данных этого типа. Он описывает свойства, которые имеют все реализации данного типа, но сам не является реальным представителем этого типа. В предыдущем примере определенный пользователем тип EmployeeType использовался для создания трех реализаций этого типа: DistManager, SalesRepl и SalesRep2.

Абстрактные типы данных

Определяемые пользователем типы данных, такие как структуры языка C и записи языка Pascal, играют важную роль во многих языках программирования, помогая разработчику программного обеспечения адаптировать представление данных к своим потребностям при создании конкретной программы. Однако традиционные пользовательские типы данных позволяют программистам определять лишь новые системы *хранения* и не предоставляют инструментов определения операций, которые могут выполняться над данными с этими структурами.

Абстрактный тип данных (*ADT* — Abstract Data Type) — это определяемый пользователем тип данных, который может включать как данные (представление), так и функции (поведение). Языки программирования, которые поддерживают создание ADT, обычно предоставляют два функциональных средства: одно — это синтаксис для определения ADT как единого блока и другое — это механизм для сокрытия внутренней структуры ADT от других частей программы, которые будут его использовать. Первая функция — важный организационный инструмент для хранения данных и функций ADT, что упрощает обслуживание и отладку программ. Вторая функция обеспечивает надежность, предотвращая доступ программного кода за пределами ADT к его данным без обращения к функциям, которые были специально предоставлены для этой цели.



Основные положения для запоминания

- **Списки и другие структуры наборов данных могут рассматриваться как абстрактные типы данных (ADT) при разработке программ.**

В качестве примера предположим, что в программе необходимо создать и использовать несколько стеков целочисленных значений. Наш подход может заключаться в реализации каждого стека в виде массива из 20 целочисленных значений. Нижний элемент в стеке будет помещен (вставлен) в первую позицию массива, а дополнительные элементы в стеке будут последовательно помещаться (вставляться) во все более и более высокие позиции массива (см. вопрос 7 в разделе 8.3). Дополнительная целочисленная переменная будет использоваться в качестве указателя вершины стека, — в ней будет содержаться индекс позиции массива, в которую нужно будет поместить следующий элемент стека. Таким образом, каждый стек будет состоять из массива, содержащего сам стек, и целочисленной переменной, выполняющей роль указателя вершины стека.

Чтобы реализовать этот план, сначала можно было бы определить пользовательский тип с именем StackType, воспользовавшись оператором языка C в форме

```
struct StackType
{
  int StackEntries[20];
  int StackPointer = 0;
};
```

(Напомним, что в таких языках, как C, C++, C# и Java, индексы для массива StackEntries будут изменяться в диапазоне от 0 до 19, поэтому переменная StackPointer инициализируется значением 0.) Сделав это объявление, далее можно было бы объявить стеки с именами StackOne, StackTwo и StackThree с помощью оператора

```
struct StackType StackOne, StackTwo, StackThree;
```

На этом этапе каждая из переменных StackOne, StackTwo и StackThree будет ссылаться на собственный блок ячеек памяти, используемый для реализации отдельного стека. Но что если теперь нам потребуется поместить значение 25 в переменную StackOne? Было бы желательно избежать использования деталей структуры массива, положенного в основу реализации стека, и просто пользоваться стеком как абстрактным инструментом — возможно, с помощью вызова функции, подобно следующему оператору:

```
push(25, StackOne);
```

Но такой оператор нельзя будет использовать, пока мы предварительно не определим соответствующую функцию с именем push(). Другие операции, которые нам может потребоваться выполнять с переменными типа StackType, включают извлечение элементов из стека, проверку, является ли стек пустым, и проверку, заполнен ли стек полностью, — и все это потребует определения дополнительных функций. Короче говоря, наше определение типа данных StackType не включает в себя все те свойства, которые мы хотели бы связать с этим типом. Более того, любая функция в программе может потенциально получить доступ к полям StackPointer и StackEntries в переменных типа StackType, минуя тщательные проверки, которые можно было бы разработать для собственных функций push() и pop(). Неаккуратный оператор присваивания в другой части программы может перезаписать элемент данных, хранящийся в середине структуры данных стека, или даже вообще разрушить поведение этой структуры по принципу LIFO, характерное для всех стеков.

Итак, нам необходимо иметь механизм для определения операций, которые будут разрешены для типа StackType, а также для защиты его внутренних переменных от внешнего вмешательства. Одним из таких механизмов является

синтаксис interface в языке Java. Например, на языке Java можно написать следующее определение:

```
interface StackType
{
  public int     pop();           /* Возвращает элемент из вершины
                                     стека  */
  public void    push(int item); /* Вставляет новый элемент в стек */
  public boolean isEmpty();       /* Проверка, является ли стек
                                     пустым    */
  public boolean isFull();        /* Проверка, заполнен ли стек
                                     полностью */
}
```

Само по себе это объявление абстрактного типа данных не определяет, как будет храниться стек или какие алгоритмы будут использоваться для выполнения функций push(), pop(), isEmpty() и isFull(). Все эти детали (представленные в этом операторе interface как абстракции) будут описаны в каком-то ином месте Java-кода. Однако, как и в случае определенного выше пользовательского типа данных, программисты теперь смогут объявлять переменные или использовать параметры-функций типа данных StackType.

Так, теперь можно будет объявить переменные StackOne, StackTwo и StackThree как имеющие тип стека, воспользовавшись оператором

```
StackType StackOne, StackTwo, StackThree;
```

Далее в программе (изначально эти три переменные представлены как нулевые ссылки и перед использованием должны быть связаны с реализациями конкретных классов языка Java, однако здесь мы не будем обсуждать все эти детали) в эти стеки можно будет помещать элементы с помощью таких операторов, как

```
StackOne.push(25);
```

Эта запись означает, что необходимо выполнить функцию push(), связанную с переменной StackOne, используя значение 25 в качестве ее фактического параметра.

В отличие от более простых определяемых пользователем типов данных, абстрактные типы данных являются полными типами данных, и их появление в таких языках, как Ada в 1980-х годах, представляло собой значительный шаг вперед в разработке языков программирования. Сегодня объектно-ориентированные языки предоставляют расширенные версии абстрактных типов данных, называемых классами, как это будет показано в следующем разделе.

8.5. Вопросы и упражнения

1. В чем различие между абстрактным типом данных и реализацией этого типа?
2. В чем различие между типом данных, определяемым пользователем, и абстрактным типом данных?
3. Опишите абстрактный тип данных для реализации списка.
4. Опишите абстрактный тип данных для реализации текущих счетов.

8.6. Классы и объекты

Как объяснялось в главе 6, использование объектно-ориентированной парадигмы приводит к построению систем, состоящих из элементов, называемых объектами, которые взаимодействуют между собой для выполнения поставленных задач. Каждый объект — это самостоятельная единица, отвечающая на сообщения, поступающие от других объектов. Объекты описываются шаблонами, которые называют классами.

Во многих отношениях эти классы фактически являются описаниями абстрактных типов данных (экземпляры которых в данном случае называются объектами). Например, на рис. 8.27 показано, как класс, которому присвоено имя `StackOfIntegers`, может быть определен на языках Java и C#. (Определение эквивалентного класса в C++ имеет ту же структуру, но немного отличается синтаксисом.) Обратите внимание, что этот класс включает реализацию для каждой из функций, объявленных в абстрактном типе данных `StackType`. Кроме того, этот класс содержит массив целых чисел с именем `StackEntries` и целочисленное значение с именем `StackPointer`, предназначенное для хранения указателя на вершину стека в массиве.

Используя этот класс в качестве шаблона, объект с именем `StackOne` может быть создан в программе на языке Java или C# с помощью оператора следующего вида:

```
StackType StackOne = new StackOfIntegers();
```

В программе на языке C++ для той же цели можно использовать оператор

```
StackOfIntegers StackOne();
```

Далее в программах значение `106` может быть вставлено как новый элемент в стек объекта `StackOne` с помощью оператора

```
StackOne.push(106);
```

```
class StackOfIntegers implements StackType
{
  private int[] StackEntries = new int[20];
  private int StackPointer = 0;

  public void push(int NewEntry)
  {  if (StackPointer < 20)
       StackEntries[StackPointer++] = NewEntry;
  }

  public int pop()
  {  if (StackPointer > 0) return StackEntries[--StackPointer];
       else return 0;
  }

  public boolean isEmpty()
  {       return (StackPointer == 0);   }

  public boolean isFull()
  {       return (StackPointer >= MAX);   }
}
```

Рис. 8.27. Стек целых чисел, реализованных в виде класса в языках Java и C#

Извлечь из стека объекта `StackOne` верхний элемент и поместить его в переменную `OldValue` можно посредством оператора

```
OldValue = StackOne.pop();
```

Эти функции, по сути, такие же, как и те, которые были связаны с абстрактными типами данных. Однако между классами и абстрактными типами данных есть различия. В действительности класс является *расширением* абстрактного типа данных. Например, как объяснялось в разделе 6.5, объектно-ориентированные языки позволяют классам наследовать свойства других классов и включать специальные методы, называемые конструкторами, которые осуществляют необходимую настройку отдельных объектов непосредственно при их создании. Кроме того, классы могут обеспечивать различную степень инкапсуляции (раздел 6.5), позволяющей защитить *внутренние* свойства их экземпляров от ошибочных обращений, но при этом обеспечивая возможность доступа извне к другим их полям.

В конечном счете можно сделать вывод, что концепции классов и объектов представляют собой еще один шаг в эволюции методов представления абстракции данных в программах. Фактически именно способность удобным способом определять и использовать абстракции и привела к росту популярности парадигмы объектно-ориентированного программирования.

8.6. Вопросы и упражнения

1. В чем сходство абстрактных типов данных и классов? Чем они различаются?
2. В чем состоит различие между классом и объектом?
3. Опишите класс, который можно будет использовать в качестве шаблона для создания объектов типа "очередь элементов целочисленного типа".

8.7. Указатели в машинном языке

В этой главе было введено понятие указателей и показано, как они используются при конструировании структур данных. В данном разделе мы обсудим, как работа с указателями реализуется в машинном языке.

Предположим, что на машинном языке Vole, описанном в приложении B, требуется написать программу, предназначенную для извлечения элемента из стека, описанного на рис. 8.12, с последующим помещением его в регистр общего назначения. Другими словами, требуется загрузить регистр содержимым той ячейки памяти, в которой сохранен элемент, представляющий собой вершину стека. В нашем машинном языке есть две инструкции для загрузки регистра: одна — с кодом операции `2` и другая — с кодом операции `1`. Вспомним, что если код операции равен `2`, то поле операнда содержит *сами* загружаемые данные, а если код операции равен `1`, то поле операнда содержит *адрес* загружаемых в регистр данных.

Поскольку мы не знаем, каким будет содержимое ячейки, мы не можем использовать для наших целей операцию с кодом `2`. Более того, мы не можем использовать и операцию с кодом `1`, так как не знаем адреса требуемой ячейки, поскольку адрес вершины стека может меняться в процессе выполнения программы. Что мы действительно знаем, так это адрес указателя вершины стека, т.е. местонахождение адреса данных, которые требуется загрузить. Поэтому необходимо иметь третий тип операции загрузки регистра, для которой поле операндов будет содержать адрес указателя на данные, которые надо загрузить.

Разработчик машины, описанной в приложении B, мог бы присвоить подобной операции код `0xD`. В этом случае машинный язык следовало бы разработать так, чтобы инструкция вида `0xDRXY` означала загрузку регистра `R` содержимым ячейки памяти, адрес которой находится по адресу `XY` (рис. 8.28). Следовательно, если указатель вершины стека находится в ячейке памяти с адресом `0xAA`, инструкция `0xD5AA` будет приводить к загрузке данных из вершины стека в регистр `5`.

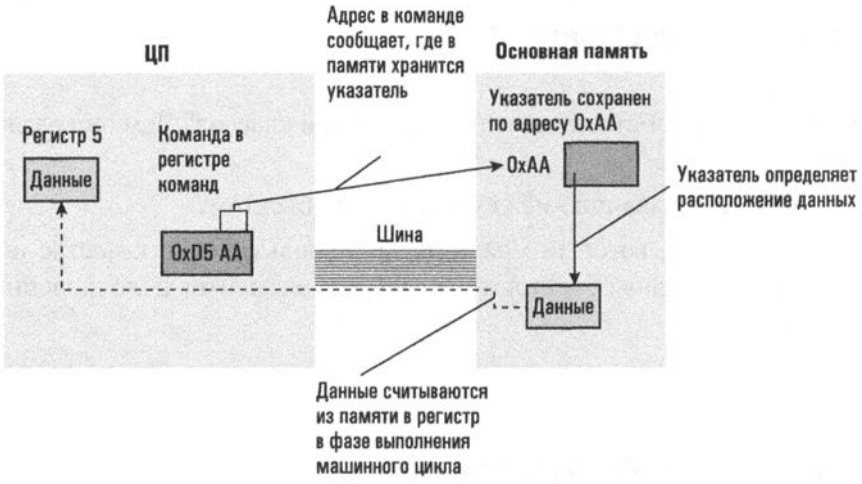


Рис. 8.28. Первая попытка расширить машинный язык Vole, описанный в приложении B, чтобы воспользоваться указателями

Однако эта машинная команда не реализует всю операцию извлечения. Для ее завершения требуется также вычесть единицу из текущего значения указателя вершины стека, чтобы теперь он указывал на новую вершину стека. Это означает, что в программе на машинном языке за командой загрузки вершины стека в регистр должны следовать команды загрузки в регистр указателя вершины стека, вычитания из него единицы и записи результата обратно в память.

Если в качестве указателя вершины стека использовать не ячейку памяти, а один из регистров, можно уменьшить число необходимых перемещений указателя вершины стека из памяти в регистр и обратно. Но для этого потребуется изменить и нашу новую команду загрузки, — так, чтобы учесть тот факт, что указатель находится в регистре, а не в ячейке основной памяти. Следовательно, в этом случае разработчик машины должен определить машинную команду с кодом операции `0xD` в виде `0xDR0S`, что будет означать загрузку в регистр `R` содержимого той ячейки памяти, на которую указывает содержимое регистра `S`, как показано на рис. 8.29. Тогда, выполнив данную команду, можно будет полностью завершить операцию извлечения элемента из стека, просто уменьшив следующей командой содержимое регистра `S` на единицу.

Отметим, что аналогичная команда необходима и для реализации операции вставки. Разработчик машины Vole может еще раз расширить ее машинный язык, описанный в приложении B, добавив машинную команду с кодом операции `0xE`, которая будет иметь вид `0xER0S` и осуществлять запись содержимого

регистра R в ту ячейку памяти, адрес которой содержится в регистре S. И вновь, выполнив данную команду, можно будет полностью завершить операцию вставки элемента в стек, просто увеличив следующей командой содержимое регистра S на единицу.

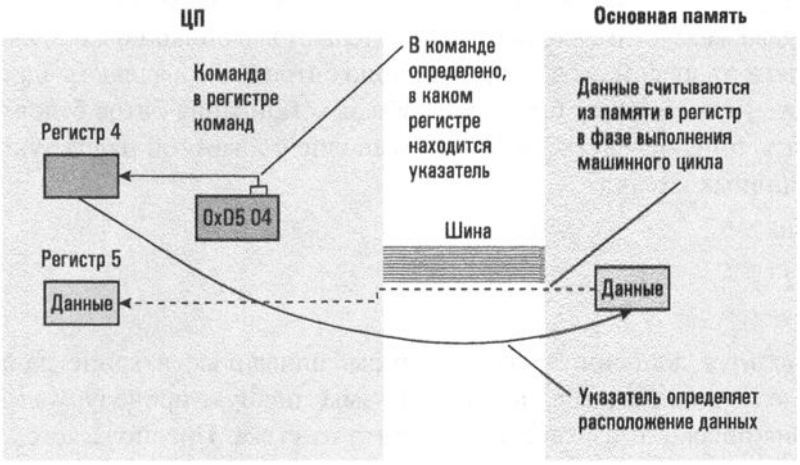


Рис. 8.29. Загрузка в регистр содержимого ячейки памяти, адрес которой определяется указателем, сохраненным в другом регистре

Новые операционные коды 0xD и 0xE, предложенные в этом разделе, демонстрируют не только то, как в машинных языках может быть реализовано манипулирование указателями, но и иную технику адресации, которой не было в исходном варианте этого машинного языка. Как объясняется в приложении B, в машинном языке Vole используется два способа идентификации данных, обрабатываемых при выполнении команды. Первый из них используется в команде с кодом операции 0x2, — здесь поле операнда команды в явном виде содержит те данные, которые будут обрабатываться. Этот тип называется **непосредственной адресацией**. Второй способ идентификации данных используется в командах с кодами операций 0x1 и 0x3. Здесь в каждом случае поле операнда команды содержат адрес, по которому обрабатываемые данные доступны в основной памяти. Этот вариант называется **прямой адресацией**. Однако предложенные выше новые операции с кодами 0xD и 0xE демонстрируют еще одну форму адресации данных в машинных командах. Поля операндов этих команд содержат *адрес адреса* данных. Этот метод называется **косвенной адресацией**. Все эти три типа адресации широко используются в современных машинных языках.

8.7. Вопросы и упражнения

1. Предположим, что машинный язык Vole, описанный в приложении B, дополнен теми командами, которые были предложены в конце этого раздела. Допустим, что регистр `8` содержит комбинацию `0xDB`, ячейка памяти с адресом `0xDB` — комбинацию битов `0xCA`, а ячейка с адресом `0xCA` — комбинацию битов `0xA5`. Какая комбинация битов будет в регистре 5 непосредственно после выполнения каждой из следующих машинных команд?

   а. `0x25A5`

   б. `0x15CA`

   в. `0xD508`

2. Используя дополнительные команды, описанные в конце данного раздела, напишите на машинном языке полную процедуру выполнения операции извлечения элемента из стека. Предполагается, что стек реализован так, как показано на рис. 8.12: указатель вершины стека находится в регистре `F`, а вершина стека должна быть извлечена в регистр `5`.

3. С помощью дополнительных команд, описанных в конце данного раздела, напишите на машинном языке программу копирования содержимого пяти последовательно расположенных ячеек памяти, начиная с адреса `0xA0`, в пять ячеек памяти, начиная с адреса `0xB0`. Исходите из того, что программа начинается с адреса 0x00.

4. В этой главе в язык Vole была введена машинная команда вида `0xDR0S`. Предположим, что эта форма была расширена до `0xDRXS`, что означает "Загрузить в регистр `R` данные, адрес которых можно определить как содержимое регистра `S` плюс значение `X`". Иначе говоря, указатель на данные определяется посредством извлечения значения из регистра `S` и последующего увеличения этого значения на `X`, при этом значение в регистре `S` не изменяется. (Так, если регистр `F` содержит комбинацию битов `0x04`, то после выполнения команды `0xDE2F` в регистр `E` будет помещено содержимое ячейки памяти с адресом `0x06`, а значение в регистре `F` останется неизменным — `0x04`.) Какие преимущества дает использование такой команды? А что можно сказать в отношении команды вида `0xDRTS`, означающей "Загрузить в регистр `R` данные, адрес которых определяется значением в регистре `S`, увеличенными на значение, содержащееся в регистре `T`"?

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. Изобразите, как приведенный ниже массив будет выглядеть в машинной памяти, если записать его с разверткой по строкам и столбцам.

A	B	C	D
E	F	G	H
I	J	K	L

2. Предположим, что массив из 6 строк и 8 столбцов записан в память с разверткой по строкам, начиная с адреса `20` (десятичное). Если для каждого элемента массива требуется одна ячейка памяти, каким будет адрес элемента из третьей строки четвертого столбца? Каков будет этот адрес, если каждый элемент будет занимать две ячейки?
3. Решите предыдущую задачу, предполагая, что массив записан с разверткой по столбцам.
4. Какие сложности могут возникнуть при попытке реализовать динамический список, используя традиционный одномерный массив?
5. Опишите метод хранения трехмерных массивов. Какой адресный полином будет использоваться для определения местоположения записи в *i*-й плоскости, *j*-й строке и *k*-м столбце?
6. Предположим, что список из букв `A`, `B`, `C`, `E`, `F` и `G` записан в непрерывном блоке ячеек памяти. Какие действия потребуется выполнить, чтобы включить букву `D` в этот список, сохранив его алфавитный порядок?
7. Приведенная ниже таблица представляет адреса и содержимое некоторых ячеек основной памяти машины. Обратите внимание, что некоторые из этих ячеек содержат буквы алфавита и что за каждой такой ячейкой следует пустая. Поместите в пустые ячейки адреса, причем так, чтобы каждая содержащая букву ячейка вместе с последующей образовали элемент связанного списка, в котором буквы следуют в алфавитном порядке. (Для указателя `null` используйте значение `0`.) Какой адрес будет содержать указатель головы списка?

```
Адрес     Содержимое
0x11         'C'
0x12
0x13         'G'
0x14
```

```
0x15          'E'
0x16
0x17          'B'
0x18
0x19          'U'
0x1A
0x1B          'F'
0x1C
```

8. Приведенная ниже таблица представляет собой часть связанного списка, размещенного в основной памяти машины. Каждый элемент этого списка состоит из двух ячеек: первая содержит букву алфавита, а вторая — указатель на следующий элемент списка. Измените указатели так, чтобы буква N больше не находилась в этом списке. Затем замените букву N буквой G и измените указатели так, чтобы новая буква размещалась в списке на положенном ей месте в алфавитном порядке.

```
Адрес       Содержимое
0x30           'J'
0x31           0x38
0x32           'B'
0x33           0x30
0x34           'X'
0x35           0x41
0x36           'N'
0x37           0x3A
0x38           'K'
0x39           0x36
0x3A0          'P'
0x3B           0x34
```

9. Приведенная ниже таблица представляет собой связанный список, имеющий тот же формат, что и в предыдущем задании. Если указатель головы списка содержит значение 0x44, какое имя представляется этим списком? Измените указатели так, чтобы список содержал имя Jean.

```
Адрес       Содержимое
0x40           'N'
0x41           0x46
0x42           'I'
0x43           0x40
0x44           'J'
0x45           0x4A
0x46           'E'
0x47           0x00
0x48           'M'
0x49           0x42
0x4A           'A'
0x4B           0x40
```

10. Какая из следующих процедур правильно включает в связанный список элемент `NewEntry` непосредственно после элемента `PreviousEntry`? Что именно неправильно в другой процедуре?

Процедура 1

```
1. Скопировать значение из поля указателя
   элемента PreviousEntry в поле
   указателя элемента NewEntry.
2. Изменить значение в поле указателя
   элемента PreviousEntry на
   адрес элемента NewEntry.
```

Процедура 2

```
1. Изменить значение в поле указателя
   элемента PreviousEntry на адрес
   элемента NewEntry.
2. Скопировать значение из поля указателя
   элемента PreviousEntry в поле
   указателя элемента NewEntry.
```

11. Разработайте функцию для конкатенации двух связанных списков (т.е. размещения одного списка вслед за другим с образованием единого списка).

12. Разработайте функцию объединения двух отсортированных списков в формате непрерывного блока ячеек памяти в один непрерывный отсортированный список. Как изменится эта процедура, если списки будут иметь связанный формат?

13. Разработайте функцию изменения порядка элементов связанного списка на обратный.

14. а. Разработайте алгоритм печати связанного списка в обратном порядке с использованием стека в качестве дополнительной структуры памяти.

б. Разработайте рекурсивную процедуру, позволяющую решить эту задачу без явного использования стека. В какой форме стек все же будет использоваться в новом рекурсивном варианте решения?

15. Иногда один и тот же связанный список представляется упорядоченным в двух направлениях посредством присоединения к каждому элементу двух указателей вместо одного. Заполните приведенную ниже таблицу так, чтобы, следуя первым указателям, помещаемым непосредственно после каждой буквы, получалось имя `Carol`, а следуя вторым указателям, буквы были упорядочены в алфавитном порядке. Какие значения находятся в указателе головного элемента каждого из этих представлений?

```
Адрес     Содержимое
0x60         'O'
0x61
```

```
0x62
0x63          'C'
0x64
0x65
0x66          'A'
0x67
0x68
0x69          'L'
0x6A
0x6B
0x6C          'R'
0x6D
0x6E
```

16. Приведенная ниже таблица представляет собой стек, записанный в непрерывном блоке ячеек памяти, в формате, описанном выше в тексте главы. Если база стека находится по адресу `10`, а указатель вершины стека содержит значение `12`, какой символ будет получен с помощью операции извлечения? Какое значение после этой операции будет иметь указатель вершины стека?

```
Адрес      Содержимое
0x10          'F'
0x11          'C'
0x12          'A'
0x13          'B'
0x14          'E'
```

17. Начертите таблицу, показывающую конечное содержимое ячеек памяти, заменив в предыдущей задаче операцию извлечения элемента из стека операцией вставки в него буквы `D`. Каким будет значение указателя вершины стека после выполнения операции вставки?

18. Разработайте процедуру удаления нижнего элемента стека таким образом, чтобы остальная часть стека сохранялась. Доступ к стеку возможен только с использованием операций вставки и извлечения. Какую вспомогательную структуру хранения следует использовать для решения поставленной задачи?

19. Разработайте процедуру сравнения содержимого двух стеков.

20. Предположим, вам дали два стека. Если бы вам было разрешено только перемещать записи по одной из одного стека в другой, какие перестановки исходных данных были бы возможны? Какие перестановки были бы возможны, если бы вам дали три стека?

21. Предположим, вам дали три стека и было разрешено перемещать записи из одного стека в другой только по одной за раз. Разработайте алгоритм для перестановки двух соседних элементов в одном из стеков.

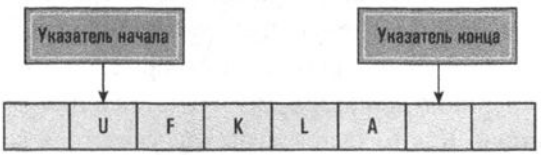
22. Предположим, что требуется создать стек имен различной длины. Почему в этом случае выгоднее сохранять имена в отдельных областях памяти, а затем строить стек из указателей на эти имена, чем просто помещать в стек сами имена?

23. В каком направлении очередь дрейфует в памяти — в направлении ее головы или в направлении ее хвоста?

24. Предположим, необходимо реализовать "очередь", в которой новым элементам сначала присваивается некоторый приоритет, а затем этот новый элемент помещается в очередь так, чтобы он находился перед всеми элементами с более низким приоритетом. Опишите систему хранения для реализации такой "очереди" и обоснуйте свое решение.

25. Предположим, что каждый элемент очереди занимает одну ячейку памяти, указатель начала очереди содержит значение `11`, а указатель конца очереди — значение `17`. Какими будут значения этих указателей после того, как один элемент будет включен в очередь, а два будут удалены?

26. а. Предположим, что очередь, реализованная как циклическая структура, находится в изображенном ниже состоянии. Начертите схему, показывающую ее структуру после того, как в очередь будут включены буквы `G` и `R`, затем три буквы удалены, а потом включены буквы `D` и `P`.



б. Какая ошибка возникнет при выполнении задания из предыдущего пункта, если буквы `G`, `R`, `D` и `P` будут включены до того, как из очереди будут удалены какие-либо буквы?

27. Опишите, как массив может быть использован для реализации очереди средствами языка высокого уровня.

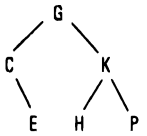
28. Предположим, вам дали две очереди и было разрешено перемещать только один элемент за раз из головы очереди в конец любой из них. Разработайте алгоритм перестановки двух соседних элементов в одной из очередей.

29. Приведенная ниже таблица представляет дерево, записанное в машинной памяти. Каждая вершина дерева состоит из трех ячеек. Первая ячейка содержит данные (букву), вторая — указатель на левую дочернюю вершину данной вершины, а третья — указатель на ее правую дочернюю вершину. Значение `0` представляет указатель `null`. Нарисуйте это дерево, если значение указателя корневой вершины равно `55`.

Адрес	Содержимое
0x40	'G'
0x41	0x0
0x42	0x0
0x43	'X'
0x44	0x0
0x45	0x0
0x46	'J'
0x47	0x49
0x48	0x0
0x49	'M'
0x4A	0x0
0x4B	0x0
0x4C	'F'
0x4D	0x43
0x4E	0x40
0x4F	'W'
0x50	0x46
0x51	0x4C

30. Приведенная ниже таблица представляет содержимое блока ячеек основной памяти машины. Обратите внимание, что некоторые ячейки содержат буквы алфавита и что за каждой такой ячейкой следуют две пустые ячейки. Заполните пустые ячейки таким образом, чтобы этот блок памяти представлял дерево, представленное на рисунке ниже и организованное следующим образом: первая ячейка, следующая за каждой буквой, содержит указатель на левую дочернюю вершину данной вершины, а вторая — указатель на ее правую дочернюю вершину. Для представления указателя `null` используйте значение `0`. Каким будет значение указателя корневой вершины?

Адрес	Содержимое
0x30	'C'
0x31	
0x32	
0x33	'H'
0x34	
0x35	
0x36	'K'
0x37	
0x38	
0x39	'E'
0x3A	
0x3B	
0x3C	'G'
0x3D	
0x3E	
0x3F	'P'
0x40	
0x41	



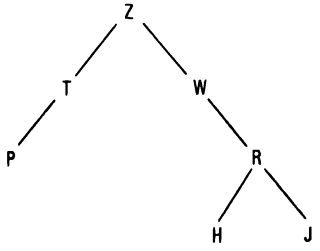
31. Разработайте нерекурсивный алгоритм на смену рекурсивному алгоритму, представленному на рис. 8.21.

32. Разработайте нерекурсивный алгоритм распечатки бинарного дерева на смену рекурсивному алгоритму, представленному на рис. 8.24. Для управления возвратами управления, которые могут потребоваться, используйте стек.

33. Примените рекурсивный алгоритм распечатки бинарного дерева, приведенный на рис. 8.24, к дереву, описанному выше, в задании 29. Начертите схему вложенных активаций алгоритма (и текущую позицию в каждой из них) для печати вершины X.

34. Оставив прежней корневую вершину и не меняя физического расположения элементов данных в дереве, описанном в задании 29, измените его указатели так, чтобы рекурсивный алгоритм печати бинарного дерева (см. рис. 8.24) распечатал его вершины в алфавитном порядке.

35. Начертите схему, показывающую, как выглядит в основной памяти приведенное ниже бинарное дерево, если оно сохранено без использования указателей в блоке непрерывно расположенных ячеек памяти, как было описано в разделе 8.3.



36. Предположим, что непрерывно расположенные ячейки, представляющие бинарное дерево в формате, описанном в разделе 8.3, содержат значения A, B, C, D, E, F и G соответственно. Нарисуйте это дерево.

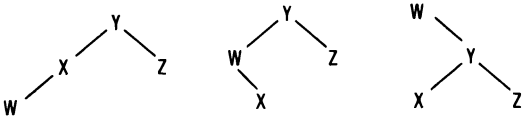
37. Приведите пример, в котором вы можете реализовать список (концептуальная структура) в виде дерева (действительно используемая базовая структура). Приведите пример, в котором вы можете реализовать дерево (концептуальная структуру) в виде списка (действительно используемая базовая структура).

38. Связанные древовидные структуры, обсуждавшиеся выше в этой главе, содержали указатели, позволяющие перемещаться по дереву вниз, от

родительских вершин к дочерним. Опишите систему указателей, которая позволила бы перемещаться по дереву вверх, от дочерних вершин к родительским. Что можно сказать о перемещении между вершинами-близницами?

39. Опишите структуру данных, пригодную для представления позиции на шахматной доске во время игры.

40. Укажите, для какого из представленных ниже деревьев алгоритм печати бинарного дерева, представленный на рис. 8.24, напечатает его вершины в алфавитном порядке.



41. Модифицируйте представленную на рис. 8.24 функцию таким образом, чтобы "список" печатался в обратном порядке.

42. Опишите структуру дерева, которое можно было бы использовать для записи сведений о генеалогии семьи. Какие операции могут выполняться с этим деревом? Если это дерево необходимо реализовать в виде связанной структуры, то какие указатели должны быть связаны с каждой из его вершин? Разработайте процедуры, выполняющие установленные вами выше операции, полагая, что дерево реализовано в виде связанной структуры с определенными вами указателями. Используя разработанную структуру хранения, объясните, как можно найти всех братьев и сестер любой из персон, представленных на генеалогическом дереве.

43. Разработайте процедуру поиска и удаления заданного значения из дерева, представленного в памяти так, как показано на рис. 8.20.

44. В традиционной реализации дерева каждый узел создается с отдельным указателем для каждого возможного дочернего элемента. Количество таких указателей является проектным решением и представляет максимальное количество дочерних элементов, которое может иметь любой узел. Если у узла меньше дочерних элементов, чем было зарезервировано указателей, неиспользуемые просто устанавливаются в нуль. Однако у таких узлов никогда не может быть дочерних элементов больше, чем указателей. Опишите, как дерево может быть реализовано без ограничения количества дочерних элементов, которые может иметь каждый узел.

45. Используя псевдокод и воспользовавшись как образцом оператором `struct` языка C, представленным в разделе 8.5, запишите определяемый пользователем тип данных, представляющий сведения о сотруднике компании (например, имя, адрес, занимаемая должность, размер оплаты и т.д.).

46. Используя псевдокод и воспользовавшись как образцом синтаксисом определения класса на языке Java, представленного на рис. 8.27, сформулируйте схематичное определение абстрактного типа данных, представляющего список имен. В частности, в какой структуре будет реализован список и какие функции манипулирования списком необходимо будет предоставить? (Давать детальное описание функций не требуется.)

47. Используя псевдокод и воспользовавшись как образцом синтаксисом определения класса на языке Java, представленного на рис. 8.27, сформулируйте схематичное определение абстрактного типа данных, представляющего очередь. Затем приведите инструкции на псевдокоде, описывающие, как могут быть созданы реализации этого типа и как отдельные элементы могут быть вставлены в эти реализации или удалены из них.

48. **а.** В чем различие между типом данных, определяемым пользователем, и базовым (примитивным) типом данных?

б. В чем различие между типом данных, определяемым пользователем, и абстрактным типом данных?

49. Укажите структуры данных и процедуры их обработки, которые могут использоваться в абстрактном типе данных, представляющем адресную книгу.

50. Укажите структуры данных и процедуры их обработки, которые могут использоваться в абстрактном типе данных, представляющем боевой космический корабль в простой компьютерной игре.

51. Модифицируйте определение класса, приведенное на рис. 8.27, и интерфейс `StackType`, представленный в разделе 8.5, таким образом, чтобы класс определял очередь, а не стек.

52. В каком смысле класс является более общей категорией в сравнении с традиционным абстрактным типом данных?

***53.** Используя команды расширения машинного языка `0xDR0S` и `0xER0S`, описанные в конце раздела 8.7, напишите на машинном языке Vole полную процедуру для вставки элемента в стек, реализованный по схеме, представленной на рис. 8.12. Предполагается, что указатель вершины стека находится в регистре `F`, а вставляемые данные — в регистре `5`.

***54.** Предположим, что каждый элемент связанного списка состоит из одной ячейки памяти с данными, за которой следует указатель следующего элемента списка. Также предположим, что новый элемент, расположенный в ячейке памяти с адресом `0xA0`, должен быть включен между элементами, размещенными в ячейках с адресами `0xB5` и `0xC4`. Используя машинный язык, описанный в приложении В, и дополнительные машинные

команды с кодами операции `0xD` и `0xE`, описанные в конце раздела 8.7, напишите на машинном языке процедуру, выполняющую требуемое включение элемента.

*55. Какие преимущества имеет машинная команда вида `0xDR0S`, представленная в разделе 8.7, в сравнении с машинной командой вида `0xDRXY`? Какое преимущество имеет машинная команда вида `0xDRXS`, представленная в упражнении 4 в конце раздела 8.7, в сравнении с машинной командой вида `0xDR0S`?

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также понять, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. Предположим, что программист-аналитик, работающий на определенную компанию, разрабатывает способ организации данных, обеспечивающий их эффективную обработку в конкретном приложении. Как можно защитить права на эту структуру данных? Является ли структура данных выражением идеи (например, как стихотворение) и, следовательно, может быть защищена авторским правом, или структуры данных попадают в те же правовые лазейки, что и алгоритмы? А что можно сказать в отношении патентного права?
2. До какой степени можно считать, что неверная информация хуже, чем отсутствие информации?
3. Во многих прикладных программах пределы роста стека определяются количеством доступной памяти. Обычно программы разрабатываются так, что, если доступная память закончится, они выдают сообщение о переполнении стека с последующим завершением работы. В большинстве случаев такая ошибка никогда не возникает и пользователю ничего не известно о ее существовании. Кто должен нести ответственность, если в результате возникновения подобной ошибки будут утрачены важные данные? Как разработчик программного обеспечения может минимизировать свою ответственность?

4. В структуре данных, построенной на использовании указателей, удаление данных обычно заключается в изменении указателя, а не в очистке ячеек памяти. Следовательно, при удалении элемента из связанного списка он на самом деле остается в памяти до тех пор, пока занимаемое им место не понадобится для других данных. Какие этические проблемы и проблемы безопасности могут иметь место в результате такой "живучести" удаляемых данных?
5. Перенести данные и программы с одного компьютера на другой достаточно легко. Следовательно, знания, хранящиеся в одной машине, легко перенести на многие другие. Человеку, напротив, иногда трудно передать свои знания другому. Например, чтобы научить одного человека какому-то языку, другому понадобится достаточно много времени. Каковы могут быть последствия подобных отличий в скорости передачи знаний, если возможности машин станут сравнимы с возможностями человека?
6. Использование указателей позволяет объединять связанные данные в памяти компьютера таким образом, который напоминает способ, которым, по мнению многих, в человеческом разуме устанавливаются ассоциации между элементами информации. Насколько такие ссылки в памяти компьютера напоминают ассоциативные связи в мозге? В какой степени они, по вашему мнению, различаются? Этично ли пытаться создавать компьютеры, которые максимально близко имитируют человеческий разум?
7. Способствовала ли популяризация компьютерных технологий возникновению новых этических проблем или же она просто создала новый контекст, в котором применимы прежние этические теории?
8. Предположим, что автор вводного курса по компьютерным наукам хочет включить в него примеры программ для демонстрации излагаемых концепций. Однако, чтобы достичь необходимой ясности, многие из этих примеров должны представлять собой упрощенные версии того кода, который действительно может быть использован в программном обеспечении профессионального качества. Автор понимает, что его примеры могут впоследствии быть скопированы ничего не подозревающими читателями и в конечном итоге попасть в реальные ответственные приложения, в которых лучше было бы использовать более надежные решения. Следует ли автору использовать упрощенные примеры или настаивать на том, чтобы все примеры были достаточно надежными, даже если это снижает их демонстративную ценность? Может, ему следует вообще отказаться от использования таких примеров, в которых не удается достичь необходимой ясности при достаточной надежности?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Carrano F.M., Henry T. *Data Abstraction and Problem Solving with C++: Walls and Mirrors*, 7th ed. — Boston, MA: Addison-Wesley, 2016. (Имеется русский перевод третьего издания этой книги: Каррано Ф.М., Причард Дж.Дж. *Абстракция данных и решение задач на C++. Стены и зеркала*, 3-е изд. — М.: Издательский дом "Вильямс", 2003.)
2. Goodrich M.T., Tamassia R., Goldwasser M.H. *Data Structures and Algorithms in Python*. — Hoboken, NJ: Wiley, 2013.
3. Gray S. *Data Structures in Java: From Abstract Data Types to the Java Collections Framework*. — Boston, MA: Addison-Wesley, 2007.
4. Lee K.D., Hubbard S. *Data Structures and Algorithms with Python*. — Cham, Switzerland: Springer, 2015.
5. Main M. *Data Structures and Other Objects Using Java*, 4th ed. — Boston, MA: AddisonWesley, 2011. (Имеется русский перевод второго издания этой книги: Мейн М., Савитч У. *Структуры данных и другие объекты в C++*, 2-е изд. — М.: Издательский дом "Вильямс", 2002.)
6. Main M., Savitch W. *Data Structures and Other Objects Using C++*, 4th ed. — Boston, MA: Addison-Wesley, 2010.
7. Prichard J., Carrano F.M. *Data Abstraction and Problem Solving with Java: Walls and Mirrors*, 3rd ed. — Boston, MA: Addison-Wesley, 2010.
8. Shaffer C.A. *Practical Introduction to Data Structures and Algorithm Analysis*, 2nd ed. — Upper Saddle River, NJ: Prentice Hall, 2001.
9. Weiss M.A. *Data Structures and Problem Solving Using Java*, 4th ed. — Boston, MA: Addison-Wesley, 2011.
10. Weiss M.A. *Data Structures and Algorithm Analysis in C++*, 4th ed. — Boston, MA: Addison-Wesley, 2013.
11. Weiss M.A. *Data Structures and Algorithm Analysis in Java*, 3rd ed. — Boston, MA: Addison-Wesley, 2011.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Браунси К. *Структуры данных в C++. Краткий справочник*. — М.: Издательский дом "Вильямс", 2002.
2. Ахо А.В., Хопкрофт Д., Ульман Д.Д. *Структуры данных и алгоритмы*. — М.: Издательский дом "Вильямс", 2000.

Глава 9

База данных — это система, которая преобразует большой набор данных в абстрактный инструмент, обеспечивающий пользователям возможность искать и извлекать необходимые элементы информации удобным для них способом. В данной главе эта тема будет подробно рассмотрена, а в качестве дополнительной информации будет предоставлен краткий обзор двух смежных областей: "добыча данных" — так называют интеллектуальный анализ больших объемов информации с целью обнаружения скрытых в ней шаблонов и тенденций, а также традиционные файловые структуры, реализующие многие инструментальные механизмы, лежащие в основе современных баз данных и систем интеллектуального анализа данных.

Системы баз данных

9.1. ОБЩИЕ ПОНЯТИЯ
- Значение систем баз данных
- Роль схемы в базе данных
- СУБД — системы управления базами данных
- Модели баз данных

9.2. РЕЛЯЦИОННАЯ МОДЕЛЬ
- Реляционное проектирование
- Реляционные операции
- Язык SQL

*9.3. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ БАЗЫ ДАННЫХ

*9.4. ОБЕСПЕЧЕНИЕ ЦЕЛОСТНОСТИ БАЗ ДАННЫХ
- Протокол фиксации/отката изменений
- Механизм блокировок

*9.5. ТРАДИЦИОННЫЕ ФАЙЛОВЫЕ СТРУКТУРЫ
- Последовательные файлы
- Индексированные файлы
- Хешированные файлы

9.6. ИНТЕЛЛЕКТУАЛЬНЫЙ АНАЛИЗ ДАННЫХ

9.7. ВЛИЯНИЕ ТЕХНОЛОГИЙ БАЗ ДАННЫХ НА ОБЩЕСТВО

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

Современные технологии позволяют хранить чрезвычайно большие объемы информации, но такие большие наборы данных будут бесполезны, если отсутствует возможность с помощью необходимых вычислительных средств извлечь из них те конкретные элементы информации, которые имеют отношение к поставленной задаче. В этой главе вы познакомитесь с системами баз данных и узнаете, как эти системы применяют абстракцию для преобразования больших конгломератов данных в полезные источники информации. В качестве смежной темы также будет рассмотрена быстро расширяющаяся область интеллектуального анализа больших объемов данных (или "добыча данных", как ее часто называют), целью которой является разработка методов выявления и изучения скрытых шаблонов и тенденций, существующих в больших совокупностях накопленных данных. В завершение будут рассмотрены принципы организации традиционных файловых структур, положенных в основу реализации современных систем баз данных и интеллектуального анализа данных.



Основные положения для запоминания

- **Эффективное использование больших наборов данных требует соответствующих вычислительных решений.**

9.1. Общие понятия

Термин **база данных** относится к совокупности данных, которая является многомерной в том смысле, что внутренние ссылки между элементами данных делают информацию в ней доступной с различных точек зрения. Такая возможность отличает базу данных от традиционной файловой системы (раздел 9.5), которую иногда называют **плоским файлом**, представляющим собой *одномерную* систему хранения. Последнее означает, что, в отличие от базы данных, в плоском файле информация всегда представлена и доступна с единственной точки зрения. Так, если плоский файл будет содержать информацию о композиторах и их произведениях, то из него можно будет извлечь только список произведений, упорядоченный по их авторам, тогда как база данных с той же информацией позволит получить сведения обо всех произведениях одного композитора, данные обо всех композиторах, которые писали музыку в определенном музыкальном стиле, и, возможно, имена композиторов, которые написали вариации на произведение другого композитора.

Значение систем баз данных

Исторически сложилось так, что, хотя вычислительная техника находила все более и более широкое применение в информационных системах управления, первоначально преобладала тенденция создания приложений в виде отдельных систем, использующих собственный набор данных. Так, начисление заработной платы осуществлялось с использованием файла расчета заработной платы, отдел кадров вел собственные записи о сотрудниках, а управление запасами осуществлялось с помощью файла остатков на складах. Это означало, что большая часть информации, необходимой для работы организации, была многократно продублирована по всей ее структуре, в то время как множество различных, но связанных между собой элементов данных хранились в разных системах. В такой ситуации системы баз данных появились, прежде всего, как средство интеграции информации, хранящейся и обрабатываемой в конкретной организации (рис. 9.1). С помощью такой системы одни и те же данные о продажах можно было использовать для формирования заказов на пополнение запасов, создания отчетов о тенденциях рынка, реализации адресной рекламы и рассылки сведений о новых изделиях тем клиентам, которые с наибольшей вероятностью отреагируют на такую информацию, и даже для расчета премиальных сотрудникам отдела сбыта.



Основные положения для запоминания

- **Базы данных способствуют эффективной организации информации и выявлению существующих в ней тенденций.**

Такие интегрированные пулы информации представляют собой ценный ресурс, с помощью которого удобно принимать управленческие решения, поскольку доступ к информации можно осуществлять заранее обдуманным образом. Со своей стороны, исследования в области технологии баз данных также часто концентрировались на разработке методов, с помощью которых информация из базы данных могла бы быть представлена в виде, упрощающем процесс принятия решений. И в этом отношении был достигнут большой прогресс. Сегодня технология баз данных в сочетании с методами интеллектуального анализа данных является важным инструментом управления, позволяющим руководству организации извлекать новые знания и оценки из огромных объемов данных, охватывающих все аспекты работы организации и ее окружения.

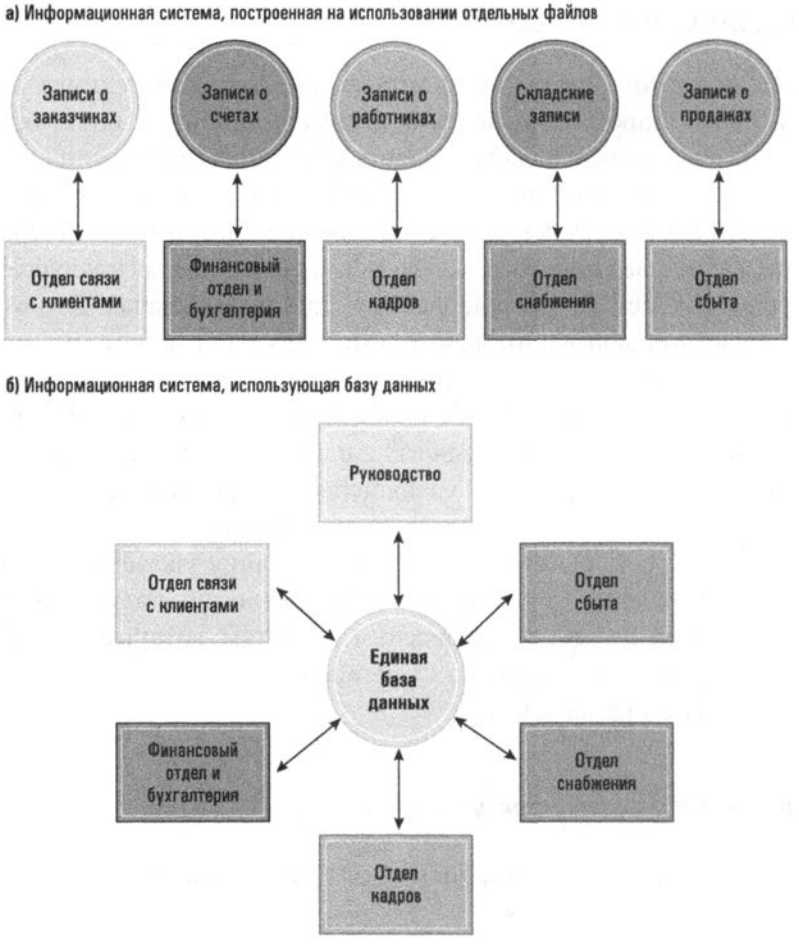


Рис. 9.1. Построение обычной файловой системы и системы базы данных

Основные положения для запоминания

- **Новые знания и углубленное понимание ситуации можно получить путем анализа и преобразования информации, представленной в цифровом виде.**

Более того, системы баз данных стали той технологией, которая поддерживает многие из наиболее популярных сайтов в Интернете. Основная тема таких сайтов, как Google, eBay и Amazon, — предоставление клиентам удобного интерфейса к базам данных. Чтобы ответить на запрос клиента, сервер опрашивает базу данных, представляет полученные результаты в виде веб-страницы

и отправляет эту страницу клиенту. Такие веб-интерфейсы популяризировали новую роль технологии баз данных, в которой база данных больше не является лишь средством хранения информации компании, а скорее представляет собой *продукт* компании. Современные базы данных хранят не только сведения о транзакциях, но также текстовый и мультимедийный контент всех типов, а также метаданные. И действительно, благодаря объединению технологий баз данных с веб-интерфейсами Интернет в настоящее время стал основным источником информации во всем мире.



Основные положения для запоминания

- **Большие наборы данных включают в себя такие данные, как сведения о транзакциях, результаты измерений, текстовая информация, звук, изображения и видео.**

Метаданные — это данные о данных

Термин **метаданные** относится к информации о других видах данных. В прежние времена, до того как компьютерные каталоги получили широкое распространение, типичной формой представления метаданных были выдвижные ящики с библиотечными карточками, предназначенные для быстрого поиска книг на полках библиотек по автору, названию, теме или другим известным лишь посвященным критериям. Компьютерные метаданные могут использоваться для описания изображений (информация о том, где была сделана фотография, с помощью какого устройства или кем) и мультимедийных файлов (автор, альбом, название песни и т.д.) или для предоставления семантических очередей для веб-страниц. Многие распространенные форматы файлов данных теперь включают в себя метаданные, внедренные в сам файл, такие как информация об авторе документа или истории внесения изменений для облачных ресурсов.

Метаданные позволяют существенно расширить возможности вычислительных инструментов в отношении анализа данных, предоставляя для анализа дополнительные структурные и информационные подсказки.

Когда в 2013 году Эдвард Сноуден публично объявил, что Агентство национальной безопасности США в течение многих лет занималось сбором метаданных о телефонных звонках американцев, сторонники этой практики поспешили указать на то, что содержание самих телефонных разговоров не записывалось. Критики ответили им, перечислив причины, по которым сбор одних только метаданных — номеров телефонов, продолжительности звонка и т.д. — уже можно считать существенным вторжением в личную жизнь без разрешающего постановления суда.



Основные положения для запоминания

- **Метаданные — это данные о данных.**
- **Метаданные могут представлять собой описательные данные об изображении, веб-странице или других сложных объектах.**
- **Метаданные могут повысить эффективность использования данных или наборов данных, предоставляя дополнительную информацию о различных аспектах этих данных.**

Роль схемы в базе данных

Одним из недостатков широкого распространения технологии баз данных является потенциальная возможность доступа к конфиденциальным данным со стороны неавторизованного персонала. Тот, кто размещает заказ на веб-сайте компании, не должен иметь доступа к финансовым данным этой компании. Аналогичным образом работник расчетного сектора бухгалтерии обязательно должен иметь доступ к информации о сотрудниках компании, но он не должен иметь доступа к данным о ее складских запасах или о реализации ее продукции. Поэтому возможность контролировать доступ к информации в базе данных часто имеет не меньшее значение, чем возможность предоставлять совместный доступ к ней.

Для предоставления различным пользователям доступа к различным данным в системах баз данных чаще всего применяются схемы и подсхемы. **Схема** представляет собой полное описание структуры базы данных, которое используется ее программным обеспечением для обслуживания базы данных в целом. **Подсхема** — это описание только той части базы данных, которая соответствует нуждам определенного пользователя. Например, схема базы данных университета может указывать, что запись о каждом из студентов, помимо информации об успеваемости, содержит его телефон и текущий адрес места жительства. Более того, схема может указывать, что запись о каждом студенте связана с записью его куратора. В свою очередь, запись о каждом сотруднике факультета включает его адрес, послужной список и т.п. Исходя из этой схемы, в базе данных поддерживается система указателей, которая необходимым образом связывает информацию о студенте с послужным списком его куратора.

Чтобы воспрепятствовать использованию подобных связей для получения конфиденциальной информации о факультете, права доступа к базе данных для рядового делопроизводителя должны быть ограничены с помощью некоторой подсхемы, в которой описание информации о факультете не включает послужные списки сотрудников. Согласно этой подсхеме такой пользователь сможет определить, кто из сотрудников факультета является куратором определенного

студента, но не сможет получить доступ к дополнительной информации об этом сотруднике. И наоборот, подсхема для работника бухгалтерии должна включать доступ к послужному списку сотрудника, но не должна устанавливать никакой связи между записями сотрудников и студентов. В результате бухгалтер сможет изменить заработную плату сотрудника, но не будет иметь возможности узнать имена студентов, курируемых этим сотрудником.

Хотя подсхемы являются полезным инструментом для управления доступом к конфиденциальной информации, на практике защитить очень большие, динамические и распределенные базы данных достаточно сложно. Взаимодействие между перекрывающимися подсхемами, бреши в физической или сетевой безопасности системы, а также ошибки людей-операторов — все это создает возможности для злоупотребления содержимым защищенной базы данных.



Основные положения для запоминания

- **При создании и использовании вычислительных систем и средств идентификации могут возникнуть проблемы с обеспечением их конфиденциальности и безопасности.**
- **Сохранение конфиденциальности больших наборов данных, содержащих личную информацию, может оказаться достаточно сложной задачей.**

СУБД — системы управления базами данных

Типичное приложение системы управления базой данных (СУБД) включает в себя несколько уровней программного обеспечения, которые можно сгруппировать в два основных слоя — уровень прикладного программного обеспечения и уровень системы управления базой данных, как показано на рис. 9.2. Прикладное программное обеспечение отвечает за взаимодействие пользователя с базой данных и может быть достаточно сложным, что подтверждается, например, приложениями, в которых пользователи получают доступ к базе данных через веб-сайт. В этом случае прикладной уровень в целом состоит из множества клиентов, разбросанных по всему Интернету, и сервера, который использует базу данных для выполнения поступающих от клиентов запросов.

Обратите внимание, что прикладное программное обеспечение не манипулирует базой данных напрямую. В действительности манипуляция данными осуществляется программным обеспечением следующего уровня, называемым **системой управления базой данных (СУБД)**. После того как прикладное программное обеспечение определило, какое действие запрашивает пользователь, оно использует СУБД в качестве абстрактного инструмента для получения

требуемых результатов. Если запрос заключается в добавлении или удалении данных, то именно СУБД фактически изменяет базу данных. Если запрос заключается в получении информации, то именно СУБД выполняет требуемый поиск.



Рис. 9.2. Концептуальные уровни построения системы управления базой данных

Такое разделение на прикладное программное обеспечение и СУБД имеет несколько преимуществ. Одно из них состоит в том, что подобное разделение позволяет создавать и использовать абстрактные инструменты, которые, в чем вы уже неоднократно убедились, являются основной упрощающей концепции в разработке программного обеспечения. Если детали того, как данные действительно хранятся в базе, изолированы в СУБД, разработка прикладного программного обеспечения существенно упрощается. Например, при хорошо спроектированной СУБД прикладное программное обеспечение не должно заботиться о том, хранится ли база данных на одном компьютере или разбросана по многим машинам в сети как **распределенная база данных**. В действительности сама СУБД будет решать все эти проблемы, позволяя прикладному программному обеспечению получать доступ к базе данных, не заботясь о том, где и как эти данные хранятся.

Второе преимущество отделения прикладного программного обеспечения от СУБД состоит в том, что такая организация предоставляет средства для контроля доступа к базе данных. Когда весь доступ к базе данных осуществляется только через СУБД, последняя получает возможность реализовать любые ограничения, накладываемые различными подсхемами. В частности, СУБД может использовать всю схему базы данных для своих внутренних нужд и при этом следить за тем, чтобы прикладное программное обеспечение, используемое каждым пользователем, имело доступ к данным исключительно в пределах, описанных подсхемой этого пользователя.

Еще одной причиной, по которой функции пользовательского интерфейса и функции собственно манипуляции данными следует разместить в двух различных уровнях программного обеспечения, является необходимость достичь

независимости данных, т.е. возможности вносить изменения в организацию базы данных без необходимости изменения прикладного программного обеспечения. Например, отделу кадров может потребоваться добавить к записи сотрудника новое поле, предназначенное для учета его участия в новой программе медицинского страхования. Если программное обеспечение приложения напрямую взаимодействует с базой данных, подобное изменение формата данных может потребовать модификации всех программных модулей, взаимодействующих с этой базой данных. В результате внесение изменений по требованию отдела кадров повлечет за собой изменения в программном обеспечении расчетного отдела или, скажем, в программном обеспечении печати почтовых наклеек для исходящей корреспонденции.

Распределенные базы данных

С ростом возможностей компьютерных сетей системы баз данных существенно выросли и теперь включают базы данных, известные как *распределенные базы данных*, т.е. состоящие из данных, размещенных на разных компьютерах. Например, международная корпорация может хранить и поддерживать локальные записи о сотрудниках на локальных сайтах, но связывать эти записи через сеть с образованием единой распределенной базы данных.

Распределенная база данных может содержать фрагментированные и/или реплицированные данные. Предыдущий пример с записями о сотрудниках служит образцом того случая, когда различные фрагменты базы данных хранятся в разных местах. Во втором случае дубликаты одного и того же компонента базы данных хранятся в различных местах. Подобная репликация может быть полезна в тех случаях, когда необходимо уменьшить время доступа к требуемой информации. В обоих случаях возникают проблемы, не характерные для более традиционных централизованных систем, — как скрыть распределенную природу базы данных, чтобы она функционировала как связная система, или как обеспечить точное соответствие реплицированных частей базы данных при обновлении данных. По этой причине распределенные базы данных являются одной из наиболее интенсивно исследуемых областей компьютерных наук.

Распределение функций между программным обеспечением приложения и СУБД устраняет всякую необходимость в подобном перепрограммировании. Для внесения изменений, необходимых для одного пользователя, потребуется лишь изменить общую схему и подсхемы тех пользователей, которые заинтересованы или нуждаются в подобной модификации. Все остальные подсхемы при этом остаются неизменными, и программное обеспечение приложений продолжает функционировать точно так же, как и до внесения изменений.

Модели баз данных

Вы уже неоднократно убеждались в том, что абстракция позволяет нам эффективно скрывать те или иные внутренние сложности. Системы управления базами данных предоставляют еще один подобный пример. Они позволяют скрыть всю сложность внутренней структуры базы данных, создавая у пользователя базы данных представление, что хранящаяся в ней информация, организована в наиболее удобном для него формате. В частности, СУБД содержит подпрограммы, обеспечивающие перевод команд, сформулированных в контексте концептуального представления базы данных, в необходимую последовательность действий, которые должны быть выполнены реальной системой хранения данных. Упомянутое выше концептуальное представление базы данных принято называть **моделью базы данных**.

В следующих разделах будут рассмотрены как модель реляционной базы данных, так и модель объектно-ориентированной базы данных. В случае модели реляционной базы данных концептуальным представлением базы данных является представление таблицы, состоящей из строк и столбцов. Например, информация о сотрудниках компании может рассматриваться как таблица, содержащая строку для каждого сотрудника и столбцы с именами, адресами, идентификационными номерами сотрудников и т.д. В свою очередь, СУБД будет содержать подпрограммы, которые позволят прикладному программному обеспечению выбрать определенные элементы данных из определенной строки таблицы или, возможно, передать сводную информацию о диапазоне значений, найденных в столбце зарплаты, даже если подобная информация в действительности не хранится непосредственно в строках и столбцах.

Эти подпрограммы и формируют абстрактные инструменты, используемые прикладным программным обеспечением для доступа к базе данных. Если говорить точнее, то прикладное программное обеспечение обычно пишется на одном из языков программирования общего назначения, таких как те, которые обсуждались в главе 6. Эти языки предоставляют основные элементы для построения алгоритмических выражений, но не включают элементов, предназначенных для манипулирования базой данных. Однако любая программа, написанная на одном из этих языков программирования, может использовать средства, предоставляемые СУБД в виде предварительно написанных подпрограмм, что фактически расширяет возможности языка таким образом, который обеспечивает поддержку концептуального представления модели базы данных.

Поиск лучших моделей баз данных — это непрекращающийся процесс. Цель его состоит в том, чтобы найти модели, которые позволят легко концептуализировать сложные системы данных, предоставят удобные способы выражения запросов на требуемую информацию и обеспечат возможность создания эффективных систем управления базами данных.

9.1. Вопросы и упражнения

1. Назовите два подразделения одного производственного предприятия, которые по-разному используют одну и ту же или сходную информацию о складских запасах. Опишите, как может различаться подсхема базы данных для этих двух отделов.
2. В чем назначение модели базы данных?
3. Охарактеризуйте роль прикладного программного обеспечения и программ СУБД.

9.2. Реляционная модель

В этом разделе вы познакомитесь с реляционной моделью базы данных. Данные в реляционной модели отображаются в виде прямоугольных таблиц, называемых **отношениями** (relation), которые похожи на формат отображения данных в электронных таблицах. Например, в реляционной модели информацию о сотрудниках некоторой фирмы можно представить в виде отношения, приведенного на рис. 9.3.

EmpId	Name	Address	SSN
25X15	Джо Бейкер	ул. Верхняя, 33	111223333
34Y70	Шерил Кларк	ул. Парковая, 5/63	999009999
23Y34	Джесси Смит	пер. Круглый, 15	111005555
•	•	•	•
•	•	•	•
•	•	•	•

Рис. 9.3. Отношение, содержащее сведения о сотрудниках

Строка в отношении называется **кортежем** (tuple). В отношении, представленном на рис. 9.3, каждый кортеж содержит информацию об одном сотруднике. Столбцы в отношении именуются **атрибутами** (attribute), поскольку каждый элемент столбца описывает некоторую характеристику (или атрибут) одной сущности, представленной соответствующим кортежем. В нашем примере каждый кортеж содержит атрибуты `EmplId` (Личный номер работника), `Name` (Имя и фамилия), `Address` (Адрес) и `SSN` (Номер полиса социального страхования).

Реляционное проектирование

Ключевым этапом в разработке реляционной базы данных является определение отношений, составляющих эту базу данных. Невзирая на кажущуюся простоту этой задачи, неискушенного разработчика поджидает множество коварных ловушек.

Предположим, что в дополнение к информации, содержащейся в отношении, представленном на рис. 9.3, необходимо включить сведения о должностях, которые занимали сотрудники. Поэтому для каждого из сотрудников потребуется ввести информацию о должностных перемещениях, включающую следующие атрибуты: `Jobtitle` — название должности (секретарь, начальник группы, начальник отдела), `JobId` — идентификационный код должности (уникальный для каждой должности), `SkillCode` — код требуемого уровня навыков (связанный с каждой должностью), `Dept` — подразделение, в котором работник занимал эту должность, начальная (`StartDate`) и конечная (`TermDate`) даты периода, в течение которого сотрудник занимал данную должность. Если сотрудник занимает данную должность и в настоящий момент, то вместо конечной даты должен указываться символ "звездочка".

Один из подходов к решению этой задачи состоит в расширении представленного на рис. 9.3 отношения путем добавления новых столбцов для дополнительных атрибутов. Результат применения этого подхода показан на рис. 9.4. Однако при более пристальном взгляде на полученное отношение можно заметить определенные осложнения. Одно из них заключается в потере эффективности в результате избыточности данных. В самом деле, полученное отношение уже не содержит по одному кортежу для каждого сотрудника — теперь один кортеж соответствует отдельному назначению определенного сотрудника на некоторую должность. Если работа сотрудника в компании сопровождалась его продвижением по службе, то в новом отношении ему будет соответствовать уже несколько кортежей, в которых информация об этом сотруднике (имя, адрес, идентификационный код и номер полиса социального страхования) будет многократно дублироваться. В приведенном примере дублируется информация о сотрудниках Бейкер и Смит, поскольку каждый из них занимал в компании больше одного поста. Более того, если какую-либо должность одновременно занимают несколько сотрудников подразделения, то название подразделения и код уровня необходимых навыков будут повторяться в каждом кортеже, соответствующем назначению отдельных работников на эту должность. В нашем примере описание должности начальника группы повторяется в отношении несколько раз, поскольку эту должность в компании занимает более одного работника.

EmpId	Name	Address	SSN	JobId	JobTitle	SkillCode	Dept	StartDate	TermDate
25X15	Джо Бейкер	ул. Верхняя, 33	111223333	F5	Начальник группы	FM3	Отдел сбыта	1-9-2012	30-9-2018
25X15	Джо Бейкер	ул. Верхняя, 33	111223333	D7	Начальник отдела	K2	Отдел сбыта	1-10-2018	*
34Y70	Шерил Кларк	ул. Парковая, 5/63	999009999	F5	Начальник группы	FM3	Отдел сбыта	1-10-2009	*
23Y34	Джесси Смит	пер. Круглый, 15	111005555	S25X	Секретарь	T5	Отдел кадров	1-3-2009	30-4-2016
23Y34	Джесси Смит	пер. Круглый, 15	111005555	S26Z	Секретарь	T6	Бухгалтерия	1-5-2016	*
•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•
•	•	•	•	•	•	•	•	•	•

Рис. 9.4. Отношение, содержащее избыточную информацию

Другая, возможно, более серьезная проблема, связанная с использованием расширенного подобным образом отношения, возникает при удалении информации из базы данных. Предположим, что единственным сотрудником, занимавшим должность с кодом D7, был Джо Бейкер. Если он покинет компанию и информация о нем будет удалена из базы данных, показанной на рис. 9.4, то будут утеряны и все сведения о должности с кодом D7. Действительно, единственным кортежем, содержащим сведения о том, что для должности с кодом D7 требуется уровень навыков с кодом K2, является кортеж, содержащий сведения о Джо Бейкере.

Вы можете возразить, что подобную проблему можно устранить путем удаления не всего кортежа, а только его части, но это может повлечь за собой другие осложнения. В частности, следует ли при удалении информации о Джо Бейкере сохранить частично стертый кортеж со сведениями о должности с кодом F5 или в каком-либо другом кортеже отношения так же содержатся сведения о ней? Более того, сама попытка использовать частично заполненные кортежи является верным признаком несовместимости выбранной структуры отношения и требований приложения.

Источником всех этих проблем является попытка использовать одно отношение для представления информации о более чем одном понятии предметной области. Предложенный на рис. 9.4 вариант расширенного отношения содержит информацию, имеющую отношение непосредственно к сотрудникам (имя, идентификационный номер, адрес, номер полиса социального страхования), информацию о штатном расписании компании (идентификационный код должности, ее название, подразделение, код необходимого уровня навыков) и хронологическую информацию о взаимосвязи сотрудников и должностей (начальная и конечная даты периода выполнения определенных должностных

обязанностей). Исходя из этих соображений можно решить возникшие проблемы за счет изменения структуры базы данных и включения в ее состав трех отношений: по одному отношению для каждого из перечисленных выше понятий. Исходное отношение EMPLOYEE (Сотрудники), представленное на рис. 9.3, можно оставить без изменений, а для хранения дополнительной информации нужно создать два новых отношения, JOB (Штатное расписание) и ASSIGNMENT (Назначения), как показано на рис. 9.5.

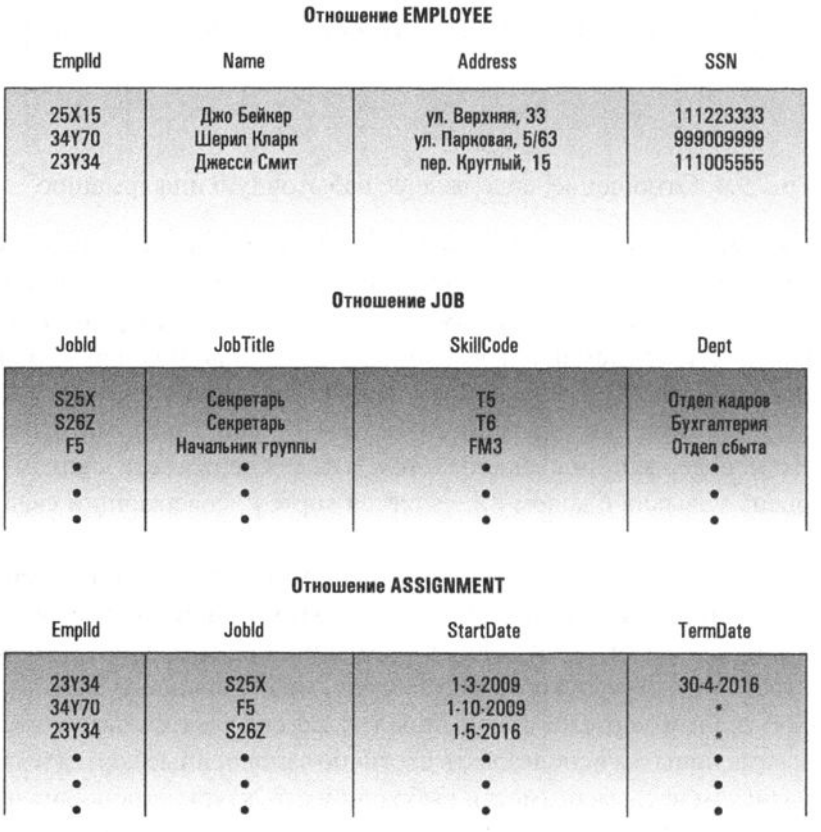


Рис. 9.5. База данных с информацией о сотрудниках, состоящая из трех отношений

База данных, состоящая из трех отношений, содержит необходимую информацию о сотрудниках (в отношении EMPLOYEE), штатном расписании (в отношении JOB) и должностном продвижении (в отношении ASSIGNMENT). Дополнительная информация может быть получена косвенно, за счет комбинирования данных из различных отношений. Например, если известен идентификационный номер работника, можно получить сведения обо всех подразделениях, в которых он работал, сначала выбрав из отношения ASSIGNMENT сведения обо

всех должностях, которые он занимал, а затем получив из отношения JOB наименования соответствующих подразделений, как показано на рис. 9.6. Таким образом, посредством процедур, подобных только что описанной, из базы данных, состоящей из трех отношений, можно извлечь любую информацию, которая могла бы быть доступна из единственного избыточного отношения, но уже без упомянутых ранее осложнений.

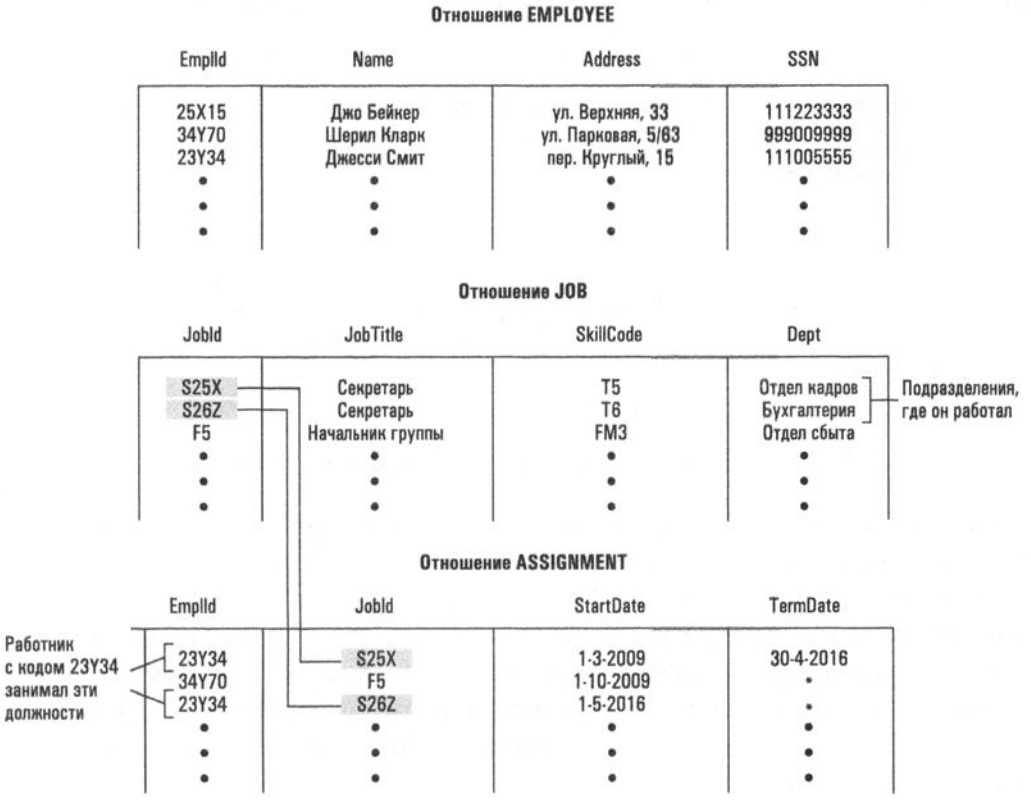


Рис. 9.6. Определение всех подразделений, в которых работал сотрудник с личным кодом 23Y34

К сожалению, процесс разделения информации на несколько отношений не всегда носит столь безобидный характер. Например, сравните исходное отношение с атрибутами EmplId, JobTitle и Dept, представленное на рис. 9.7, с его декомпозицией на два отношения, предложенное ниже. На первый взгляд кажется, что система из двух отношений содержит ту же информацию, что и система из одного отношения, но в действительности это не так. Например, рассмотрим задачу определения подразделения, в котором работает некоторый сотрудник. При использовании системы с одним отношением эта задача сводится к извлечению информации из атрибута Dept того кортежа отношения, у

которого значение атрибута Emplld совпадает с личным номером указанного сотрудника. Однако для системы из двух отношений эта информация далеко не всегда может быть определена однозначно. Всегда можно установить название занимаемой некоторым сотрудником должности, а также выяснить, в каких подразделениях имеется эта должность. Но это не означает, что будут получены сведения о том, где работает указанный сотрудник, поскольку занимаемая им должность может существовать сразу в нескольких подразделениях.

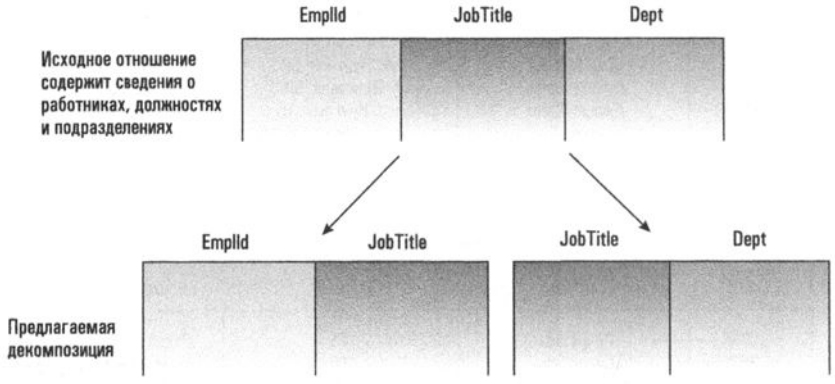


Рис. 9.7. Исходное отношение и его предлагаемая декомпозиция

Как видите, в одних случаях деление отношения на более мелкие отношения приводит к потере информации, а в других — нет (последний вариант называют **декомпозицией без потерь**). Такие реляционные характеристики — важный предмет рассмотрения при проектировании. Цель состоит в том, чтобы идентифицировать реляционные характеристики, способные привести к проблемам при проектировании базы данных, и найти такие способы реорганизации соответствующих отношений, которые позволят устранить эти проблемные характеристики.

Реляционные операции

Теперь, когда вы уже получили общее представление о том, как данные могут быть организованы в терминах реляционной модели, пришло время обсудить, как можно извлекать информацию из базы данных, представляющей собой совокупность отношений. Мы начнем наше обсуждение с рассмотрения нескольких операций над отношениями, которые в этом смысле могут оказаться полезными.

Иногда возникает потребность в простом извлечении кортежей из отношений. Для получения информации о сотруднике нам необходимо выбрать из отношения EMPLOYEE кортеж с соответствующим значением атрибута EmplID, а

для получения списка существующих в некотором подразделении должностей следует выбрать из отношения JOB все кортежи с соответствующим значением атрибута Dept. В результате выполнения подобных операций будет создано новое отношение (таблица), состоящее из отобранных кортежей исходного отношения. В случае выбора информации о сотруднике новое отношение будет содержать только один кортеж из отношения EMPLOYEE. При определении существующих в некотором подразделении должностей результирующее отношение, вероятно, будет содержать несколько кортежей из исходного отношения JOB.

Таким образом, одной из операций, которую нам может потребоваться выполнить над некоторым отношением, является выборка кортежей с определенными характеристиками и помещение этих кортежей в новое отношение. Для формального представления этой операции мы будем использовать следующий синтаксис:

```
NEW ← SELECT from EMPLOYEE where EmplId = '34Y70'
```

Семантика этого выражения — создать новое отношение с именем NEW, содержащее те кортежи (в приведенном случае он будет единственным) отношения EMPLOYEE, в которых значение атрибута Emplld равно "34Y70" (рис. 9.8).

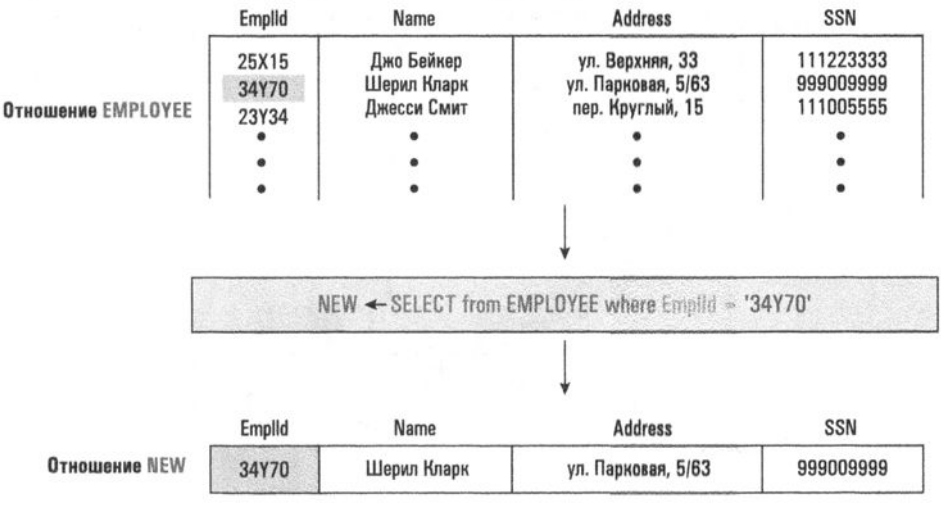


Рис. 9.8. Операция SELECT

В противоположность операции выборки SELECT, которая извлекает из отношения строки, операция проекции PROJECT предназначена для извлечения столбцов. Предположим, что при определении списка существующих в некотором подразделении должностей уже была выполнена операция SELECT, извлекающая из отношения JOB все кортежи, относящиеся к указанному подразделению. Выбранные кортежи были помещены в новое отношение NEW1.

Требуемый список наименований должностей содержится в столбце JobTitle нового отношения. Операция PROJECT позволяет извлечь этот столбец (или несколько столбцов) и поместить результат в новое отношение. Эту операцию можно представить следующим образом:

```
NEW2 ¬ PROJECT JobTitle from NEW1
```

В результате будет создано еще одно новое отношение (с именем NEW2), состоящее из единственного столбца значений, выбранных из столбца JobTitle отношения NEW1.

Ниже приведен еще один пример использования операции PROJECT:

```
MAIL ¬ PROJECT Name, Address from EMPLOYEE
```

Эта операция позволяет получить список имен и адресов всех сотрудников компании. Искомый список содержится во вновь созданном отношении с именем MAIL, с двумя атрибутами Name и Address (рис. 9.9).

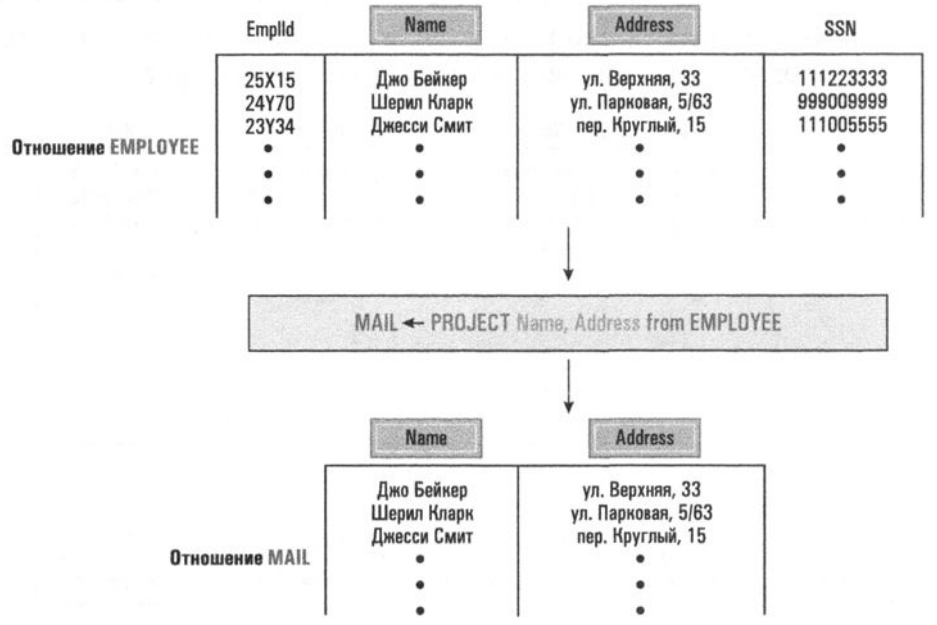


Рис. 9.9. Операция PROJECT

Другой операцией, также используемой в реляционных базах данных, является операция соединения JOIN. Она предназначена для объединения данных из двух разных отношений. Соединение двух отношений посредством операции JOIN приводит к созданию нового отношения, набор атрибутов которого включает все атрибуты исходных отношений (рис. 9.10). Имена атрибутов нового отношения не отличаются от имен атрибутов исходных отношений, за

исключением того, что перед каждым из них в качестве префикса, отделяемого точкой, указывается имя исходного отношения. (Если отношение A содержит атрибуты V и W, а отношение B — атрибуты X, Y и Z, то полученное в результате их соединения отношение будет содержать атрибуты A.V, A.W, B.X, B.Y и B.Z.) Такой принцип именования атрибутов гарантирует уникальность имен атрибутов нового отношения даже в том случае, когда исходные отношения содержат одноименные атрибуты.

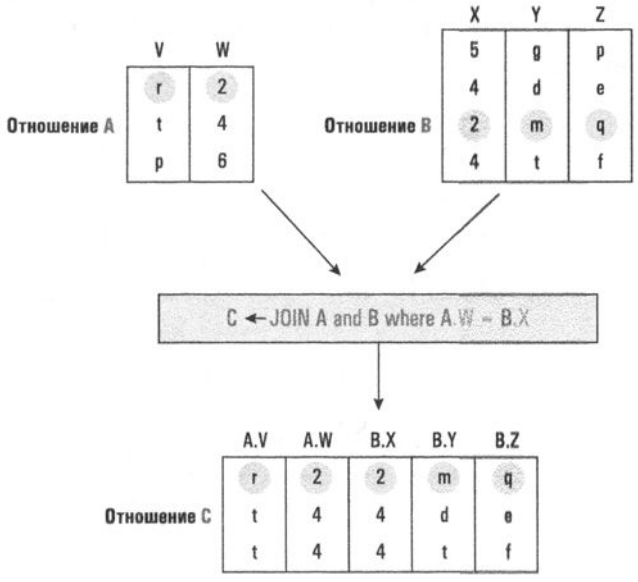


Рис. 9.10. Операция JOIN

Кортежи (строки) нового отношения образуются посредством конкатенации кортежей двух исходных отношений, как показано на рис. 9.10. Для того чтобы определить, какие именно строки из исходных отношений должны быть объединены, в операции JOIN задается некоторое условие. Одним из вариантов такого условия является задание пары атрибутов исходных отношений, значения в которых должны совпадать. Именно этот вариант условия представлен на рис. 9.10, который демонстрирует механизм выполнения следующего оператора:

```
C ← JOIN A and B where A.W = B.X
```

В данном примере кортеж из отношения A должен быть объединен с кортежем из отношения B только при точном совпадении в этих кортежах значений атрибутов W и X. Таким образом, конкатенация кортежа (r, 2) из отношения A и кортежа (2, m, q) из отношения B присутствует в результирующей таблице по причине равенства в них значений атрибута W из отношения A и атрибута X

из отношения B. И наоборот, в результирующем отношении отсутствует конкатенация кортежа (r, 2) из отношения A и кортежа (5, g, p) из отношения B, поскольку значения атрибутов W и X в этих кортежах различны.

Еще один пример приведен на рис. 9.11; на нем представлен результат выполнения следующего оператора:

```
C ¬ JOIN A and B where A.W < B.X
```

Обратите внимание, что в результирующее отношение помещены только те кортежи из отношения A, в которых значение атрибута W меньше значения атрибута X из отношения B.

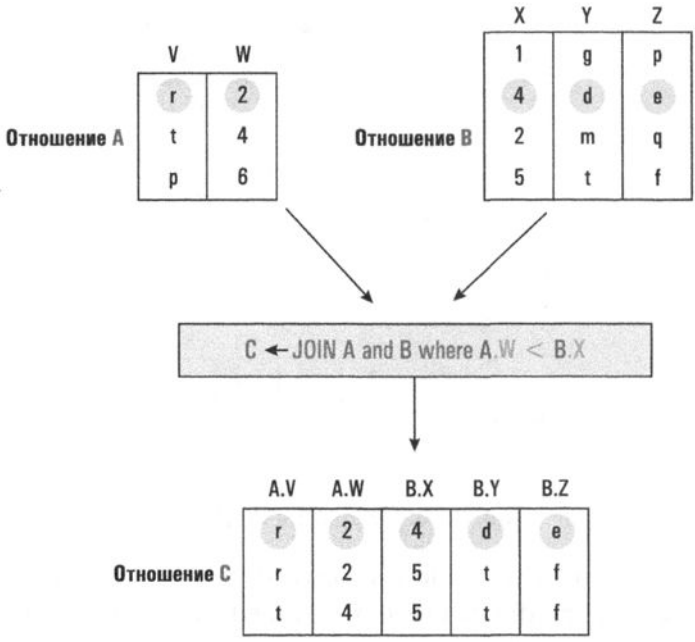


Рис. 9.11. Еще один пример выполнения операции JOIN

Теперь рассмотрим, как операция JOIN может быть использована для получения из представленной на рис. 9.5 базы данных списка личных номеров всех работников с указанием подразделений, в которых они работают в настоящее время. При первом же взгляде на эту базу данных становится ясно, что требуемая информация размещена более чем в одном отношении, поэтому для получения желаемого результата будет недостаточно одних только операций SELECT и PROJECT. В действительности в этом случае необходимо выполнить такую операцию:

```
NEW1 ¬ JOIN ASSIGNMENT and JOB
  where ASSIGNMENT.JobId = JOB.JobId
```

В результате будет создано отношение с именем NEW1, как показано на рис. 9.12. Далее с помощью операции SELECT из нового отношения извлекаются те кортежи, в которых атрибут ASSIGNMENT.TermDate имеет значение '*' (это признак того, что сотрудник занимает данную должность и в настоящее время). После этого к полученному результату применяется операция PROJECT для извлечения атрибутов ASSIGNMENT.Emplid и JOB.Dept. Ниже приведена последовательность всех операций, которые следует выполнить для извлечения требуемой информации из базы данных, представленной на рис. 9.5.

```
NEW1 ¬ JOIN ASSIGNMENT and JOB
  where ASSIGNMENT.JobId = JOB.JobId
NEW2 ¬ SELECT from NEW1 where ASSIGNMENT.TermDate = '*'
LIST ¬ PROJECT ASSIGNMENT.EmplId, JOB.Dept from NEW2
```

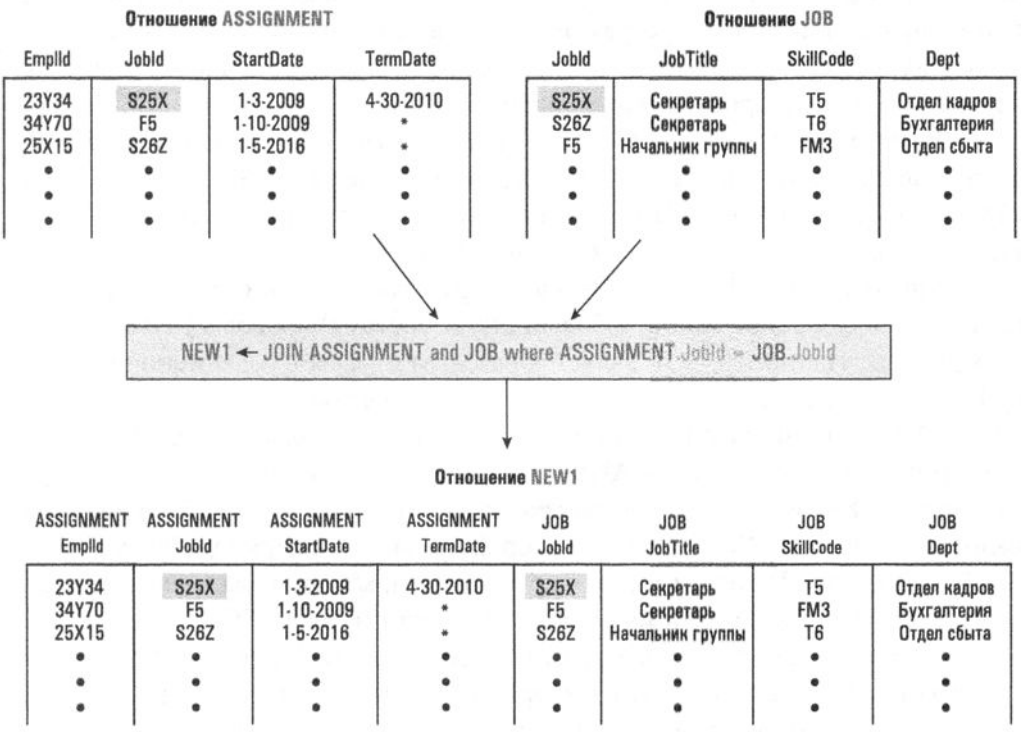


Рис. 9.12. Пример использования операции JOIN

Язык SQL

Теперь, ознакомившись с основными реляционными операциями, давайте вновь обратимся к общей структуре системы базы данных. Напомним, что в действительности физический способ хранения информации в базе данных

описывается в терминах систем массовой памяти. Чтобы оградить прикладного программиста от подобных забот, СУБД должна позволять ему создавать прикладное программное обеспечение в терминах принятой модели базы данных, подобной обсуждавшейся нами раньше. Поэтому реляционная СУБД должна принимать команды, сформулированные в терминах реляционной модели, а затем неявно преобразовывать их в последовательность действий, необходимых для доступа к реальным структурам хранения информации. Подобная задача обычно решается с помощью набора процедур, которые могут использоваться в прикладном программном обеспечении как абстрактные инструменты. Следовательно, СУБД, использующие реляционную модель, должны включать процедуры для выполнения реляционных операций `SELECT`, `PROJECT` и `JOIN`, которые могли бы вызываться из прикладного программного обеспечения с использованием синтаксических структур, совместимых с базовым языком. В этом случае прикладное программное обеспечение можно было бы разрабатывать так, как если бы данные действительно хранились в виде тех простых таблиц, которыми оперирует реляционная модель.

Современные системы управления реляционными базами данных не обязательно предоставляют процедуры для выполнения операций `SELECT`, `PROJECT` и `JOIN` в их простейшем виде. Вместо этого они предоставляют процедуры, которые могут быть комбинациями этих основных шагов. Примером является язык SQL (*Structured Query Language* — язык структурированных запросов), который представляет собой основу в большинстве систем запросов в реляционных базах данных. Например, SQL является базовым языком в реляционной СУБД MySQL, которая используется на множестве серверов баз данных в Интернете.

Одной из причин такой популярности языка SQL является издание стандарта этого языка институтом ANSI (Американский национальный институт стандартов). Другая причина состоит в том, что SQL был разработан и впервые реализован в фирме IBM, авторитет которой на рынке информационных технологий несомненен. В этом разделе мы рассмотрим, как запросы к реляционной базе данных могут быть сформулированы средствами языка SQL.

Хотя можно было бы ожидать, что запрос, сформулированный на языке SQL, будет напоминать команду в императивном стиле, реальность такова, что, по сути, это оператор объявления. Оператор на языке SQL следует воспринимать как описание той информации, которая вам необходима, а не как последовательность действий, которые нужно выполнить для ее получения. Смысл такого подхода состоит в том, что язык SQL имеет своей целью освобождение прикладных программистов от бремени разработки алгоритмов манипулирования отношениями: все, что нужно сделать, — это просто описать, какая именно информация им требуется.

В качестве первого примера оператора на языке SQL давайте соответствующим образом преобразуем наш последний запрос, в котором был представлен трехэтапный процесс получения всех идентификационных номеров сотрудников вместе с наименованиями отделов, в которых они работают. В языке SQL требуемый запрос может быть сформулирован с помощью единственного оператора:

```
SELECT EmplId, Dept
FROM Assignment, Job
WHERE Assignment.JobId = Job.JobId
  AND Assignment.TermDate = '*'
```

Как видно из приведенного примера, каждое SQL-выражение может содержать три отдельных предложения: *select* (выбрать), *from* (из) и *where* (где). Грубо говоря, подобный оператор представляет собой запрос на получение результатов операции JOIN для всех отношений, перечисленных в предложении *from*, с последующей выборкой из этого результата всех кортежей, которые удовлетворяют условию, заданному в предложении *where*. Выполнение запроса должно завершаться операцией PROJECT для всех атрибутов, перечисленных в предложении *select*. (Обратите внимание, что терминология, используемая в языке SQL, в некотором смысле противоположна употреблявшейся нами ранее, т.е. предложение *select* в SQL-выражении определяет атрибуты, которые должны использоваться в операции PROJECT.) Ниже рассмотрено несколько простых примеров.

С помощью этого SQL-оператора создается список имен и адресов всех сотрудников, сведения о которых присутствуют в отношении EMPLOYEE. Обратите внимание, что, в сущности, это всего лишь операция PROJECT.

```
SELECT Name, Address
FROM Employee
```

С помощью следующего SQL-оператора извлекается вся информация из кортежа отношения Employee, относящегося к работнику по имени Шерил Кларк. По существу, это просто операция SELECT.

```
SELECT EmplId, Name, Address, SSNum
FROM Employee
WHERE Name = 'Шерил Кларк'
```

С помощью очередного SQL-оператора из таблицы Employee извлекаются только имя и адрес работника по имени Шерил Кларк. Это пример комбинации операций SELECT и PROJECT.

```
SELECT Name, Address
FROM Employee
WHERE Name = 'Шерил Кларк'
```

Наконец, данный SQL-оператор позволяет извлечь список имен и дат поступления на работу всех сотрудников организации. Отметим, что фактически это результат применения операции JOIN к отношениям Employee и Assignment с последующим выполнением операций SELECT для кортежей, определенных в предложении *where*, и операции PROJECT для атрибутов, определенных в предложении *select*.

```
SELECT Employee.Name, Assignment.StartDate
FROM Employee, Assignment
WHERE Employee.EmplId = Assignment.EmplId
```

В завершение следует сказать, что, помимо операторов для выполнения запросов, язык SQL располагает также операторами для определения структуры отношений, создания отношений и изменения содержимого отношений. Ниже приведены примеры применения операторов INSERT INTO (Вставить в), DELETE FROM (Удалить из) и UPDATE (Обновить).

9.2. Вопросы и упражнения

1. На основании информации из отношений Employee, Job и Assignment (см. рис. 9.5) дайте ответы на следующие вопросы.
   - **а.** Кто из секретарей, работающих в бухгалтерии, имеет опыт работы в отделе кадров?
   - **б.** Кто занимает должность начальника группы в отделе сбыта?
   - **в.** Какую должность занимает сейчас Джерри Смит?
2. Используя информацию из отношений Employee, Job и Assignment (см. рис. 9.5), напишите последовательность реляционных операций для получения списка всех должностей, имеющихся в отделе кадров.
3. На основании информации из отношений Employee, Job и Assignment (см. рис. 9.5) напишите последовательность реляционных операций для получения списка имен сотрудников с указанием подразделений, в которых они работают.
4. Преобразуйте ваши ответы на вопросы 2 и 3 в форму операторов языка SQL.
5. Как в реляционной модели обеспечивается независимость данных?
6. Каким образом различные отношения связываются между собой в реляционной модели данных?

С помощью этого SQL-оператора в отношение Employee помещается новый кортеж с указанными значениями атрибутов.

```
INSERT INTO Employee
VALUES ('42Z12', 'Сью Барт', 'ул. Южная, 33','444661111')
```

Далее с помощью приведенного ниже SQL-оператора из таблицы Employee удаляется кортеж, содержащий информацию о работнике Джерри Смите.

```
DELETE FROM Employee
WHERE Name = 'Джерри Смит'
```

И наконец с помощью данного оператора изменяется значение атрибута Address того кортежа таблицы Employee, который содержит информацию о работнике Джо Бейкере.

```
UPDATE Employee
SET Address = 'ул. Северная, 18'
WHERE Name = 'Джо Бейкер'
```

9.3. Объектно-ориентированные базы данных

Другая модель базы данных основана на объектно-ориентированной парадигме. Этот подход приводит к объектно-ориентированной базе данных, состоящей из объектов, которые связаны между собой для отражения их отношений. Например, объектно-ориентированная реализация базы данных сотрудников из предыдущего раздела может состоять из трех классов (типов объектов): Employee, Job и Assignment. Объект из класса Employee может содержать такие компоненты, как EmplId, Name, Address и SSNum, а объект из класса Job может содержать такие компоненты, как JobId, JobTitle, SkillCode и Dept. Наконец, каждый объект из класса Assignment может содержать такие компоненты, как StartDate и TermDate.

Концептуальное представление такой базы данных представлено в виде схемы на рис. 9.13, на которой для отображения связей, существующих между различными объектами, используются соединяющие их линии. Взглянув на объект класса Employee, можно заметить, что он связан с коллекцией, состоящей из нескольких объектов класса Assignment, представляющих различные назначения, которые получал конкретный сотрудник. В свою очередь, каждый из объектов класса Assignment в этой коллекции связан с некоторым объектом класса Job, описывающим должность, которую занимал данный работник. Таким образом, все назначения сотрудника могут быть прослежены по ссылкам, присутствующим в объекте, представляющем этого сотрудника. Аналогичным образом по ссылкам, присутствующим в объекте и представляющим некоторую должность, можно установить всех сотрудников, когда-либо занимавших ее.

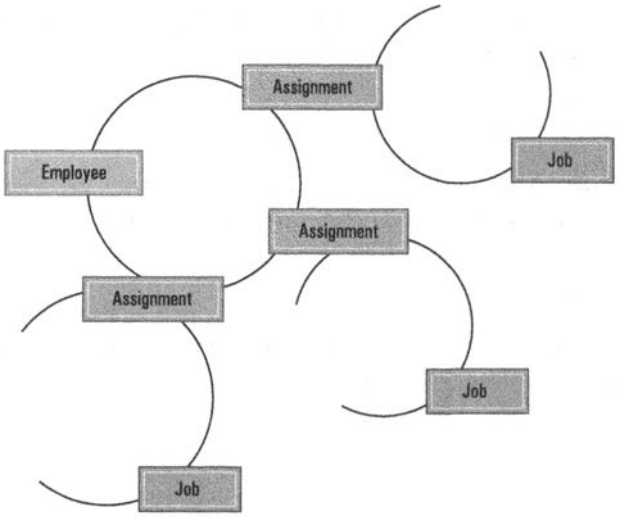


Рис. 9.13. Связи между объектами в объектно-ориентированной базе данных

В объектно-ориентированной базе данных связи между объектами, как правило, поддерживаются средствами СУБД, поэтому детали реализации этого процесса не должны занимать прикладного программиста. В прикладной программе просто указывается, с какими именно объектами должны быть установлены связи при создании нового объекта некоторого класса. Далее СУБД сама создает такую систему указателей, которая необходима для представления заданных связей. Например, объекты, представляющие последовательность назначений некоторого сотрудника на различные должности, могут быть реализованы в СУБД посредством связанного списка.

Другая задача, которую должна решать объектно-ориентированная СУБД, состоит в обеспечении постоянного хранения вверенных ей объектов. Это требование кажется совершенно естественным, однако на самом деле оно существенно отличается от обычных методов работы с объектами. Дело в том, что объекты, созданные в процессе выполнения объектно-ориентированной программы, обычно разрушаются при ее завершении. В этом смысле объекты в ООП рассматриваются как временные конструкции. Однако объекты, созданные и помещенные в базу данных в процессе выполнения объектно-ориентированной программы, должны быть сохранены в ней и после завершения программы. Такие объекты называются **постоянными**. Отсюда можно сделать вывод, что требование постоянного хранения объектов является существенным отклонением от норм, принятых в ООП.

Сторонники объектно-ориентированных баз данных приводят многочисленные аргументы с целью показать, почему объектно-ориентированный подход

к проектированию баз данных лучше, чем реляционный подход. Один из них заключается в том, что объектно-ориентированный подход позволяет всю программную систему (прикладное программное обеспечение, СУБД и саму базу данных) проектировать в одной и той же парадигме. Это противоречит исторически распространенной практике использования императивного языка программирования для разработки прикладного программного обеспечения, взаимодействующего с реляционной базой данных. Такому подходу неизбежно присущ конфликт между императивной и реляционной парадигмами. Эти различия являются достаточно тонкими для нашего уровня изложения материала, однако они служили источником множества ошибок в программном обеспечении на протяжении многих лет. Даже на своем уровне мы можем оценить, что объектно-ориентированная база данных в сочетании с объектно-ориентированной прикладной программой создает однородную картину из объектов, взаимодействующих между собой по всей системе. С другой стороны, реляционная база данных в сочетании с императивной прикладной программой вызывает образ двух разных по своей сути структур, пытающихся отыскать общий подход к взаимодействию.

Чтобы оценить еще одно преимущество, которое объектно-ориентированные СУБД имеют в сравнении с реляционными аналогами, рассмотрим проблему хранения имен сотрудников в реляционной базе данных. Если полное имя хранится в отношении как один атрибут, то создание запросов на извлечение только его фамилии существенно усложняется. Если же полное имя хранится в виде отдельных атрибутов, таких как имя, отчество и фамилия, то количество атрибутов становится проблематичным, поскольку не все имена соответствуют определенной структуре — даже если речь идет о группе людей, принадлежащих одной культуре. В объектно-ориентированной базе данных эти проблемы могут быть скрыты в объекте, содержащем все сведения о сотруднике. Имя сотрудника может храниться как интеллектуальный объект, способный сообщать имя сотрудника в различных форматах. Таким образом, вне этих объектов будет так же просто иметь дело только с фамилиями, как с полными именами, девичьими фамилиями или псевдонимами. Детали реализации, связанные с каждым из подобных вариантов, будут полностью скрыты в самом объекте.

Подобная возможность инкапсулировать особенности представления различных форматов данных выгодна и в других случаях. В реляционной базе данных атрибуты отношения являются частью общей структуры базы данных, поэтому типы данных, связанные с этими атрибутами, распространяются на всю базу данных в целом. (Переменные для временного хранения должны быть объявлены как соответствующий тип, а также необходимо разработать функции для манипулирования данными различных типов.) В результате расширение реляционной базы данных для включения атрибутов новых типов (например, аудио и видео) может оказаться проблематичным. В частности, может

потребоваться расширить ряд функций в рамках всей структуры базы данных, чтобы включить поддержку этих новых типов данных. Что же касается объектно-ориентированной среды, то те же функции, которые использовались для извлечения объекта, представляющего имя сотрудника, могут использоваться и для извлечения объекта, представляющего движущееся изображение, поскольку различия, связанные с типом данных, могут быть скрыты непосредственно в самих объектах. Таким образом, объектно-ориентированный подход представляется более совместимым с созданием мультимедийных баз данных — возможность, которую можно расценивать как большое преимущество.

Еще одним преимуществом, которое обеспечивает объектно-ориентированная парадигма при проектировании баз данных, является возможность хранения *интеллектуальных объектов*, а не просто данных. Иначе говоря, объект может содержать методы, описывающие, как он должен реагировать на поступающие сообщения в отношении его содержимого и связей с другими объектами. Например, на рис. 9.13 каждый объект класса `Employee` может иметь методы извлечения и обновления информации о сотруднике, метод извлечения информации о послужном списке сотрудника и, возможно, метод изменения должности, занимаемой сотрудником. Аналогичным образом каждый объект из класса `JOB` может включать метод для извлечения описания должности и, возможно, метод для выдачи списка всех сотрудников, занимавших данную должность. В результате для получения послужного списка сотрудника нам уже не нужно будет писать собственную сложную процедуру. Вместо этого можно будет просто попросить соответствующий объект сотрудника сообщить историю его работы. Эта способность создавать базы данных, компоненты которых разумно реагируют на запросы, предлагает широкий спектр возможностей, выходящих за рамки традиционных реляционных баз данных.

9.3. Вопросы и упражнения

1. Какие методы должен содержать каждый экземпляр объекта класса `Assignment` в обсуждавшейся в этом разделе базе данных сотрудников?
2. Что такое постоянный объект?
3. Определите несколько классов, которые могли бы использоваться в объектно-ориентированной базе данных складского учета, и укажите некоторые из их необходимых внутренних характеристик.
4. Назовите преимущества объектно-ориентированных баз данных в сравнении с реляционными.

9.4. Обеспечение целостности баз данных

Недорогие СУБД для индивидуального использования являются относительно простыми и создаются с единственной целью — оградить пользователя от технических подробностей реализации базы данных. Поддерживаемые подобными системами базы данных относительно невелики и обычно содержат информацию, потеря которой скорее доставит неудобства, нежели приведет к катастрофическим последствиям. При возникновении проблемы пользователь, как правило, может либо непосредственно внести исправления в ошибочные данные, либо восстановить базу данных из резервной копии, а затем вручную внести все изменения, имевшие место с момента создания этой копии. Подобный процесс, безусловно, неудобен, но стоимость модернизации программного обеспечения значительно превышает неудобства. К тому же подобные неудобства касаются лишь нескольких человек, а финансовые потери от возникших проблем чаще всего весьма ограничены.

В случае больших многопользовательских коммерческих баз данных уровень риска значительно выше. Цена, которую придется платить за некорректные или утраченные данные, может оказаться огромной, а последствия — катастрофическими. Кроме того, поскольку базы данных постоянно растут в отношении количества их одновременно работающих пользователей, допустимые операции над базой данных должны позволять соответствующее масштабирование. В подобной ситуации важнейшая роль СУБД заключается в обеспечении целостности данных, которая достигается в результате предупреждения проблем, возможных при частичном завершении операций, а также посредством исключения возможности взаимного влияния независимо выполняемых операций, что может привести к нарушению достоверности хранящейся в базе данных информации. Именно эту функцию СУБД мы обсудим в этом разделе.



Основные положения для запоминания

- **Масштабируемость систем является важным показателем, когда наборы данных достигают больших размеров.**

Протокол фиксации/отката изменений

Единичная транзакция, примерами которой могут служить перевод денежных средств с одного банковского счета на другой, отмена резервирования авиабилета или регистрация студента как слушателя университетского курса, на уровне базы данных может выполняться в несколько шагов. Например, перевод

денежных средств между банковскими счетами требует уменьшения остатка на одном счете и увеличения на другом. Между этими двумя шагами информация в базе данных оказывается несогласованной. Действительно, на протяжении короткого периода времени, между уменьшением остатка на одном счете и увеличением на другом, снятая сумма денег просто отсутствует. Подобным же образом при переопределении места пассажира некоторого авиарейса какое-то время у пассажира вообще нет места, т.е. в списке пассажиров на одного человека больше, чем в действительности имеется распределенных мест.

При использовании больших баз данных, подвергающихся интенсивной обработке со множеством выполняемых транзакций, весьма вероятно, что в любой произвольно выбранный момент несколько транзакций в базе данных будут находиться в процессе выполнения. Это означает, что запрос на выполнение новой транзакции или сбой оборудования, как правило, происходят в тот момент, когда база данных находится в противоречивом состоянии.

Рассмотрим случай возникновения сбоя. Задача СУБД заключается в том, чтобы при его возникновении предотвратить переход системы в противоречивое состояние. Чаще всего эта задача решается с помощью ведения журнала, в котором фиксируются сведения о каждом выполненном шаге транзакции, завершившемся физической записью информации на постоянное устройство хранения, такое как жесткий диск. Прежде чем транзакции будет позволено внести изменения в базу данных, характер предполагаемых изменений заносится в журнал в виде соответствующей записи. Таким образом, журнал содержит сведения о каждом действии выполняемой транзакции.

Момент, когда все шаги выполняемой транзакции оказываются записанными в журнал, называется **точкой фиксации**. Именно с этого момента СУБД получает в свое распоряжение всю информацию, необходимую для восстановления результатов выполнения транзакции, если это понадобится. И только с этого момента СУБД может ответственно гарантировать, что все выполненные транзакцией действия действительно отражены в базе данных. При сбое в работе оборудования СУБД сможет использовать находящуюся в журнале информацию для восстановления результатов выполнения всех транзакций, имевших место с момента последнего резервного копирования базы данных.

Если проблема возникла до достижения точки фиксации, СУБД легко сможет обнаружить эту частично выполненную транзакцию. В этом случае данные журнала могут быть использованы СУБД для **отката** (отмены) всех выполненных в транзакции действий. Например, при аппаратном сбое СУБД сможет вернуть базу данных в согласованное состояние за счет отката результатов выполнения всех транзакций, которые не достигли точки фиксации к моменту сбоя.

Применение процедуры отката транзакций не ограничивается лишь процессом восстановления системы после сбоев. Очень часто откат транзакции

является элементом нормальной работы СУБД. Например, транзакция может быть прервана до ее завершения из-за обнаружения системой попытки несанкционированного доступа к защищенной информации. Откат транзакции может потребоваться и для вывода системы из состояния взаимной блокировки, когда каждая из пары конкурирующих за доступ к одним и тем же данным транзакций находится в состоянии бесконечного ожидания получения доступа к данным, используемым другой транзакцией этой пары. При обнаружении подобной ситуации СУБД использует данные журнала для отката транзакции и предотвращения перехода базы данных в противоречивое состояние вследствие незавершенности транзакций.

Чтобы подчеркнуть сложность задач, решаемых при разработке СУБД, следует отметить, что существуют некоторые не вполне очевидные проблемы, связанные с процедурой отката транзакции. Отмена одной транзакции может затрагивать данные, которые уже были использованы другой транзакцией. Например, возвращаемая транзакция может восстановить прежнее значение баланса некоторого счета, в то время как другая уже использовала новое значение. Это означает, что эти дополнительные транзакции также должны быть отменены, что, в свою очередь, может потребовать отмены других транзакций. Данная проблема получила название **каскадного отката** транзакций.

Механизм блокировок

Теперь давайте рассмотрим ситуацию, при которой транзакция начинается в тот момент, когда база данных находится в промежуточном состоянии вследствие выполнения другой транзакции. Очевидно, что подобная ситуация может привести к ошибочным результатам, возникающим из-за неправильного взаимодействия транзакций. Например, может иметь место так называемая **проблема недостоверных итогов**, которая возникает, когда одна транзакция выполняет перевод средств с одного счета на другой, а другая подсчитывает общую сумму всех банковских вкладов. В зависимости от порядка выполнения отдельных шагов транзакции перевода, результат суммирования может оказаться меньше или больше действительного значения. Другой тип возможных ошибок связан с **проблемой потерянного обновления**, когда две транзакции одновременно обновляют текущее значение одного и того же счета. Если одна транзакция считывает текущее значение остатка на счете в тот момент, когда другая уже его прочла, но еще не изменила, то обе транзакции будут выполнять обновление значения на счете, исходя из одного и того же начального значения. В результате одно значение остатка будет перекрыто значением, вычисленным другой транзакцией.

Для разрешения подобных проблем СУБД должна обеспечить режим поочередного выполнения транзакций во всей его полноте посредством помещения

каждой из них в очередь до тех пор, пока предыдущая транзакция не будет полностью завершена. Однако транзакции часто долго ожидают завершения операций ввода-вывода в массовой памяти. Чередуя выполнение этапов различных транзакций, можно организовать работу так, что время ожидания доступа к данным в одной транзакции может быть использовано для обработки данных в другой транзакции, уже получившей всю необходимую информацию. Поэтому более мощные СУБД для координации разделения времени между транзакциями используют специальную программу-планировщик, функционирующую во многом аналогично планировщику задач в многозадачных операционных системах (см. раздел 3.3).

Во избежание аномалий, подобных проблеме недостоверных итогов или проблеме потерянного обновления, планировщик использует механизм, называемый **протоколом блокировки**, который требует, чтобы каждый используемый транзакцией элемент базы данных помечался специальным образом. Эти метки называются блокировками, а отмеченные подобным образом элементы базы данных являются заблокированными. Существуют два типа блокировок — **разделяемая** и **эксклюзивная**, которые соответствуют, в свою очередь, двум типам доступа к данным, необходимым транзакциям, — совместному и эксклюзивному. Если в процессе выполнения транзакции используемый ею элемент данных изменяться не будет, то доступ к нему может быть разделяемым. Это означает, что другим транзакциям также будет разрешено считывать этот элемент данных. Однако если транзакция намеревается вносить изменения в элемент данных, то ей необходимо получить к нему эксклюзивный доступ, означающий, что никакая другая транзакция не сможет в этот момент получить доступ к этому элементу данных.

В соответствии с протоколом блокировки любая транзакция при каждом запросе на получение доступа к некоторому элементу данных должна указывать СУБД тип необходимого ей доступа. Если транзакция запрашивает разделяемый доступ к элементу данных, который в данный момент не заблокирован или заблокирован разделяемой блокировкой, то требуемый доступ предоставляется и этот элемент данных отмечается как заблокированный разделяемой блокировкой. Однако если требуемый элемент данных помечен как заблокированный эксклюзивной блокировкой, то разделяемый доступ к данным не предоставляется. Если транзакция требует получения эксклюзивного доступа к данным, то он может быть предоставлен только в том случае, если требуемый элемент данных не отмечен как заблокированный. В результате транзакция, намеревающаяся изменить некоторый элемент данных, сможет предотвратить доступ к нему со стороны других транзакций посредством получения эксклюзивного доступа к этому элементу. В то же время несколько транзакций смогут получить совместный доступ к данным, если ни одна из них не намеревается вносить в них

изменения. Безусловно, как только транзакция завершит работу с некоторым элементом данных, она должна будет уведомить об этом СУБД, и блокировка с этого элемента данных будет снята.

Для обработки ситуации отказа транзакции в доступе к требуемому ей элементу данных используются самые разнообразные алгоритмы. В соответствии с одним из них выполнение транзакции приостанавливается до тех пор, пока требуемые данные не станут доступными. Однако такой подход способен привести к ситуации взаимной блокировки, когда две транзакции, требующие эксклюзивного доступа к одной паре элементов данных, блокируют выполнение друг друга за счет того, что одна транзакция блокирует первый элемент данных и ожидает, пока другая освободит второй, уже заблокированный этой транзакцией элемент данных. Во избежание подобных ситуаций некоторые СУБД предоставляют приоритет тем транзакциям, которые начали свое выполнение раньше. Если более ранней транзакции потребуется доступ к данным, заблокированным более поздней транзакцией, выполнение последней прекратится, все заблокированные ею элементы данных разблокируются и для этой транзакции будет выполнен принудительный откат (с использованием данных журнала). В результате более ранняя транзакция получает доступ к необходимым ей данным, а отмененная вновь запускается с начальной точки. Если более поздняя транзакция будет перезапускаться несколько раз подряд, то в итоге она непременно окажется более ранней в сравнении с другими выполняющимися транзакциями. Этот протокол, называемый **протоколом принудительной отмены-ожидания** (более ранние транзакции принудительно отменяют более поздние, а более поздние просто дожидаются окончания более ранних), гарантирует, что каждая транзакция, в конце концов, сможет завершить свою работу.

9.4. Вопросы и упражнения

1. В чем состоит различие между транзакцией, достигшей точки фиксации, и транзакцией, не достигшей этой точки?
2. Каким образом СУБД может предупреждать возникновение обширных каскадных откатов транзакций?
3. Покажите, каким образом неконтролируемое чередование этапов двух одновременно выполняемых транзакций, одна из которых уменьшает значение остатка на счете на 100 долларов, а другая — на 200 долларов, может привести к конечному значению 100, 200 или 300 долларов, если принять исходное значение остатка равным 400 долларам?

4. Предположим, что в СУБД используется механизм блокировок.
   а. Перечислите возможные варианты ответов на запрос транзакции о получении разделяемого доступа к элементу данных в базе.
   б. Перечислите возможные варианты ответов на запрос транзакции о получении эксклюзивного доступа к элементу данных в базе.
5. Опишите последовательность событий, которая может привести к возникновению ситуации взаимной блокировки между двумя транзакциями, выполняющимися в базе данных.
6. Предложите способ предотвращения возникновения той ситуации взаимной блокировки, которая была описана вами в ответе на предыдущий вопрос. Требует ли предложенное вами решение использования журнала СУБД? Поясните ваш ответ.

9.5. Традиционные файловые структуры

В этом разделе мы отступим от нашего исследования многомерных систем баз данных, чтобы рассмотреть традиционные файловые структуры. Эти структуры представляют собой историческое начало систем хранения и поиска данных, из которых развились современные технологии баз данных. Многие из методов, разработанных для этих структур (таких, как индексирование и хеширование), являются важными инструментами при построении современных объемных и сложных баз данных.

Последовательные файлы

Последовательным файлом является такой файл, доступ к которому осуществляется строго последовательно, от начала до конца, как будто информация в нем представляет собой одну длинную строку. Примерами могут служить аудиофайлы, видеофайлы или файлы, содержащие программы либо текстовые документы. Фактически большинство файлов, создаваемых обычным пользователем персонального компьютера, являются последовательными. Например, когда электронная таблица сохраняется на некотором носителе, ее информация кодируется и сохраняется в виде последовательного файла, из которого прикладное программное обеспечение электронной таблицы впоследствии сможет восстановить эту электронную таблицу в прежнем виде.

Текстовые файлы, являющиеся последовательными файлами, в которых каждая логическая запись представляет собой отдельный символ, закодированный с использованием кодировок ASCII или Unicode, часто используются в качестве

базовой структуры при создании более сложных последовательных файлов, таких, например, как файл записей о сотрудниках. Нужно лишь принять единый формат представления информации о каждом сотруднике в виде строки текста, а затем кодировать информацию в соответствии с этим форматом и записывать полученные в результате записи о сотрудниках одну за другой в виде одной непрерывной строки текста. Например, можно создать простой файл о сотрудниках, приняв соглашение строить запись о каждом сотруднике в виде строки из 31 символа, которая состоит из поля длиной 25 символов, содержащего имя сотрудника (с добавлением при необходимости достаточного количества пробелов до общей длины в 25 символов), за которым следует поле из 6 символов, содержащее идентификационный номер сотрудника. Конечный файл будет представлять собой длинную строку кодов символов, в которой каждый блок из 31-го символа будет представлять информацию об одном сотруднике, как показано на рис. 9.14. Информацию из такого файла нужно будет извлекать в виде логических записей, содержащихся в отдельных блоках длиной 31 символ. В пределах каждого из этих блоков отдельные поля будут идентифицироваться в соответствии с принятым форматом, согласно которому эти блоки были построены.

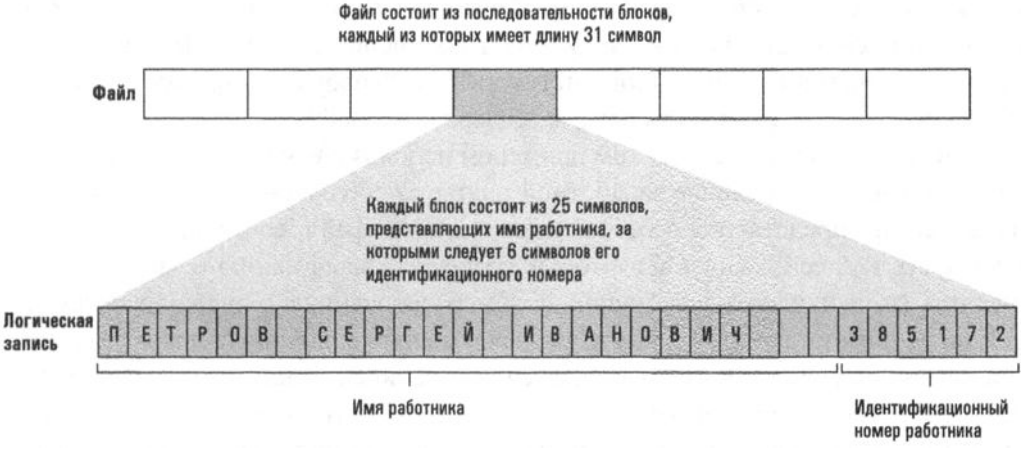


Рис. 9.14. Структура простого файла записей о сотрудниках, реализованного в виде текстового файла

Данные в последовательном файле должны быть записаны в массовом хранилище таким образом, чтобы гарантировать сохранение последовательного характера файла. Если система запоминающих устройств сама по себе последовательная (как в случае магнитной ленты или компакт-диска), это совсем несложно. Достаточно будет просто записать файл на носитель в полном соответствии с последовательными свойствами этого носителя. Тогда обработка данного файла будет заключаться в простом чтении и обработке его содержимого в том порядке, в котором оно находится на носителе. Это именно тот процесс,

который используется при воспроизведении аудиокомпакт-дисков, на которых музыка сохраняется в виде последовательного файла записанных на носитель секторов, — сектор за сектором вдоль одной непрерывной спиральной дорожки.

Однако если для записи используется дисковая память, то содержимое файла будет распределено по различным секторам, которые, в принципе, можно считывать в различном порядке. Чтобы сохранить требуемый порядок, большинство операционных систем (точнее, программа управления файлами) создает и поддерживает список секторов, в которых хранится каждый файл. Данный список сохраняется как часть системы каталогов диска на том же дисковом устройстве, где и сам файл. С помощью этого списка операционная система может считывать секторы в требуемом порядке, что позволяет писать прикладные программы так, как будто данные файла хранятся последовательно, хотя в действительности они разбросаны по разным участкам диска.

При обработке последовательного файла необходимо контролировать достижение конца этого файла. В общем случае конец последовательного файла принято обозначать как **EOF** (*End Of File*). Существует множество способов определения EOF. Один из них состоит в размещении в конце файла специальной записи, называемой **меткой конца файла** (*sentinel*). Другой способ — для определения ситуации достижения конца файла использовать информацию из системы каталогов операционной системы. Иначе говоря, поскольку операционная система знает, в каких секторах содержится файл, она также знает, где файл заканчивается. Классическим примером использования последовательных файлов является начисление заработной платы в небольшой компании. В этом случае можно представить себе последовательный файл, состоящий из серии логических записей, каждая из которых содержит информацию о зарплате сотрудника (имя и идентификационный номер сотрудника, тарифную сетку и т.д.), на основании которой должны распечатываться платежные документы по установленному образцу. По мере извлечения записей отдельных сотрудников рассчитывается их заработная плата и печатается соответствующий документ. Процедуру выполнения такой обработки можно проиллюстрировать следующим оператором псевдокода.

```
while (конец файла не достигнут):
  считать следующую запись файла и обработать ее
```

Когда логические записи в последовательном файле идентифицируются значениями ключевых полей, файл обычно упорядочивается таким образом, что записи появляются в порядке, определяемом ключами (возможно, в алфавитном или числовом формате). Такое расположение упрощает задачу обработки информации в файле. Например, предположим, что при начислении заработной платы требуется, чтобы для каждого сотрудника зарплата начислялась с учетом действительно отработанного им времени. Если оба файла — содержащий

записи табеля рабочего времени и содержащий записи с информацией о сотрудниках — будут одинаково упорядочены по одним и тем же ключевым полям, то процесс начисления может быть осуществлен с использованием механизма последовательного доступа одновременно к обоим файлам — с использованием в расчетах табельной информации, полученной из одного файла, и всей остальной информации, полученной из соответствующей записи другого файла. Это решение существенно лучше по сравнению с многократным повторным выполнением поиска, что потребовалось бы, если бы файлы не были соответствующим образом упорядочены. В результате обновление классических последовательных файлов обычно выполняется в несколько этапов. Во-первых, новая информация (например, данные табельного учета) записывается в последовательный файл, называемый файлом изменений, и этот файл сортируется так, чтобы информация в нем была упорядочена так же, как и в обновляемом файле, который в этом случае называется основным. Затем записи основного файла обновляются в процессе последовательного считывания записей обоих файлов.

Другим классическим примером последовательной обработки является слияние двух файлов для создания нового файла, содержащего записи двух исходных. Предполагается, что записи входных файлов организованы в порядке возрастания значений общего поля ключа и что выходной файл должен обладать тем же свойством. Классический алгоритм слияния файлов представлен на рис. 9.15. В этом случае задача заключается в построении выходного файла по мере последовательного сканирования двух входных файлов, как показано на рис. 9.16.

```
def MergeFiles (InputFileA, InputFileB, OutputFile):
  if (В обоих входных файлах достигнут EOF):
    Stop, причем OutputFile пуст
  if (Для InputFileA EOF не достигнут):
    Объявить его первую запись текущей записью этого файла
  if (Для InputFileB EOF не достигнут):
    Объявить его первую запись текущей записью этого файла
  while (EOF не достигнут ни в одном из входных файлов):
    Поместить текущую запись с "меньшим" значением поля
      ключа в файл OutputFile
    if (Эта текущая запись является последней
        в соответствующем входном файле):
        Считать этот входной файл достигшим EOF
    else:
      Объявить следующую запись этого входного файла
        его текущей записью
  Начиная с текущей записи файла, не достигшего EOF, копировать
    его оставшиеся записи в файл OutputFile
```

Рис. 9.15. Процедура слияния двух файлов с последовательным доступом

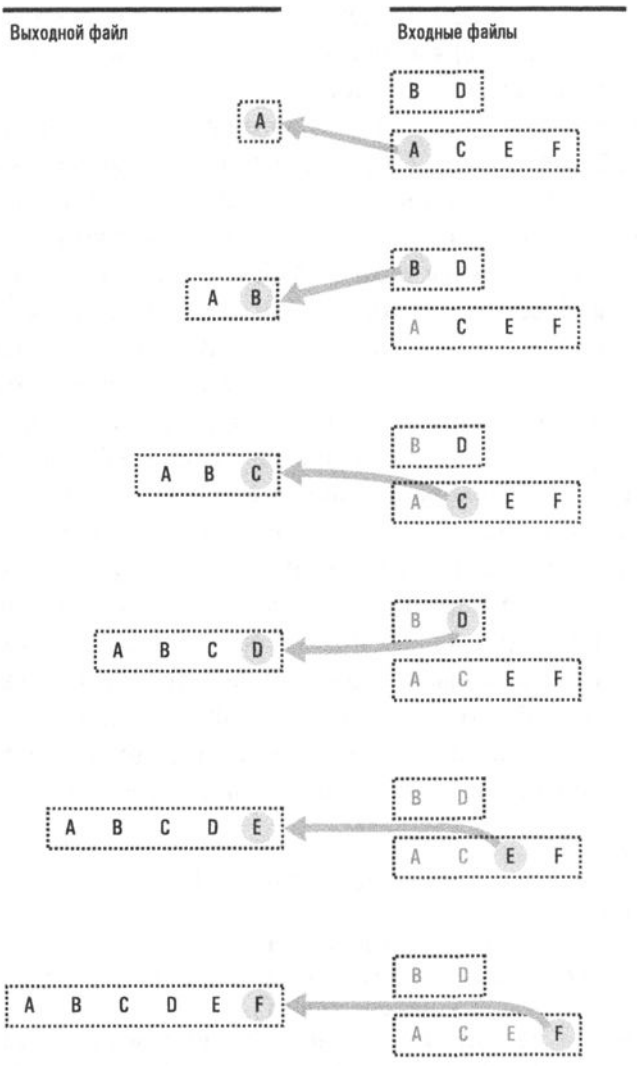


Рис. 9.16. Применение алгоритма слияния (буквы использованы для представления содержимого записей, каждая буква представляет собой значение поля ключа соответствующей записи)

Индексированные файлы

Последовательные файлы идеально подходят для хранения данных, которые будут обрабатываться в том порядке, в котором хранятся записи в этих файлах. Однако такие файлы неэффективны, когда записи в файле должны извлекаться в непредсказуемом порядке. В таких ситуациях необходим способ быстро

определить местоположение нужной логической записи. Популярное решение состоит в том, чтобы создать для файла *индекс*, — почти так же, как в книгах предметный указатель используется для поиска интересующих тем. Такая файловая система называется **индексированным файлом**.

Индекс файла содержит список значений поля ключа для всех записей файла вместе с информацией о том, где хранится запись, содержащая данный ключ. Таким образом, чтобы найти конкретную запись, достаточно найти нужный ключ в индексе, а затем извлечь блок информации, хранящейся в местоположении, связанном в индексе с этим ключом.

Индекс файла обычно хранится как отдельный файл на том же устройстве хранения данных, где и индексированный файл. Индекс обычно считывается в основную память до начала обработки самого файла, поэтому он будет легко доступен, когда потребуется получить доступ к той или иной записи в файле (рис. 9.17).



Рис. 9.17. Открытие индексированного файла

Классическим примером задачи, требующей использования индексированного файла, является ведение записей о служащих. В этом случае индекс можно использовать во избежание продолжительных последовательных просмотров с целью извлечения отдельных записей. В частности, если файл записей о служащих будет индексирован по их идентификационным номерам, то можно быстро считать запись любого служащего, зная его идентификационный номер. Другим примером может служить аудиокомпакт-диск, на котором индекс используется для осуществления быстрого доступа к отдельным музыкальным записям.

За прошедшие годы были использованы многочисленные вариации базовой концепции индексированного файла. В одном из вариантов индекс строится иерархическим образом, в результате чего он приобретает многоуровневую или древовидную структуру. Ярким примером такого подхода является

иерархическая система каталогов, используемая большинством операционных систем для организации хранения файлов. В этом случае каталоги или папки играют роль индексов, каждый из которых содержит ссылки на свои подкаталоги. С этой точки зрения вся файловая система — это всего лишь один большой индексированный файл.

Хешированные файлы

Хотя индексация и обеспечивает прямой доступ к записям файла, это делается за счет организации и ведения индекса. **Хеширование** — это метод, обеспечивающий прямой доступ к записям без использования каких-либо дополнительных структур. Как и в случае индексированных файлов, хеширование позволяет определить местоположение записи в файле по значению ее ключевого поля. Но вместо того, чтобы искать ключ в индексе, метод хеширования обеспечивает идентификацию местоположения записи непосредственно по значению ее ключа.

Процесс можно кратко описать следующим образом. Пространство, где хранится файл, делится на несколько секторов, каждый из которых называется **сегментом** (bucket), — это участок памяти, адресуемый как единое целое,

Хеширование как метод аутентификации

Хеширование — это нечто большее, чем просто метод построения эффективных систем хранения данных. Например, хеширование может использоваться как средство аутентификации сообщений, передаваемых через Интернет. Основная идея состоит в хешировании сообщения тайным способом. Полученное значение затем передается вместе с сообщением. Для проверки подлинности сообщения получатель хеширует полученное сообщение (таким же секретным способом) и убеждается в том, что вычисленное им значение соответствует полученному значению. (Предполагается, что шансы получить при хешировании искаженного сообщения тот же результат очень и очень малы.) Если полученное значение не соответствует исходному значению, сообщение считается поврежденным. Те, кого эта тема интересует, могут поискать в Интернете информацию об алгоритме MD5, который является хеш-функцией, широко используемой в приложениях аутентификации.

Полезно будет попробовать взглянуть на методы обнаружения ошибок как на применение метода хеширования для целей аутентификации. Например, использование битов четности, по существу, является системой хеширования, в которой битовая комбинация хешируется для получения значения либо 0, либо 1, после чего это значение передается вместе с исходной комбинацией. Если полученная в конечном счете комбинация битов не хешируется с тем же результатом, эта битовая комбинация считается поврежденной.

в котором может содержаться несколько записей. Записи распределяются по этим сегментам согласно некоторому алгоритму, преобразующему значение поля ключа в номер сегмента. (Алгоритм преобразования значения ключа в номер сегмента называют **хеш-функцией** или **функцией свертки**.) Каждая запись файла хранится в сегменте, определяемом этим процессом. Следовательно, запись можно извлечь, применив алгоритм хеширования к значению ее поля ключа для определения номера сегмента, а затем считав записи соответствующего сегмента и отыскав среди них нужную запись.

Хеширование используется не только как средство извлечения данных из массовой памяти, но и как средство извлечения отдельных элементов из больших блоков данных, хранящихся в основной памяти. Когда хеширование используется в структурах хранения данных в массовой памяти, результат называют **хеш-файлом**. При применении хеширования к структурам хранения данных в основной памяти компьютера результат обычно называют **хеш-таблицей**.

Давайте применим метод хеширования к классическому файлу с информацией о служащих, в котором каждая запись содержит информацию об одном работнике компании. Прежде всего создадим несколько доступных областей в массовой памяти, которые будут играть роль отдельных сегментов. Количество используемых сегментов и их размер являются важным вопросом проектирования, и к этой теме мы еще вернемся. Пока предположим, что создан 41 сегмент, который мы будем называть сегмент 0, сегмент 1, ..., сегмент 40. (Причина, по которой было решено использовать именно 41 сегмент, а не, скажем, 40, будет подробно обсуждаться чуть ниже.)

Договоримся, что в качестве поля ключа, идентифицирующего отдельную запись, будет использоваться идентификационный номер работника. Наша следующая задача — разработать функцию хеширования, которая позволит преобразовать любое значение поля ключа в номер сегмента. Вспомним, что хотя идентификационные "номера" работников могут иметь вид `25X3Z` или `J2X35`, а значит, они не являются числовыми значениями, эти значения в любом случае хранятся в памяти машины в виде битовых комбинаций, каждая из которых может интерпретироваться как числовое значение в двоичном представлении. Используя такую числовую интерпретацию, можно поделить любое хранящееся в памяти значение поля ключа на количество доступных сегментов и воспользоваться для наших целей остатком от деления, который в этом случае может принимать значения в диапазоне от 0 до 40. Таким образом, каждому возможному остатку от деления мы можем однозначно поставить в соответствие один из доступных 41 сегментов, как показано на рис. 9.18.

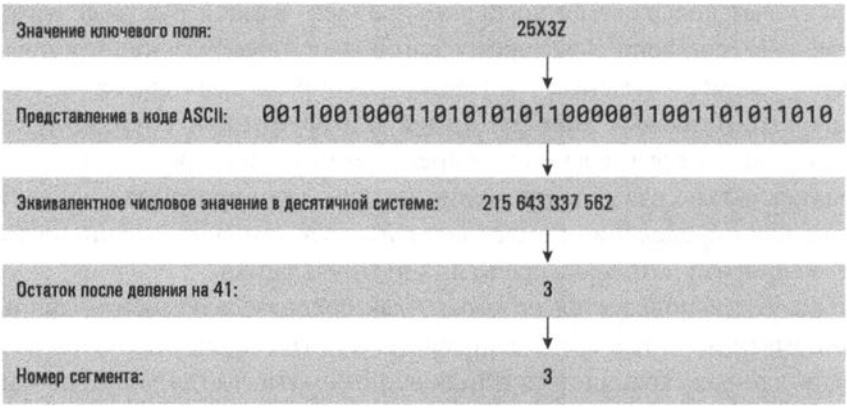


Рис. 9.18. Хеширование записи, значение которой в поле ключа равно 25X3Z, в один из 41 сегмента файла

Используя этот алгоритм в качестве хеш-функции, приступим к созданию файла, рассматривая каждую запись в отдельности и применяя выбранную хеш-функцию деления на 41 к ее ключу для получения номера сегмента, а затем сохраняя запись в этом сегменте, как показано на рис. 9.19. Позднее, если потребуется извлечь эту запись из файла, достаточно будет просто применить ту же хеш-функцию к ее ключу, чтобы определить соответствующий сегмент, а затем выполнить в этом сегменте поиск требуемой записи.

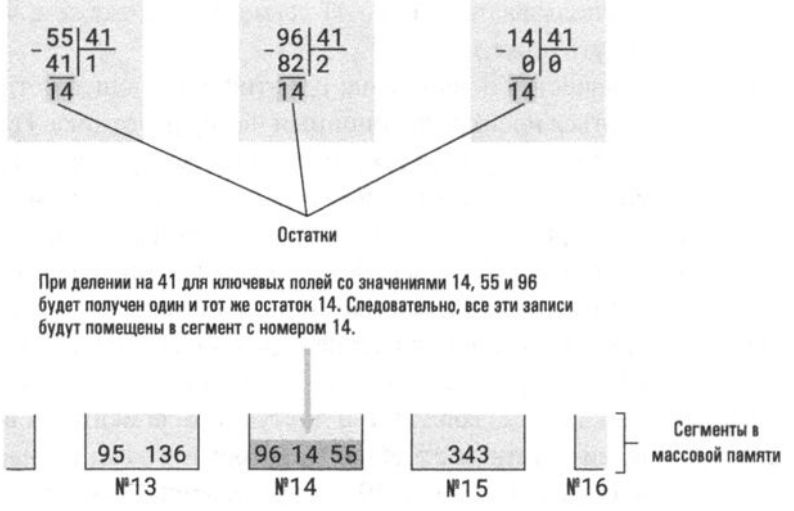


Рис. 9.19. Использование остатков от деления при создании хешированного файла

А теперь пришло время заново обдумать принятое выше решение разделить область хранения на 41 сегмент. Во-первых, обратите внимание, что для получения эффективной хеш-системы сохраняемые записи должны быть равномерно распределены по сегментам. Если непропорциональное количество ключей будет хешировано в один и тот же сегмент (явление, называемое **кластеризацией**), то в одном блоке будет сохранено непропорционально большое количество записей. А это, в свою очередь, может привести к тому, что для извлечения записей из этого сегмента потребуется длительный поиск, что в конечном счете приведет к потере всех преимуществ, получаемых за счет хеширования.

А теперь обратите внимание, что, если бы мы решили разделить область хранения на 40 сегментов, а не на 41, наша хеш-функция предусматривала бы деление ключей на значение 40, а не на 41. Но если и делимое, и делитель имеют в разложении общий множитель, то он будет присутствовать и в остатке. В частности, если бы все ключи к записям, хранящимся в нашем хеш-файле, оказались кратными 5 (а это число является одним из множителей числа 40), то множитель 5 появился бы и в остатках при делении на 40, и все записи оказались бы кластеризованными в тех сегментах, которые связаны с остатками 0, 5, 10, 15, 20, 25, 30 и 35. Аналогичные ситуации могут возникнуть и в тех случаях, когда ключи записей будут кратны 2, 4, 8, 10 и 20, поскольку все эти числа также являются множителями числа 40. Именно по этой причине было решено разделить область хранения на 41 сегмент, поскольку 41 — это *простое* число, что полностью исключает появление общих множителей у значений ключа и делителя в хеш-функции, а следовательно, снижает вероятность кластеризации.

К сожалению, возможность появления кластеризации никогда нельзя исключить полностью. Даже с хорошо спроектированной хеш-функцией весьма вероятно, что два ключа будут хешироваться в одно и то же значение — явление, называемое *коллизией*, причем даже в самом начале процесса создания файла. Чтобы понять, почему так происходит, рассмотрим следующий сценарий.

Предположим, что был найден некоторый алгоритм хеширования, произвольно распределяющий записи по всему 41 сегменту, и что исходно файл пуст и записи в него помещаются по одной. При помещении первой записи она обязательно попадает в пустой сегмент. Однако при помещении в файл второй записи только 40 из 41 сегмента являются пустыми; следовательно, вероятность того, что вторая запись будет помещена в пустой сегмент, составляет 40/41. Если предположить, что вторая запись также была помещена в пустой сегмент, то для третьей записи найдется только 39 пустых сегментов, и вероятность ее помещения в пустой сегмент составит 39/41. Продолжая этот процесс, можно обнаружить, что если первые семь записей файла помещены в пустые сегменты, то вероятность помещения восьмой записи в один из оставшихся пустых сегментов составит лишь 34/41.

Приведенный выше анализ позволяет вычислить вероятность того, что первые восемь записей будут помещены в пустые сегменты, как произведение вероятностей того, что каждая из этих записей помещена в пустой сегмент в предположении, что предыдущие также были помещены в пустые сегменты. Следовательно, эта вероятность вычисляется так:

```
(41/41)(40/41)(39/41)(38/41) . . . (34/41) = 0,482
```

Полученный результат меньше одной второй! Поэтому весьма вероятно, что, по крайней мере две из первых восьми записей будут распределены в один сегмент. Таким образом, весьма вероятно, что коллизия будет иметь место уже при помещении всего лишь восьми записей в файл, разделенный на 41 сегмент.

Высокая вероятность возникновения коллизий показывает, что независимо от того, насколько хорошо выбрана хеш-функция, любая хеш-система должна быть спроектирована с учетом возможности возникновения кластеризации. В частности, вполне возможно, что некоторый сегмент может заполниться и даже переполниться. Один из подходов к решению этой проблемы состоит в увеличении размера сегментов. Другой подход заключается в том, чтобы позволить контейнерам перетекать в область переполнения, которая будет специально зарезервирована для этой цели. В любом случае возникновение кластеризации и переполнения сегментов может значительно снизить производительность хеш-файла.

Исследования показали, что хеш-файлы, как правило, хорошо работают, пока отношение количества записей к максимально возможному числу записей в файле (отношение, известное как **коэффициент заполнения**) остается ниже 50 процентов. Если же коэффициент заполнения поднимается выше 75 процентов, производительность системы, как правило, существенно ухудшается (кластеризация начинает проявлять себя, вызывая заполнение некоторых сегментов и, возможно, их переполнение). По этой причине системы хранения с хешированием обычно реконструируется с целью получения большей общей емкости, когда коэффициент их заполнения приближается к 75-процентному значению. Можно сделать обоснованное заключение, что высокая эффективность извлечения записей, получаемая за счет реализации хешированных систем, не может быть достигнута без дополнительных затрат.

9.5. Вопросы и упражнения

1. Выполните слияние файлов, следуя алгоритму, представленному на рис. 9.15. Предположим, что один входной файл содержит записи со значениями поля ключа, эквивалентными `B` и `E`, а второй — со значениями, эквивалентными `A`, `C`, `D` и `F`.

2. Алгоритм слияния является основой популярного алгоритма сортировки, называемого сортировкой слиянием. Можете ли вы записать этот алгоритм? (*Подсказка*: любой непустой файл может рассматриваться как набор одноэлементных файлов.)
3. Характеристика "последовательный" является физическим или концептуальным свойством файла?
4. Какую последовательность действий необходимо выполнить, чтобы извлечь определенную запись из индексированного файла?
5. Объясните, как неудачный выбор хеш-функции может привести к тому, что работа с хешированным файлом мало чем будет отличаться от работы с обычным последовательным файлом.
6. Предположим, что хешированный файл создается с помощью описанного в тексте алгоритма хеширования, основанного на операции деления, но с выделением шести сегментов. Для каждого из приведенных ниже значений поля ключа определите номер сегмента, в который будет помещена запись с этим значением поля ключа. Что будет происходить не так, как хотелось бы, и почему?

   а. 24 **б.** 30 **в.** 3 **г.** 18 **д.** 15

   е. 21 **ж.** 9 **з.** 39 **и.** 27 **к.** 0
7. Сколько людей нужно собрать вместе, чтобы вероятность того, что у двух членов этой группы дни рождения совпадают, превысила одну вторую? Какое отношение эта задача имеет к материалу данного раздела?

9.6. Интеллектуальный анализ данных

Быстро развивающаяся область компьютерных наук, тесно связанная с технологиями баз данных, — это интеллектуальный анализ больших объемов данных (или "добыча данных"), предполагающий использование различных методов обнаружения скрытых шаблонов в больших массивах данных. Интеллектуальный анализ данных стал важным инструментом во многих областях, включая маркетинг, управление запасами, контроль качества, управление кредитными рисками, выявление мошенничества и анализ инвестиций. Методы интеллектуального анализа данных находят применение даже в тех ситуациях, когда это может показаться маловероятными, что иллюстрируется их использованием для идентификации функций конкретных генов, входящих в состав молекул ДНК, а также для выявления свойств организмов.



Основные положения для запоминания

- **Шаблоны могут обнаруживать себя при преобразовании данных с использованием вычислительных инструментов.**
- **Интеллектуальный анализ данных позволил внедрить важные инновации в медицине, бизнесе и науке.**

Деятельность, связанная с проведением интеллектуального анализа данных, отличается от традиционного опроса базы данных тем, что интеллектуальный анализ проводится с целью идентифицировать в данных ранее неизвестные шаблоны, — в отличие от традиционных запросов к базе данных, которые просто запрашивают извлечение сохраненных фактов. Более того, интеллектуальный анализ данных практикуется в статических наборах данных, называемых **хранилищами данных** (data warehouses), а не в интерактивных базах данных, которые постоянно обновляются. Эти хранилища часто представляют собой "моментальный снимок" содержимого базы данных или целого набора баз данных. Они используются вместо реальных активно действующих баз данных потому, что поиск шаблонов в статической системе выполнить проще, чем в динамической.

Следует также отметить, что предмет интеллектуального анализа данных (или добычи данных) не ограничивается лишь областью проведения вычислений, — он имеет ответвления, уходящие далеко в область статистического анализа. Фактически многие утверждают, что, поскольку основы интеллектуального анализа данных были заложены при попытках проведения статистического анализа больших и разнородных массивов данных, он является лишь ответвлением статистики, а не самостоятельной областью в компьютерных науках.

Двумя наиболее распространенными формами интеллектуального анализа данных являются **описание классов** и **различение классов**. Описание классов связано с выявлением свойств, которые характеризуют данную группу элементов данных, тогда как различение классов имеет дело с идентификацией свойств, которые разделяют определенные группы. Например, методы описания классов будут использоваться для определения характеристик людей, которые покупают небольшие экономичные транспортные средства, тогда как методы различения классов будут использоваться для поиска свойств, которые отличают людей, покупающих подержанные автомобили, от тех, кто покупает новые.

Другой формой интеллектуального анализа данных является **кластерный анализ**, проводимый с целью *обнаружения* классов. Обратите внимание, что эта задача отличается от задачи описания класса, цель которой — обнаружить общие свойства членов в пределах классов, которые уже определены. Если говорить точнее, то кластерный анализ проводится для того, чтобы найти такие

свойства элементов данных, которые приводят к выявлению их группировок. Например, если анализируется информация о возрасте людей, смотревших определенную кинокартину, то кластерный анализ может, например, обнаружить, что все данные можно разделить на две возрастные группы: от 4 до 10 лет и от 25 до 40 лет. (Возможно, кинофильм привлек детей и их родителей?)

Еще одной формой интеллектуального анализа данных является **ассоциативный анализ**, при котором осуществляется поиск связей между группами данных. Так, ассоциативный анализ может, например, выявить, что посетители магазина, которые покупают картофельные чипсы, также часто покупают пиво или газированную воду, либо что посетители, которые в будний день делают покупки в течение традиционного рабочего дня, чаще всего также получают пенсионные выплаты.

Анализ выбросов является еще одной формой интеллектуального анализа данных. В этом случае проводится попытка идентифицировать те записи данных, которые не соответствуют норме. Анализ выбросов может быть использован для выявления ошибок в массивах данных, для выявления кражи кредитных карт посредством обнаружения внезапных отклонений от обычных моделей покупок клиента и, возможно, для обнаружения потенциальных террористов путем выявления нетипичного поведения.



Основные положения для запоминания

- **Объединение источников данных, кластеризация данных и классификация данных являются частью процесса использования компьютеров для обработки информации.**

Биоинформатика

Достижения в области технологий баз данных и методов интеллектуального анализа данных расширяют перечень инструментов, доступных для биологов в различных областях исследований, включая идентификацию структур и классификацию органических соединений. В результате возникла новая область биологических наук, получившая название "биоинформатика". Биоинформатика берет свое начало от первых попыток расшифровать ДНК и в наши дни включает в себя такие задачи, как каталогизация белков и понимание последовательности взаимодействия белков (так называемые *биохимические пути*). Хотя обычно ее рассматривают как часть биологии, биоинформатика является ярким примером того, как компьютерные науки оказывают влияние на другие области знаний и даже проникают в них.

Наконец, еще одна форма интеллектуального анализа данных, называемая **анализом последовательных шаблонов**, пытается идентифицировать шаблоны поведения во времени. Например, анализ последовательных шаблонов может выявить тенденции в экономических системах, таких как фондовые рынки, или в системах окружающей среды, таких как климатические условия.

Как следует из последнего примера, результаты интеллектуального анализа данных могут быть использованы и для прогнозирования будущего поведения. Если сущность обладает свойствами, характерными для некоего класса, то она, вероятнее всего, будет и вести себя, как другие члены этого класса. Тем не менее многие проекты интеллектуального анализа данных направлены лишь на то, чтобы просто лучше понять анализируемые данные, — об этом свидетельствует использование интеллектуального анализа данных для раскрытия тайн ДНК. В любом случае сфера применения приложений интеллектуального анализа данных потенциально огромна, а это говорит о том, что область интеллектуального анализа данных обещает оставаться активной областью исследований еще на долгие годы.

Обратите внимание, что технологии баз данных и интеллектуальный анализ данных являются близкими родственниками, и, таким образом, исследования, проводимые в одной области, будут иметь определенные последствия и для другой. Методы баз данных широко используются для реализации в хранилищах данных инструментов представления информации в форме **кубов данных** (т.е. данные рассматриваются с *разных* точек зрения — термин "куб" здесь используется как наглядный образ объекта с несколькими измерениями), что как раз и делает возможным проведение интеллектуального анализа данных. В свою очередь, по мере того как исследователи в области проведения интеллектуального анализа данных будут совершенствовать методы реализации кубов данных, эти результаты будут приносить дивиденды и в области проектирования баз данных.

В заключение мы должны признать, что успешный интеллектуальный анализ данных предполагает нечто большее, чем просто идентификацию шаблонов в наборе данных. Для определения, можно ли считать найденные закономерности существенными или это просто совпадения, необходимо применять разумные суждения. Тот факт, что в конкретном магазине было продано большое количество выигрышных лотерейных билетов, вероятно, не следует считать важным для того, кто планирует купить лотерейный билет, но обнаружение того, что клиенты, покупающие легкие закуски, также склонны покупать и замороженные обеды, может представлять собой важную информацию для менеджера продуктового магазина. Аналогичным образом интеллектуальный анализ данных включает в себя огромное количество этических вопросов, касающихся прав отдельных лиц, информация о которых представлена в хранилище данных, точности и способов использования сделанных выводов и даже целесообразности собственно проводимого анализа данных в первую очередь.

9.6. Вопросы и упражнения

1. Почему интеллектуальный анализ данных не проводится в интерактивных базах данных?
2. Приведите дополнительный пример шаблона, который может быть найден каждым из типов интеллектуального анализа данных, обсуждавшихся в этом разделе.
3. Укажите несколько разных точек зрения, которые могут использоваться в технологии куба данных при анализе сведений о продажах.
4. Чем интеллектуальный анализ данных отличается от традиционных запросов к базам данных?

9.7. Влияние технологий баз данных на общество

С развитием технологий баз данных стала доступной информация, которая когда-то была скрыта и практически недоступна. Во многих случаях автоматизированные библиотечные системы обеспечивают легкость доступа к сведениям о предпочтениях читателей, предприятия розничной торговли ведут учет покупок своих клиентов, а поисковые системы Интернета собирают информацию о запросах своих пользователей. В свою очередь, эта информация становится потенциально доступной для маркетинговых фирм, правоохранительных органов, политических партий, работодателей и даже частных лиц.



Основные положения для запоминания

- **Проблемы безопасности и конфиденциальности возникают в связи с доступностью данных, содержащих личную информацию.**

Все это дает представление о потенциальных проблемах, которые пронизывают весь спектр существующих приложений баз данных. Технология позволяет достаточно легко собирать огромные объемы данных и объединять или сравнивать разные их наборы, чтобы получить информацию о связях, которые в противном случае остались бы погребенными под грудой фактов. Правовые и этические последствия этого, как положительные, так и отрицательные, огромны. И сейчас эти последствия являются не просто предметом академических дебатов, а реальностью нашего времени.



Основные положения для запоминания

- **Инновации, связанные с развитием компьютерных технологий, вызывают множество юридических и этических проблем.**

В одних случаях процесс сбора данных очевиден, а в других — скрыт. Примеры первой ситуации имеют место, когда кто-то явно просит предоставить ему информацию. При проведении опросов населения или заполнении конкурсных регистрационных форм этот сбор данных проходит на добровольной основе, однако на основании решений правительственных органов могут проводиться и принудительные сборы данных. Иногда ответ на вопрос, носит ли сбор данных принудительный или добровольный характер, может зависеть лишь от точки зрения. Так, ответ на вопрос, будет ли ваше согласие на требование предоставить информацию личного характера при предоставлении займа добровольным или принудительным, зависит от того, является ли его получение для вас простым удобством или же необходимостью. При оплате покупки с помощью кредитной карточки некоторые торговые предприятия теперь требуют согласия покупателя на представление его личной подписи в оцифрованном виде. И вновь, является ли это обязательным требованием или только пожеланием, зависит от характера приобретения.

При более скрытом сборе данных прямого контакта с объектом избегают. Примерами могут служить кредитные компании, осуществляющие сбор данных о покупках, совершенных владельцами их кредитных карточек; организации, собирающие информацию о посетителях принадлежащих им веб-узлов; сотрудники правоохранительных органов, записывающие номерные знаки автомобилей, припаркованных у стен некоторого учреждения. В этих случаях объект опроса может быть вообще не осведомлен о сборе информации и тем более о существовании баз данных, в которых будут сохранены касающиеся его данные.

Иногда такие факты становятся вполне очевидными, если просто остановиться и подумать. Например, продуктовый магазин может предлагать скидки тем постоянным покупателям, которые в нем предварительно зарегистрируются. В процессе регистрации клиенту выдают личную дисконтную карточку, которую он должен предъявлять при покупке для получения скидки. В результате магазин получает возможность вести запись всех покупок клиента, а ценность этой информации может быть значительно выше размера предоставленных ему скидок.

Конечно, движущей силой этого бума в сборе данных является ценность данных, которая усиливается новыми достижениями в технологии баз данных, позволяющими связывать данные такими способами, которые раскрывают информацию, которая в противном случае оставалась бы незамеченной. Например,

модели покупки владельцев кредитных карт могут быть классифицированы и внесены в перекрестный список для получения профилей клиентов, имеющих огромную маркетинговую ценность. Подписные формы для журналов по бодибилдингу могут быть отправлены по почте тем, кто недавно приобрел тренажеры, тогда как подписные формы для журналов о дрессировке собак могут быть адресованы тем, кто недавно приобрел корм для собак. Альтернативные способы объединения информации иногда бывают очень творческими. Так, данные об уровне благосостояния сравнивались с данными о судимости с целью найти и привлечь к ответственности нарушителей режима условного досрочного освобождения, а в 1984 году призывная комиссия США использовала список дней рождения граждан, полученный от фирмы — популярного поставщика мороженого, для определения лиц, уклонившихся от регистрации в ходе призывной кампании.

Существует несколько подходов к организации защиты общества от злоупотреблений с использованием баз данных. Один из них состоит в применении ограничений правового характера. К сожалению, принятие закона не исключает возможности определенного действия; чаще всего такое действие просто переходит в разряд нелегальных. Одним из ярких примеров является принятие в 1974 году закона США о защите частной жизни, призванного, в частности, защитить граждан от злоупотреблений информацией из государственных баз данных. В этом законе говорится, что государственные агентства должны сообщать о существовании таких баз данных в едином федеральном регистре, чтобы каждый гражданин имел возможность доступа к информации о самом себе и проверки ее достоверности. Однако государственные агентства медлили с исполнением этого требования. Во многих случаях это было скорее следствием бюрократичности, нежели злонамеренности. Однако сам факт, что бюрократия способна создавать базы данных с личными данными, о существовании которых ничего неизвестно, не успокаивает.

Другим, может быть, более действенным средством контроля за злоупотреблениями при использовании баз данных является общественное мнение. Базы данных не будут использоваться недопустимым образом, если потери от этого превзойдут возможный выигрыш. Для большинства компаний не существует более страшного наказания, чем неблагоприятное общественное мнение. В начале 1990-х годов именно общественное мнение положило конец продаже крупными кредитными компаниями списков адресов своих клиентов для целей маркетинга. Не так давно, в 2011 году, корпорация Google прекратила использование своего инструмента социальных сетей Google Buzz, после того как реализованный в нем автоматический обмен контактной информацией, взятой из популярной почтовой службы Gmail, был встречен резкой общественной критикой. Даже государственным учреждениям приходилось сгибаться

под давлением общественного мнения. В 1997 году Министерство социальной защиты США изменило свой план сделать доступными в Интернете записи о социальной защите граждан, после того как общественное мнение выразило озабоченность по поводу защиты этих данных. В описанных случаях результат был получен в течение считанных дней, хотя для правового урегулирования проблемы через суды потребовалось бы существенно больше времени.



Основные положения для запоминания

- **Когда конфиденциальность и другие аспекты защиты информации игнорируются, может иметь место незаконный сбор и использование информации со стороны коммерческих и правительственных структур.**

Безусловно, в большинстве случаев приложения баз данных приносят выгоду как их владельцу, так и источнику информации, однако здесь имеет место определенная потеря конфиденциальности, которую не следует оставлять без внимания. Если данные верны, то такая потеря конфиденциальности является серьезной проблемой, однако если информация ошибочна, то последствия могут иметь просто катастрофический характер. Представьте себе, что должен чувствовать человек, узнавший о неблагоприятном изменении его кредитного рейтинга из-за ошибки при вводе информации. Подумайте, насколько осложнится возникшая перед этим человеком проблема, если эта ошибочная информация была открыта для совместного доступа и со стороны других учреждений. Проблемы конфиденциальности данных есть и останутся в будущем важнейшим побочным эффектом развития технологий в целом и технологии баз данных в частности. Решение подобных проблем требует продуманных, сознательных и активных действий со стороны всего общества.

9.7. Вопросы и упражнения

1. Должны ли правоохранительные органы обладать правом доступа к базам данных для определения лиц с преступными наклонностями, даже если ранее они не совершали каких-либо преступлений?
2. Должны ли страховые компании иметь доступ к данным о возможных потенциальных проблемах со здоровьем своих клиентов, если клиенты внешне не проявляют каких-либо симптомов заболеваний?
3. Предположим, что в финансовом отношении вы преуспеваете. Какие позитивные для вас последствия могут иметь место, если информация

об этом будет доступна самому широкому кругу организаций? Какие негативные последствия может иметь распространение подобной информации? А что вы обо всем этом скажете, если ваше финансовое положение оставляет желать лучшего?

4. Какова роль независимых средств массовой информации в контроле над использованием баз данных не по прямому назначению? (Например, в какой степени пресса может воздействовать на общественное мнение или разоблачать преступления?)

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.)

1. Перечислите различия между плоским файлом и базой данных.
2. Что понимается под независимостью данных?
3. Какова роль СУБД в многоуровневом подходе к реализации базы данных?
4. В чем состоят различия между схемой и подсхемой?
5. Опишите два основных преимущества разделения функций прикладного программного обеспечения и функций СУБД.
6. Опишите сходство между абстрактным типом данных (глава 8) и моделью базы данных.
7. Определите, в чью компетенцию (пользователя, прикладного программиста, разработчика СУБД) входит каждый из следующих вопросов.
   а. Как обеспечить наиболее эффективное хранение данных на диске?
   б. Есть ли свободные места на рейс № 243?
   в. Как отношения должны быть представлены в массовой памяти компьютера?
   г. Сколько раз следует разрешить пользователю ошибаться при вводе пароля, прежде чем система должна будет отказаться от работы с ним?
   д. Каким образом может быть реализована операция PROJECT?
8. Какие из приведенных ниже задач решаются средствами СУБД?
   а. Предоставление гарантии, что доступ пользователя к базе данных будет ограничен соответствующей подсхемой.

б. Перевод команд, представленных в терминах модели базы данных, в последовательности действий, соответствующих особенностям реальной системы хранения данных.

в. Сокрытие того факта, что данные в базе фактически разбросаны по многим компьютерам в сети.

9. Опишите, каким образом приведенная ниже информация об авиакомпаниях, рейсах (на конкретный день) и пассажирах может быть представлена в реляционной базе данных.

```
Авиалинии: "Clear Sky", "Long Hop", "Tree Top"
Рейсы компании Clear Sky: CS205, CS37, CS102
Рейсы компании Long Hop: LH67, LH89
Рейсы компании Tree Top: TT331, TT809
Пассажир Смит зарезервировал билеты на рейсы:
  CS205 (место 12B), CS37 (место 18C) и LH89 (место 14A)
Пассажир Бейкер зарезервировал билеты на рейсы:
  CS37 (место 18B) и LH89 (место 14B)
Пассажир Кларк зарезервировал билеты на рейсы:
  LH67 (место 5A) и TT331 (место 4B)
```

10. До какой степени важен порядок, в котором операции SELECT и PROJECT применяются к отношению? Иначе говоря, при каких условиях выполнение операции SELECT, а затем операции PROJECT даст тот же результат, что и при обратном порядке их выполнения, т.е. сначала операция PROJECT, а затем — операция SELECT?

11. Приведите аргумент, показывающий, что предложение `where` в операции JOIN, как это описано в разделе 9.2, не является необходимым. (То есть покажите, что любой запрос, использующий предложение `where`, может быть переопределен с использованием операции JOIN, которая осуществляет конкатенацию каждого кортежа в одном отношении с каждым кортежем в другом.)

12. Исходя из представленных ниже отношений покажите, как будет выглядеть отношение `Result` после выполнения следующих операций.

Отношение X

U	V	W
A	Z	5
B	D	3
C	Q	5

Отношение Y

R	S
3	J
4	K

а. `Result ¬ PROJECT W from X`

б. `Result ¬ SELECT from X where W = 5`

в. `Result ¬ PROJECT S from Y`

г. `Result ¬ JOIN X and Y where X.W ¤ Y.R`

13. Исходя из представленных ниже отношений `Part` (Деталь) и `Manufacturer` (Изготовитель) и воспользовавшись командами `SELECT`, `PROJECT` и `JOIN`, напишите последовательность операций, необходимых для получения ответа на каждый из приведенных ниже вопросов о деталях и их изготовителях.

Отношение Part

PartName	Weight
Винт 2X	1
Винт 2Z	1,5
Гайка V5	0,5

Отношение Manufacturer

CompanyName	PartName	Cost
Компания X	Винт 2Z	0,03
Компания X	Гайка V5	0,01
Компания Y	Винт 2X	0,02
Компания Y	Гайка V5	0,01
Компания Y	Винт 2Z	0,04
Компания Z	Гайка V5	0,01

а. Какие компании производят деталь "Винт 2Z"?

б. Получите список деталей, производимых компанией X, с указанием их стоимости (атрибут `Cost`).

в. Какие компании выпускают детали, вес которых (атрибут `Weight`) равен 1?

14. Выполните задание 13, используя язык SQL.

15. Используя команды `SELECT`, `PROJECT` и `JOIN`, напишите последовательности операций, необходимые для получения ответа на вопросы, касающиеся отношений `EMPLOYEE`, `JOB` и `ASSIGNMENT`, представленных на рис. 9.5.

а. Получите список имен и адресов сотрудников компании.

б. Получите список имен и адресов тех, кто работал и работает в отделе кадров.

в. Получите список имен и адресов тех, кто работает в отделе кадров.

16. Выполните задание 15, используя язык SQL.

17. Разработайте структуру реляционной базы данных, предназначенной для хранения информации о композиторах, их биографиях и списках созданных ими произведений. (Избегайте избыточности данных, подобной той, которая присутствует на рис. 9.4.)

18. Разработайте структуру реляционной базы данных, предназначенной для хранения информации о музыкальных исполнителях, записях в их

исполнении и тех композиторах, произведения которых были записаны в их исполнении. (Избегайте избыточности данных, подобной той, которая присутствует на рис. 9.4.)

19. Разработайте структуру реляционной базы данных, предназначенной для хранения информации о производителях компьютерного оборудования и выпускаемой ими продукции. (Избегайте избыточности данных, подобной той, которая присутствует на рис. 9.4.)
20. Разработайте структуру реляционной базы данных, предназначенной для хранения информации об издательствах, журналах и их подписчиках. (Избегайте избыточности данных, подобной той, которая присутствует на рис. 9.4.)
21. Разработайте структуру реляционной базы данных, содержащей информацию о деталях, поставщиках и заказчиках. Предусмотрите, что каждая деталь может поставляться несколькими поставщиками и заказываться несколькими заказчиками. Каждый поставщик может поставлять множество деталей и обслуживать нескольких заказчиков. Наконец, каждый заказчик может заказывать множество деталей у разных поставщиков, причем одну и ту же деталь можно заказывать у разных поставщиков.
22. Составьте последовательность операторов, используя операции `SELECT`, `PROJECT` и `JOIN`, чтобы получить значения полей `Jobld`, `StartDate` и `TermDate` для каждой из должностей в бухгалтерии из реляционной базы данных, представленной на рис 9.5.
23. Выполните задание 22, используя язык SQL.
24. Составьте последовательность операторов (используя операции `SELECT`, `PROJECT` и `JOIN`), чтобы получить информацию о значениях полей `Name`, `Address`, `JobTitle` и `Dept` для каждого работающего в настоящий момент сотрудника из реляционной базы данных, представленной на рис 9.5.
25. Выполните задание 24, используя язык SQL.
26. Составьте последовательность операторов (используя операции `SELECT`, `PROJECT` и `JOIN`), чтобы получить информацию о значениях полей `Name` и `JobTitle` для каждого работающего в настоящий момент сотрудника из реляционной базы данных, представленной на рис 9.5.
27. Выполните задание 26, используя язык SQL.
28. Чем отличается информация из таблицы

Name	Department	TelephoneNumber
Джонс	Отдел сбыта	555-2222
Смит	Отдел сбыта	555-3333
Бейкер	Отдел кадров	555-4444

от информации в двух следующих таблицах:

Name	Department
Джонс	Отдел сбыта
Смит	Отдел сбыта
Бейкер	Отдел кадров

Department	TelephoneNumber
Отдел сбыта	555-2222
Отдел сбыта	555-3333
Отдел кадров	555-4444

29. Разработайте реляционную базу данных, содержащую информацию об автомобильных узлах и входящих в них деталях. Предусмотрите возможность того, что каждый узел может не только состоять из меньших узлов и деталей, но в то же время являться составной частью большего узла.

30. Выберите популярный веб-сайт, такой как `http://www.google.com`, `www.amazon.com` или `www.ebay.com`, и создайте реляционную базу данных, которую вы могли бы предложить использовать в качестве вспомогательной базы данных для этого сайта.

31. Исходя из информации в базе данных, представленной на рис. 9.5, сформулируйте запрос, ответ на который дает следующий фрагмент программного кода:

```
TEMP ¬ SELECT from ASSIGNMENT
  where TermDate = '*'
RESULT ¬ PROJECT JobId, StartDate
  from TEMP
```

32. Запишите сформулированный в задании 29 запрос средствами языка SQL.

33. Исходя из информации в базе данных, представленной на рис. 9.5, сформулируйте запрос, ответ на который дает следующий фрагмент программного кода:

```
TEMP1  ¬ JOIN EMPLOYEE and ASSIGNMENT
  where EMPLOYEE.EmplId =
    ASSIGNMENT.EmplId
TEMP2  ¬ SELECT from TEMP1 where
  TermDate = '*'
RESULT ¬ PROJECT name, StartDate
  from TEMP2
```

34. Запишите сформулированный в задании 33 запрос средствами языка SQL.

35. Исходя из информации в базе данных, представленной на рис. 9.5, сформулируйте запрос, ответ на который дает следующий фрагмент программного кода:

```
TEMP1  ¬ JOIN EMPLOYEE and JOB
  where EMPLOYEE.EmplId = JOB.EmplId
TEMP2  ¬ SELECT from TEMP1 where
  Dept = 'SALES'
RESULT ¬ PROJECT Name from TEMP2
```

36. Запишите сформулированный в задании 35 запрос средствами языка SQL.

37. Преобразуйте следующий SQL-запрос в последовательность операций SELECT, PROJECT и JOIN:

```
SELECT Job.JobTitle
FROM Assignment, Job
WHERE Assignment.JoblId = Job.JobId
  AND Assignment.EmplId = '34Y70'
```

38. Преобразуйте следующий SQL-запрос в последовательность операций SELECT, PROJECT и JOIN:

```
SELECT Assignment.StartDate
FROM Assignment, Employee
WHERE Assignment.EmplId =
  Employee.EmplId
  AND Employee.Name =
    'Joe E. Baker'
```

39. Какой эффект окажет на базу данных из задания 13 выполнение следующего SQL-оператора:

```
INSERT INTO Manufacturer
VALUES ('Company Z', 'Bolt 2X', .03)
```

40. Какой эффект окажет на базу данных из задания 13 выполнение следующего SQL-оператора:

```
UPDATE Manufacturer
  SET Cost = .03
  WHERE CompanyName = 'Company Y'
    AND PartName = 'Bolt 2X'
```

***41.** Назовите несколько объектов, которые можно использовать в объектно-ориентированной базе данных для ведения учета товаров на складе продуктового магазина. Какие методы могут понадобиться этим объектам?

***42.** Назовите несколько объектов, которые можно использовать в объектно-ориентированной базе данных для ведения картотеки читателей библиотеки. Какие методы могут понадобиться этим объектам?

*43. Какие ошибки будут внесены в информацию при следующем выполнении транзакций T1 и T2?

Транзакция T1 предназначена для вычисления суммы остатка на счетах A и B, а транзакция T2 — для перечисления 100 долларов со счета A на счет B. Транзакция T1 начинает свое выполнение первой и считывает значение остатка на счете A. Затем запускается транзакция T2 и выполняет соответствующие перечисления. Наконец транзакция T1 считывает остаток на счете B и вычисляет итоговое значение.

*44. Объясните, каким образом описанный в этой главе протокол блокировки может разрешить проблему, возникшую в сценарии, предложенном в задании 43?

*45. Объясните, какое значение мог бы иметь описанный в этой главе протокол принудительной отмены/ожидания для изложенной в задании 43 последовательности событий, если предположить, что транзакция T1 запущена после транзакции T2? А если транзакция T2 запущена после транзакции T1?

*46. Предположим, что одна транзакция предпринимает попытку увеличить на 100 долларов сумму счета, остаток на котором составляет 200 долларов, а другая — уменьшить значение этого счета на 100 долларов. Покажите, какой вариант чередования отдельных этапов этих транзакций может привести к получению остатка, равного 100 долларам? А какой вариант чередования отдельных этапов этих транзакций может привести к получению остатка, равного 300 долларам?

*47. В чем разница между транзакцией, имеющей монопольный доступ, и транзакцией, имеющей общий доступ к элементу в базе данных, и почему это различие важно?

*48. Проблемы, обсуждаемые в разделе 9.4 и связанные с одновременными транзакциями, не ограничиваются лишь областью баз данных. Какие схожие проблемы могут возникнуть при доступе к документу с помощью текстовых процессоров? (Если у вас есть компьютер с текстовым процессором, попробуйте получить доступ к одному и тому же документу с помощью двух активаций текстового процессора и посмотрите, что произойдет.)

*49. Предположим, что последовательный файл содержит 50 000 записей и для просмотра одной записи требуется 5 миллисекунд. Сколько времени потребуется для доступа к записи в середине файла?

*50. Перечислите этапы выполнения алгоритма слияния, приведенного на рис. 9.15, если в исходном состоянии один из входных файлов будет пуст.

***51.** Модифицируйте алгоритм, приведенный на рис. 9.15, так, чтобы он позволял обрабатывать случаи, когда оба входных файла содержат запись с одинаковым значением в поле ключа. Исходите из того, что эти записи идентичны и только одна из них должна присутствовать в выходном файле.

***52.** Разработайте систему, с помощью которой хранящийся на диске файл можно будет обрабатывать как последовательный файл, имеющий одну из двух различных упорядоченностей.

***53.** Опишите, как можно создать последовательный файл, содержащий информацию о подписчиках журнала, используя текстовый файл в качестве базовой структуры.

***54.** Разработайте метод, с помощью которого последовательный файл, логические записи которого не имеют одного и того же фиксированного размера, может быть реализован как текстовый файл. Например, предположим, что необходимо создать последовательный файл, в котором каждая логическая запись будет содержать информацию о некотором писателе, а также список работ этого автора.

***55.** Какие преимущества имеет индексированный файл по сравнению с хешированным? Каковы преимущества хешированного файла перед индексированным?

***56.** В тексте главы проведена параллель между традиционным индексированным файлом и системой каталогов файлов, которую ведет операционная система. Чем отличается система каталогов файлов операционной системы от традиционного индекса?

***57.** Если хешированный файл разделен на 10 сегментов, то какова вероятность того, что по крайней мере две из трех произвольно взятых записей будут помещены в один и тот же сегмент? (Предполагается, что в алгоритме хеширования ни один из сегментов не имеет приоритета.) Сколько записей должно быть помещено в файл, чтобы вероятность возникновения коллизии превысила 0,5?

***58.** Выполните предыдущее задание, предполагая, что файл разделен на 100 сегментов вместо 10.

***59.** В каком сегменте следует искать запись, двоичное значение которой в поле ключа интерпретируется как число 124, если в качестве алгоритма хеширования используется метод деления, описанный в этой главе, а пространство хранения файла разделено на 23 сегмента?

***60.** Сравните реализацию хешированного файла с организацией однородного двухмерного массива. В чем сходство хеш-функции и адресного полинома?

*61. Охарактеризуйте преимущества:

а. последовательного файла по сравнению с индексированным;

б. последовательного файла по сравнению с хешированным;

в. индексированного файла по сравнению с последовательным;

г. индексированного файла по сравнению с хешированным;

д. хешированного файла по сравнению с последовательным;

е. хешированного файла по сравнению с индексированным.

*62. В каком смысле последовательный файл аналогичен связанному списку?

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также понять, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. В США записи обо всех федеральных заключенных сохраняются в единой базе данных в целях проведения криминальных расследований. Этично ли использовать эту информацию в других целях, скажем для генетических исследований? Если да, то для каких целей? Если нет, то почему? Каковы "за" и "против" в обоих случаях?
2. В какой степени университеты обладают правом распространять информацию о своих студентах? Допустимо ли это в отношении имен и адресов студентов? Можно ли распространять данные о показателях успеваемости в университете без указания имен студентов? Согласуется ли ваш ответ с ответом на вопрос 1?
3. Какие ограничения могут применяться при разработке структуры базы данных для хранения персональной информации о людях? Какую информацию о гражданах своей страны имеют право хранить государственные органы? Какую информацию о своих клиентах имеет право хранить страховая компания? Какую информацию о своих сотрудниках имеет право хранить любая компания? Должен ли в этих областях осуществляться контроль и если должен, то как?

4. Имеет ли право кредитная компания продавать маркетинговым фирмам информацию о покупательских предпочтениях владельцев выпущенных ею кредитных карточек? Может ли фирма, специализирующаяся на заказах спортивных автомобилей по почте, продавать список почтовых адресов своих клиентов журналам, освещающим ту же тематику? Допустимо ли, чтобы налоговая администрация продавала биржевым брокерам адреса и имена состоятельных налогоплательщиков? Если вы не можете безоговорочно ответить на вопрос "да" или "нет", то что бы вы предложили в качестве приемлемого набора правил?
5. В какой мере разработчик базы данных ответствен за возможное использование информации не по прямому назначению?
6. Предположим, что вследствие ошибки СУБД злоумышленник смог получить несанкционированный доступ к базе данных. Если в результате этого некоторая информация была извлечена и использована неблагоприятным образом, то в какой степени разработчики базы данных должны разделять ответственность за это? Зависит ли ответ на этот вопрос от тех усилий, которые злоумышленник вынужден был предпринять для обнаружения в конструкции базы данных той бреши, которая позволила ему осуществить это проникновение?
7. Широкое распространение интеллектуального анализа данных поднимает многочисленные вопросы этики и конфиденциальности. Нарушается ли ваша конфиденциальность, если при подобном анализе данных обнаруживаются определенные характеристики всех тех, кто входит в ваш коллектив, сообщество или общину? Способствует ли использование интеллектуального анализа данных хорошей деловой практике или же, наоборот, фанатизму? Насколько уместно заставлять граждан участвовать в переписи, зная, что из данных будет извлечено больше информации, чем ее содержится в отдельных опросных листах? Дает ли интеллектуальный анализ маркетинговым фирмам несправедливое преимущество перед ничего не подозревающей аудиторией? Насколько профилирование хорошо или плохо?
8. В какой степени отдельным лицам или организациям можно позволить собирать и хранить информацию об отдельных людях? Что если собранная информация уже доступна всем желающим, хотя и разбросана по разным источникам? В какой степени отдельный человек или компания должна защищать такую информацию?
9. Многие библиотеки предлагают справочную службу, чтобы посетители могли обращаться за помощью к библиотекарю при поиске информации. Может ли существование Интернета и технологий баз данных сделать

эту услугу устаревшей? Если это так, будет ли это шагом вперед или назад? Если нет, то почему? Как существование Интернета и технологий баз данных может повлиять на существование самих библиотек?

10. В какой степени вы подвержены риску возможной кражи личных данных? Какие шаги вы можете предпринять, чтобы минимизировать этот риск? Как вы можете пострадать, если стали жертвой кражи личных данных? Должны ли вы нести ответственность за кражу ваших личных данных?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Berstein A., Kifer M., Lewis P.M. *Database Systems*, 2nd ed. — Boston, MA: Addision-Wesley, 2006.
2. Connolly T., Beg C.E. *Database Systems: A Practical Approach to Design, Implementation and Management*, 5th ed. — Boston, MA: Addison-Wesley, 2009.
3. Date C.J. *An Introduction to Database Systems*, 8th ed. — Boston, MA: Addison-Wesley, 2004. (Имеется русский перевод этой книги: Дейт К.Дж. *Введение в системы баз данных*, 8-е изд. — М.: Издательский дом "Вильямс", 2005.)
4. Date C.J. *Databases, Types and the Relational Model*, 3rd ed. — Boston, MA: Addison-Wesley, 2007.
5. Elmasri R., Navathe S. *Fundamentals of Database Systems*, 6th ed. — Boston, MA: Addison-Wesley, 2011.
6. Patrick J.J. *SQL Fundamentals*, 3rd ed. — Upper Saddle River, NJ: Prentice-Hall, 2009.
7. Silberschatz A., Korth H., Sudarshan S. *Database Systems Concepts*, 6th ed. — New York: McGraw-Hill, 2009.
8. Ullman J.D., Widom J.D. *A First Course in Database Systems*, 3rd ed. — Upper Saddle River, NJ: Prentice-Hall, 2008.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Форта Б. *SQL за 10 минут*, 4-е изд. — М.: Издательский дом "Вильямс", 2014.
2. Грофф Дж.Р., Вайнберг П.Н., Оппель Э.Дж. *SQL: полное руководство*, 3-е изд. — М.: Издательский дом "Вильямс", 2010.

3. Коннолли Т., Бегг К., Страчан А. *Базы данных: проектирование, реализация и сопровождение. Теория и практика.* — М.: Издательский дом "Вильямс", 2003.
4. Ролланд Ф. *Реляционные базы данных. Краткий справочник.* — М.: Издательский дом "Вильямс", 2002.
5. Гарсиа-Молина Г., Ульман Дж.Д., Уидом Дж. *Системы баз данных. Полный курс.* — М.: Издательский дом "Вильямс", 2002.
6. Джордан Д. *Обработка объектных баз данных в C++. Программирование по стандарту ODMG.* — М.: Издательский дом "Вильямс", 2001.
7. Риккарди Г. *Системы баз данных. Теория и практика использования в Internet и среде Java.* — М.: Издательский дом "Вильямс", 2001.

Глава 10

В этой главе мы исследуем компьютерную графику — область компьютерных наук, которая оказывает большое влияние на производство кинофильмов и интерактивных видеоигр. И действительно, достижения в компьютерной графике освобождают визуальные медиа от ограничений реальности, и многие утверждают, что компьютерная анимация может вскоре вообще заменить потребность в настоящих актерах, съемочных площадках и киносъемках в индустрии кино и телевизионной продукции.

Компьютерная графика

10.1. ПРЕДМЕТ КОМПЬЮТЕРНОЙ ГРАФИКИ

10.2. ЧТО ТАКОЕ 3D-ГРАФИКА

10.3. МОДЕЛИРОВАНИЕ
- Моделирование отдельных объектов
- Моделирование целых сцен

10.4. РЕНДЕРИНГ
- Взаимодействие света с поверхностью
- Отсечение, построчное сканирование и удаление невидимых поверхностей
- Затенение
- Специализированное оборудование для реализации конвейера рендеринга

*10.5. МОДЕЛИРОВАНИЕ ГЛОБАЛЬНОГО ОСВЕЩЕНИЯ
- Трассировка лучей
- Метод излучательности

10.6. АНИМАЦИЯ
- Основы анимации
- Кинематика и динамика
- Процесс анимации

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

Компьютерная графика — это область компьютерных наук, в которой компьютерные технологии применяются для создания визуальных представлений и манипулирования ими. Эта область охватывает широкое разнообразие тем, включая представление текста, построение графиков и диаграмм, разработку графических пользовательских интерфейсов, обработку фотографий, создание видеоигр и анимированных кинофильмов. Однако термин "компьютерная графика" все чаще используется в отношении более узкой конкретной области, называемой трехмерной графикой (или 3D-графикой), и большая часть этой главы посвящена именно этой теме. А начнем мы с определения трехмерной графики и выяснения ее роли в более широкой интерпретации области компьютерной графики.

10.1. Предмет компьютерной графики

С появлением цифровых фотокамер популярность программного обеспечения для манипулирования цифровыми изображениями стала быстро возрастать. Это программное обеспечение позволяет "подправлять" фотографии, устраняя их дефекты и страшные "красные глаза", а также вырезать и вставлять фрагменты разных фотографий с целью создания изображений, которые не обязательно отражают реальность.

Подобные методы часто применяются для создания специальных эффектов в кино- и телевизионной индустрии. Фактически для этих отраслей подобные приложения являлись основным мотивирующим фактором перехода от аналоговых систем, таких как кинопленка, к цифровым кодированным изображениям. Подобные приложения обеспечивают удаление из кадра различных вспомогательных структур, подобных подвескам и страховкам, наложение нескольких изображений или создание коротких последовательностей новых изображений, которые используются для изменения действия, которое первоначально было отснято камерой.



Основные положения для запоминания

- **Цифровые изображения могут быть созданы путем построения пиксельных структур, манипулирования существующими цифровыми изображениями или объединения изображений.**

В дополнение к программному обеспечению для манипулирования цифровыми фотографиями и кадрами кинофильмов в настоящее время существует

широкий спектр пакетов утилит и прикладных программ, позволяющих создавать двухмерные изображения, начиная от простых линейных чертежей и заканчивая сложными произведениями искусства. (Простейшим всем известным примером является приложение в ОС Microsoft Windows под названием Paint.) Как минимум эти программы предоставляют пользователю инструменты для рисования точек и линий, отрисовки простых геометрических фигур (таких, как овалы или прямоугольники), заливки областей изображения цветом или текстурой, а также вырезания и вставки указанных фрагментов рисунка. Производители кинофильмов, телевизионных программ и компьютерных игр могут использовать для своих целей уже существующие коммерческие инструменты, но могут и создавать собственные программные средства, предназначенные для реализации новых визуальных эффектов и анимации.



Основные положения для запоминания

- **Цифровые эффекты и анимации могут быть созданы с использованием существующего или модифицированного программного обеспечения, включающего в себя функции реализации различных эффектов и анимации.**

Обратите внимание: все упомянутые выше приложения предназначены для манипулирования лишь плоскими двухмерными фигурами или изображениями. Это означает, что они являются представителями двух смежных областей компьютерной графики: одна из них — **2D-графика**, а другая — **обработка изображений**. Различие между ними состоит в том, что *2D-графика* фокусируется на задаче преобразования двухмерных фигур (кругов, прямоугольников, букв и т.д.) в структуры пикселей с целью получения соответствующего изображения, тогда как *обработка изображений*, с которой мы познакомимся позже, при изучении искусственного интеллекта, фокусируется на анализе пикселей уже существующих изображений с целью определить шаблоны, которые можно было бы использовать для улучшения или, возможно, "понимания" содержания изображения. Короче говоря, 2D-графика связана с созданием изображений, а обработка изображений — с анализом уже существующих изображений.



Основные положения для запоминания

- **Компьютерные вычисления позволяют творчески исследовать как реальные, так и виртуальные объекты восприятия.**

В отличие от задач преобразования двухмерных фигур в изображения, что имеет место в 2D-графике, область **3D-графики** имеет дело с задачами преобразования в изображения трехмерных фигур. Процесс заключается в создании цифровых версий трехмерных сцен с последующей имитацией фотографического процесса для получения двухмерных цифровых изображений этих сцен. В целом задача аналогична традиционной фотографии, за исключением того, что та сцена, которая "фотографируется" с использованием методов трехмерной графики, не существует как физическая реальность; вместо этого она представлена как некоторый набор исходных данных и алгоритмов. Таким образом, трехмерная графика предполагает "фотографирование" виртуальных миров, в то время как традиционная фотография предполагает фотографирование реального мира.

Важно отметить, что создание изображения с использованием методов трехмерной графики предусматривает два отдельных этапа. Первым является создание, кодирование, хранение и манипулирование фотографируемой сценой. Второй — это процесс получения изображения. Первый этап — это творческий, художественный процесс, тогда как второй предполагает лишь проведение значительного объема компьютерных вычислений. Это те темы, которые будут рассматриваться в следующих четырех разделах данной главы.

Тот факт, что 3D-графика создает "фотографии" виртуальных сцен, делает ее идеальной для использования в интерактивных видеоиграх и в производстве анимационных фильмов, где в противном случае реальность накладывала бы на возможные действия свои ограничения. Будет справедливо отметить, что мощные вычислительные инструменты создания новых визуализаций, музыки и эффектов меняют даже тип историй, которые пытается донести до зрителя индустрия развлечений. Интерактивная видеоигра состоит из закодированной трехмерной виртуальной среды, с которой взаимодействует игрок. Изображения, которые видит игрок, создаются с помощью технологии 3D-графики. Анимированные движущиеся изображения создаются аналогичным образом, за исключением того, что аниматор-человек взаимодействует с виртуальной средой, а не с конечным зрителем. Продукт, в конечном итоге распространяемый



Основные положения для запоминания

- **Создание цифровых эффектов, изображений, аудио, видео и анимации преобразовало всю индустрию развлечений.**
- **Достижения в области вычислительной техники создали и многократно усилили творческий потенциал в других областях.**

среди широкой публики, представляет собой последовательность двухмерных изображений, как это было определено выпускающим фильм режиссером или продюсером.

Подробно использование трехмерной графики в анимации будет рассматриваться в разделе 10.6. А пока, в завершение этого раздела, попробуем представить, куда могут привести нас новые достижения, которых можно ожидать по мере развития технологии 3D-графики. Сегодня движущиеся изображения представляются и распространяются в виде последовательности двухмерных изображений. Хотя в проекторах, отображающих эту информацию, уже совершен переход от аналоговых устройств с кинопленкой к цифровым технологиям с использованием DVD-плееров и плоскопанельных дисплеев, они все еще имеют дело только с двухмерными представлениями.

А теперь попробуем представить, как все это сможет измениться, когда наши возможности создания и управления реалистичными трехмерными виртуальными мирами расширятся. Вместо того чтобы "фотографировать" эти виртуальные миры и распространять кинофильмы в форме последовательностей двухмерных изображений, можно будет распространять сами виртуальные миры. В результате потенциальный зритель получит доступ к *набору* кинофильмов, а не к единственному кинофильму. Этот трехмерный набор будет просматриваться им с помощью "трехмерного графического проектора", — примерно так, как видеоигры отображаются с помощью "игровых приставок" специального назначения. Сначала можно будет посмотреть "предложенный сюжет", т.е. отображение фильма в том виде, в каком его представлял себе режиссер или продюсер. Однако у зрителя также будет возможность взаимодействия с виртуальным содержанием фильма способом, напоминающим видеоигру, что позволит ему создавать собственные сценарии развития сюжета, фактически получая новые версии фильма. Как видите, возможности здесь весьма широки, особенно если учесть потенциал разрабатываемых в настоящее время трехмерных интерфейсов "человек–машина".

10.1. Вопросы и упражнения

1. Назовите различия, существующие между обработкой изображений, 2D-графикой и 3D-графикой.
2. Чем 3D-графика отличается от традиционной фотографии?
3. Каковы два основных этапа создания "фотографии" с использованием 3D-графики?

10.2. Что такое 3D-графика

Давайте начнем изучение 3D-графики с рассмотрения процесса создания и отображения трехмерных изображений в целом — процесса, состоящего из трех этапов: моделирования, рендеринга и отображения. Этап моделирования (который подробно обсуждается в разделе 10.3) аналогичен проектированию и созданию декораций в традиционной индустрии кино, за исключением того, что сцена трехмерной графики "построена" из данных и алгоритмов, закодированных в цифровой форме. Таким образом, сцена, созданная в контексте компьютерной графики, в реальности может никогда не существовать.

Следующим этапом является создание двухмерного изображения сцены путем вычисления того, как объекты на сцене будут выглядеть на фотографии, сделанной камерой в указанной позиции. Данный этап называется рендерингом — это предмет обсуждения в разделах 10.4 и 10.5. Рендеринг включает в себя применение математического аппарата аналитической геометрии для вычисления проекции объектов в сцене на плоскую поверхность, известную как **проекционная** (или картинная) **плоскость**, аналогично камере, проецирующей сцену на пленку (рис. 10.1). Применяемый тип проекции — это **перспективная проекция**; это означает, что все объекты проецируются с помощью прямых линий, называемых **проекторами**, которые выходят из общей точки, называемой **центром проекции** или **точкой зрения**. (Это отличает данную проекцию от **параллельной проекции**, в которой все проекторы параллельны. Центральная проекция создает изображение, похожее на то, которое видит человеческий глаз, тогда как параллельная проекция создает "истинный" профиль объекта, который может быть полезен при создании технических чертежей.)

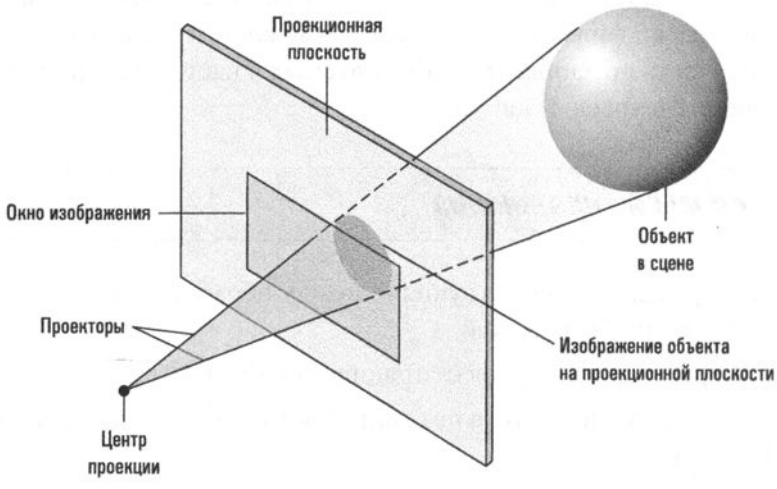


Рис. 10.1. Парадигма 3D-графики

Ту часть плоскости проекции, которая определяет границы получаемого изображения, называют **окном изображения**. Оно соответствует прямоугольнику, который отображается в видоискателе большинства обычных фотокамер камер для указания границ будущего изображения. И действительно, видоискатель большинства обычных фотокамер позволяет видеть в плоскости проецирования камеры чуть больше пространства, чем умещается в ее видовом окне. Так, в видоискатель вы можете увидеть верхнюю часть головы фотографируемого человека, но если эта верхняя часть головы не будет находиться в рамке границ изображения, она будет отсутствовать на фотографии.

Как только будет выявлена та часть сцены, которая проецируется в окно изображения, вычисляется внешний вид каждого пикселя конечного изображения. Этот попиксельный процесс может быть весьма сложным в вычислительном отношении, поскольку он требует определения того, как объекты в сцене взаимодействуют со светом — твердая блестящая поверхность при ярком освещении должна выглядеть иначе, чем мягкая прозрачная поверхность при непрямом свете. В свою очередь, процесс рендеринга сильно зависит от многих других областей знания, включая материаловедение и физику. Более того, определение внешнего вида одного объекта часто требует принимать во внимание наличие в сцене других объектов. Так, объект может находиться в тени другого объекта, или же объект может иметь зеркальную поверхность и его внешний вид по существу будет определяться внешним видом других объектов.

Как только внешний вид каждого пикселя будет определен, все полученные результаты сохраняются вместе в области хранения, называемой **буфером кадра**, — как представление изображения в виде битовой карты. Этот буфер может быть областью основной памяти или же это может быть блок памяти особого назначения в случае аппаратуры, разработанной специально для поддержки графических приложений.

Наконец, изображение, хранящееся в буфере кадров, либо отображается для просмотра, либо передается в область памяти для более длительного хранения с целью последующего отображения. Если изображение создается для использования в движущемся изображении, оно может быть сохранено и, возможно, даже изменено до получения его окончательного представления. Однако в интерактивной видеоигре или в имитаторе полета изображения должны отображаться в том виде, в котором они создаются в режиме реального времени, что часто ограничивает качество создаваемых изображений. Вот почему качество графики полнофункциональных анимационных фильмов, распространяемых киноиндустрией, существенно превосходит качество современных интерактивных видеоигр.

В завершение нашего краткого знакомства с трехмерной графикой проанализируем компьютерную систему типичной видеоигры. Сама игра, по сути,

является закодированным виртуальным миром вместе с программным обеспечением, позволяющим игроку так или иначе манипулировать этим миром. Когда игрок воздействует на этот мир, игровая система раз за разом визуализирует сцену и сохраняет полученное изображение в буфере изображений. Чтобы преодолеть ограничения реального времени, бо́льшая часть всего процесса рендеринга осуществляется с использованием специального оборудования. В действительности именно наличие подобного оборудования является главным различием между игровой системой и обычным персональным компьютером. Наконец, устройство отображения в игровой системе выводит на экран содержимое буфера кадров, создавая у игрока иллюзию изменения сцены.

10.2. Вопросы и упражнения

1. Назовите три этапа создания изображения с использованием 3D-графики.
2. В чем состоит отличие проекционной плоскости от окна изображения?
3. Что такое "буфер кадра"?

10.3. Моделирование

Реализация проекта компьютерной 3D-графики начинается во многом так же, как обычная театральная постановка: необходимо спроектировать и изготовить декорации, подобрать или изготовить реквизит. В терминологии компьютерной графики определенный набор декораций называется **сценой**, а реквизит — **объектами**. Имейте в виду, что трехмерная графическая сцена является виртуальной, поскольку состоит из объектов, которые "построены" как модели, представленные цифровым кодом, а не как материальные физические структуры.

В этом разделе будут рассмотрены темы, связанные с "конструированием" объектов и сцен. Начнем с вопросов моделирования отдельных объектов, а в заключение рассмотрим задачу связывания этих объектов между собой с целью формирования сцены.

Моделирование отдельных объектов

На театральной сцене степень, в которой декорация соответствует реальности, зависит от того, как она будет использоваться в спектакле. Здесь может не понадобиться целый автомобиль, телефон не обязательно должен быть

функциональным, а фоновый пейзаж может быть нарисован на плоском фоне. Аналогичным образом в случае компьютерной графики степень, в которой программная модель объекта будет точно отражать его истинные свойства, зависит от требований ситуации. Для моделирования объектов на переднем плане потребуется больше деталей, чем для объектов заднего плана. Больше деталей для объектов можно использовать и в тех случаях, когда нет строгих ограничений режима реального времени. Однако, поскольку высокая детализация связана с дополнительными затратами как в отношении объема необходимых вычислений, так в отношении представления данных (объем памяти), имеет место сильный стимул абстрагироваться в моделях от ненужных деталей. Компьютерные модели объектов и моделирование их взаимодействий должны начинаться с абстракций, которые будут существенно менее сложными, чем их аналоги в реальном мире.



Основные положения для запоминания

- **Модели и имитации — это упрощенное представление более сложных объектов или явлений.**

Следовательно, одни модели объектов могут быть относительно простыми, тогда как другие могут оказаться чрезвычайно сложными. Как правило, более точные модели обеспечивают более высокое качество изображения, но одновременно увеличивают время рендеринга. В свою очередь, большая часть проводимых сейчас исследований в области компьютерной графики направлена на разработку методов построения высокодетализированных и одновременно эффективных объектных моделей. Некоторые из этих исследований посвящены разработке моделей, которые могут обеспечить различные уровни детализации в зависимости от конечной роли объекта в сцене, — в результате получается единая объектная модель, которая может использоваться в изменяющейся среде.

Информация, необходимая для описания объекта, включает форму объекта, а также дополнительные свойства, такие как характеристики поверхности, определяющие, как объект будет взаимодействовать со светом. А сейчас давайте рассмотрим задачу моделирования формы.

Форма

Форма объекта в 3D-графике обычно описывается в виде набора небольших плоских поверхностей, называемых **плоскими гранями**, каждая из которых имеет вид многоугольника. В совокупности весь набор этих многоугольников

образует **полигональную сетку** (или **многоугольный каркас**), которая приблизительно соответствует форме описываемого объекта, как показано на рис. 10.2. Используя небольшие плоские многоугольники, аппроксимацию формы объекта можно сделать настолько точно, насколько это необходимо.

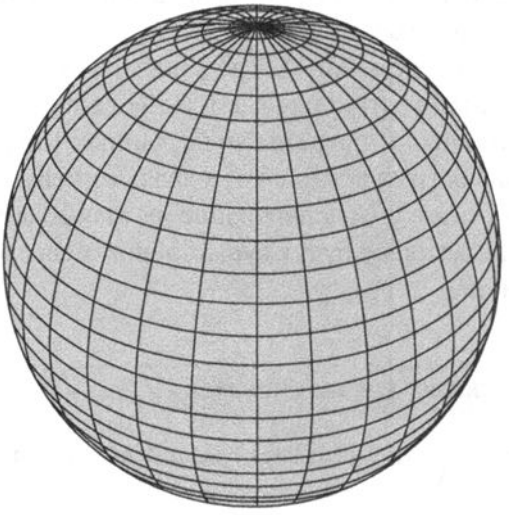


Рис. 10.2. Полигональная сетка для сферы

Плоские грани в полигональной сетке часто выбираются в виде треугольников, поскольку каждый треугольник может быть представлен всего тремя вершинами, что является минимальным количеством точек, необходимых для идентификации плоской поверхности в трехмерном пространстве. В любом случае полигональная сетка представляется как совокупность вершин ее плоских участков.

Представление объекта в виде полигональной сетки может быть получено различными способами. Один из них — начать с точного геометрического описания желаемой формы и использовать это описание для построения полигональной сетки. Например, аналитическая геометрия говорит, что сфера (с центром в начале координат) радиуса r описывается следующим уравнением.

$$r^2 = x^2 + y^2 + z^2$$

Исходя из этой формулы, можно записать уравнения для линий широты и долготы на поверхности сферы и определить точки, в которых эти линии пересекаются, а затем использовать найденные точки пересечения в качестве вершин многоугольной сетки. Подобные методы могут быть применены и к другим традиционным геометрическим фигурам, — именно по этой причине персонажи в недорогих компьютерных анимациях очень часто оказываются составленными из таких простейших компонентов, как сферы, цилиндры и конусы.

Более произвольные формы могут быть описаны более сложными аналитическими методами. Один из них основан на использовании **кривых Безье** (названных в честь Пьера Безье, который разработал эту концепцию в начале 1970-х годов, когда работал на должности инженера в автомобильной компании Renault). Этот метод позволяет изгибать отрезок прямой линии в трехмерном пространстве, манипулируя лишь немногим точками, называемыми *контрольными точками*, — две из них представляют концы сегмента кривой, а остальные определяют, как эта кривая изгибается. В качестве примера на рис. 10.3 показана кривая, определяемая четырьмя контрольными точками. Обратите внимание: кажется, что кривая как бы тянется к двум контрольным точкам, которые не определяют концы сегмента. Перемещая эти точки, можно тем или иным образом изгибать кривую, придавая ей различную форму. (Возможно, вы уже сталкивались с подобными методами при построении изогнутых линий в программных пакетах для рисования, таких как Microsoft Paint.) Хотя здесь мы не будем подробно останавливаться на этой теме, метод Безье для описания кривых может быть расширен для описания трехмерных поверхностей, известных как **поверхности Безье**. В свою очередь, поверхности Безье показали себя как эффективный первый шаг в процессе получения полигональных сеток для сложных поверхностей.

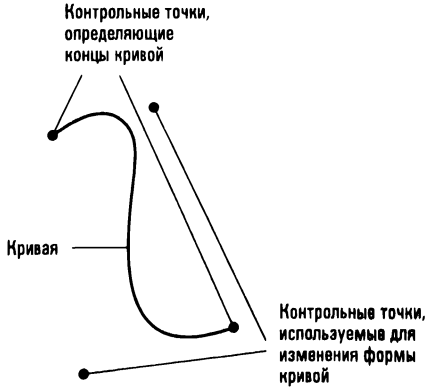


Рис. 10.3. Кривая Безье

У вас может возникнуть вопрос, почему необходимо преобразовать точное описание формы, такое как краткая формула сферы или формула, описывающая поверхность Безье, в приближенное представление этой формы с использованием полигональной сетки. Ответ на него состоит в том, что представление формы всех объектов с помощью полигональных сеток устанавливает единый подход к процессу рендеринга, — это важная особенность, позволяющая более эффективно воспроизводить целые сцены. В результате, хотя геометрические формулы обеспечивают точное описание форм, они служат лишь инструментами для построения полигональных сеток.

Другим способом построения полигональной сетки является создание ее методом "грубой силы". Этот подход популярен в тех случаях, когда требуемая форма не поддается представлению с помощью элегантных математических методов. Процедура состоит в том, что сначала строится физическая модель требуемого объекта, а затем местоположение определенных точек на поверхности этой модели определяется и записывается посредством прикосновения к ним специальной ручкой, фиксирующей положение точки в трехмерном пространстве. Этот процесс известен как **оцифровка**. Полученный в результате набор точек далее можно будет использовать в качестве вершин при построении полигональной сетки, описывающей данную форму.

К сожалению, некоторые формы настолько сложны, что получить реалистичные модели с помощью геометрического моделирования или ручной оцифровки оказывается просто невозможно. Как типичные примеры можно привести растительные структуры, такие как деревья, сильно пересеченную местность, такую как горные цепи, или газообразные субстанции, такие как облака, дым или языки пламени. В подобных случаях полигональные сетки можно получить, только написав программу, которая будет автоматически генерировать желаемую форму. В своей совокупности такие программы известны как **процедурные модели**. Иначе говоря, процедурная модель — это программный модуль, который использует некоторый специальный алгоритм для генерации желаемой структуры.

Вот конкретный пример: процедурную модель можно использовать для генерации полигональной сетки горного хребта посредством выполнения следующих манипуляций. Начинаем с одного треугольника и определяем середины его сторон (рис. 10.4, *а*). Затем соединяем эти средние точки и получаем в общей сложности четыре меньших треугольника (рис. 10.4, *б*). Теперь, сохраняя положение вершин исходного треугольника неизменным, перемещаем средние точки его сторон в трехмерном пространстве (при этом стороны треугольника могут растягиваться или сжиматься), искажая исходную форму треугольника (рис. 10.4, *в*). Далее этот процесс повторяется с каждым из получившихся меньших треугольников (рис. 10.4, *г*) и не прекращается до тех пор, пока не будет достигнута желаемая степень детализации.

Процедурные модели предоставляют эффективные средства для создания множеств сложных объектов, которые схожи между собой, но в то же время каждый из них остается уникальным. Например, процедурную модель можно использовать для построения множества реалистичных объектов деревьев — все они будут схожи, но каждое со своей структурой ветвей. Один из подходов к созданию таких моделей деревьев состоит в применении определенных правил ветвления в процессе "выращивания" объектов деревьев подобно тому, как синтаксический анализатор (см. раздел 6.4) создает дерево анализа на

основании правил грамматики. В действительности набор правил ветвления, используемых в подобных случаях, также часто называют *грамматикой*. Одна грамматика может быть предназначена для "выращивания" сосен, тогда как другая — для "выращивания" дубов.

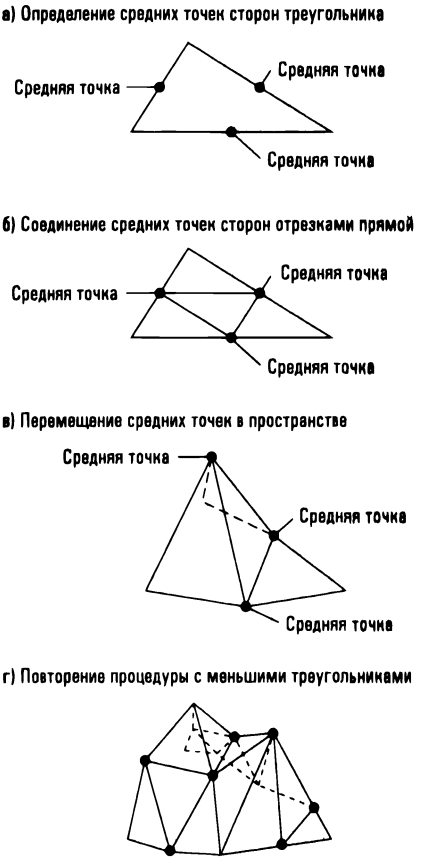


Рис. 10.4. Создание полигональной сетки горного хребта

Другой подход к построению процедурных моделей состоит в том, чтобы моделировать базовую структуру объекта как большой набор отдельных частиц. Такие модели называют **системами частиц**. Как правило, к системам частиц применяют некоторые заранее определенные правила перемещения отдельных частиц в системе (возможно, способом, напоминающим молекулярные взаимодействия) для получения желаемой общей формы объекта. Например, системы частиц могут использоваться для анимации процесса выплескивания воды, как будет показано ниже в этой главе при обсуждении анимации. (Представьте себе ведро воды, смоделированное как ведро стеклянных шариков. По мере того как ведро взлетает и опускается, шарики выпадают из него в разные стороны,

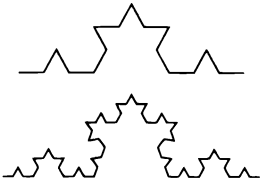
имитируя движение воды.) Другими примерами использования систем частиц является моделирование колеблющихся языков пламени, движущихся облаков или массовых сцен с множеством персонажей.

Фракталы

Построение горного хребта с помощью процедурной модели, как описывалось в этом разделе (см. рис. 10.5), — пример того, какая важная роль отводится фракталам в трехмерной графике. Говоря научным языком, **фрактал** — это геометрический объект, "размерность Хаусдорфа которого больше, чем его топологическая размерность". Если говорить обычным языком, то это означает, что конечный результат формируется путем компоновки с уже имеющейся частью новых копий исходного объекта все более и более уменьшающихся размеров. Фракталы обычно формируются с использованием рекурсивного процесса, в ходе которого каждая активация в рекурсии "собирает воедино" дополнительные (но меньшие по размеру) копии исходного шаблона, использованного для построения фрактала. Полученный фрактал самоподобен в том смысле, что каждая часть при увеличении выглядит как его копия. Традиционным примером фрактала является снежинка фон Коха, которая образуется путем многократной замены отрезков прямых в структуре вида



меньшими версиями той же структуры. Это приводит к последовательности уточнений, которая выглядит следующим образом:



В 3D-графике фракталы часто являются основой построения процедурных моделей. И действительно, они неоднократно использовались для создания реалистичных изображений горных цепей, растительности, облаков или дыма.

Результатом построения процедурной модели обычно является полигональная сетка, которая аппроксимирует форму требуемого объекта. В некоторых случаях — как, например, при создании горного хребта из треугольников — сетка является естественным следствием процесса генерации. В других случаях, таких как выращивание дерева на основании правил ветвления, построение сетки может быть дополнительным, завершающим этапом. Так, в случае систем частиц частицы, которые расположены на внешнем краю системы,

естественным образом являются кандидатами в вершины конечной полигональной сетки системы в целом.

Насколько точна будет сетка, сгенерированная процедурной моделью, в принципе, зависит от ситуации. Процедурное моделирование дерева на заднем плане сцены может привести к созданию упрощенной контурной сетки, которая будет отражать только базовую форму дерева, тогда как процедурное моделирование дерева на переднем плане может иметь результатом полигональную сетку, в которой будут различимы отдельные ветви и даже листья.

Характеристики поверхности

Модель, состоящая только из полигональной сетки, определяет лишь форму объекта. Большинство систем рендеринга по запросу пользователя непосредственно в процессе рендеринга способны обогащать такие модели определенными деталями с целью имитации различных характеристик их поверхности. Например, применяя различные методы затенения (эта тема обсуждается в разделе 10.4), пользователь может указать, что полигональная сетка шара на проекционной плоскости будет отображаться как красный гладкий шар или же как зеленый шероховатый шар. В некоторых случаях такая гибкость весьма желательна. Однако в ситуациях, требующих точной визуализации исходного объекта, в модель должна быть включена более конкретная информация о данном объекте, чтобы система рендеринга знала, что она должна делать.

Существует множество способов кодирования информации об объекте в дополнение к его форме. Например, наряду с координатами каждой вершины полигональной сетки можно кодировать цвет исходного объекта в этой точке. Позднее эта информация может использоваться в процессе рендеринга для воссоздания внешнего вида исходного объекта.

В других случаях требуемые цветовые шаблоны могут быть связаны с поверхностью объекта с помощью процесса, известного как **отображение текстуры**. Отображение текстуры аналогично процессу наклейки на стену обоев — в том смысле, что оно связывает с поверхностью объекта некоторое заранее определенное изображение. Оно может быть цифровой фотографией, работой художника или изображением, сгенерированным компьютером. Традиционные текстурные изображения включают кирпичные стены, деревянные поверхности с годичными кольцами и сучками и даже мраморные фасады.

Например, предположим, что нам необходимо смоделировать каменную стену. Форму этой стены можно представить с помощью простой полигональной сетки, представляющей длинное прямоугольное тело. Затем с этой сеткой можно было бы связать цветное двухмерное изображение каменной кладки. Тогда в процессе рендеринга это изображение будет автоматически применено

к прямоугольному объекту стены так, чтобы в конечном счете он выглядел, как каменная стена. Если говорить точнее, то каждый раз, когда по ходу рендеринга потребуется определить внешний вид точки на стене, программа просто использует внешний вид соответствующей точки на изображении каменной кладки.

Процедура отображения текстуры работает лучше всего, когда применяется к относительно плоским поверхностям. Если же текстурное изображение приходится сильно искажать, полученный результат может выглядеть слишком искусственно. Представьте себе результат попытки наклеить плоские обои на футбольный мяч! Кроме того, если текстурное изображение полностью оборачивается вокруг изогнутой поверхности, в месте стыка рисунок может не совпадать, образуя заметный шов. Как бы там ни было, метод наложения текстуры в конечном счете оказался весьма эффективным средством имитации самых различных поверхностей и поэтому широко используется в ситуациях, моделируемых в режиме реального времени, и самый яркий пример тому — интерактивные видеоигры.

В поисках реализма

Построение объектных моделей, позволяющих получить реалистичные изображения, — важная тема непрекращающихся научных исследований. Особый интерес представляют материалы, связанные с живыми персонажами, такими как кожа, волосы, мех или перья. Большая часть подобных исследований относится к конкретным материалам и ставит основной целью разработку методов их моделирования и рендеринга. Например, чтобы получить реалистичные модели человеческой кожи, некоторые исследователи учитывали в своих разработках степень, в которой свет проникает через верхние и нижние слои кожи, а также, как состав и структура этих слоев влияют на ее внешний вид.

Другой пример связан с моделированием человеческих волос. Если волосы видны на достаточном расстоянии, то вполне могут подойти и более традиционные методы моделирования. Но для крупных планов реалистично выглядящая имитация волос может оказаться непростой задачей. Возникающие при этом проблемы включают свойства прозрачности и текстурной глубины, особенности укладки и то, как волосы реагируют на внешние силы, такие как ветер. Чтобы преодолеть эти проблемы, некоторые приложения прибегают к моделированию отдельных прядей волос, — очень непростая задача, поскольку общее количество волос на голове человека может достигать порядка 100 тысяч. Однако еще более поразительным является тот факт, что некоторые исследователи создали модели волос, позволяющие учитывать масштабирование текстуры, вариации цвета и даже механическую динамику для каждой отдельной пряди.

Еще один пример, предполагающий принятие во внимание множества деталей, — это моделирование одежды. В этом случае мелкие детали плетения нитей необходимо использовать для получения надлежащих текстурных различий между тканями разных типов, такими как саржа и сатин. В этом случае характерные особенности нитей комбинируются с различными шаблонами их плетения, что позволяет создать, например, модели трикотажного полотна, позволяющие достичь исключительной реалистичности изображений. Кроме того, сведения из области физики и текстильного машиностроения потребовалось применить даже к отдельным нитям тканей, что было необходимо для вычисления реалистичной формы складок тканей с учетом таких их аспектов, как растяжение нитей или искажение исходного узора плетения.

Создание реалистичных изображений продолжает оставаться активной областью исследований, которая, как уже говорилось, включает методы и для процессов моделирования, и для процессов рендеринга. Как правило, по мере достижения прогресса новые методы сначала включаются в приложения, на которые не распространяются ограничения режима реального времени, такие как графическое программное обеспечение киностудий, в котором допустима значительная задержка во времени между процессами моделирования и рендеринга, и окончательным представлением изображений. По мере того как эти новые методы совершенствуются и оптимизируются, они находят свое применение и в приложениях реального времени, за счет чего качество графики в этих приложения также улучшается. Действительно реалистичное взаимодействие в реальном времени в виртуальных мирах может стать явью уже в не очень далеком будущем.

Моделирование целых сцен

После того как объекты в сцене были надлежащим образом описаны и закодированы в цифровой форме, каждому из них назначаются местоположение, размер и ориентация в пределах сцены. Вся эта информация затем связывается для формирования структуры данных, называемой **графом сцены**. Кроме того, граф сцены содержит ссылки на специальные объекты, представляющие источники света, а также некоторый объект, представляющий видеокамеру. В этом объекте записываются местоположение, ориентация и фокусные свойства камеры, записывающей создаваемый видеофайл.

Следовательно, граф сцены аналогичен обстановке и оформлению студии в традиционной фотографии. Он включает камеру, источники света, реквизит и фоновый пейзаж — все, что необходимо для создания требуемой фотографии по щелчку затвора, — и это все размещено по своим местам. Различие здесь состоит лишь в том, что традиционная фотография предполагает использование

физических объектов, тогда как граф сцены оперирует цифровым представлением изображаемых объектов. Короче говоря, граф сцены полностью описывает виртуальный мир.

Моделирование волокон

Достижения в компьютерном моделировании волокон ткани, меха и волос были обусловлены спросом на реалистичность компьютерных изображений для развлекательных целей. Обратите внимание, например, на детали тканей, из которых изготовлены шляпы и одежда смурфиков на приведенном ниже изображении, а также на реалистичность светлых волос Смурфетты. Такое внимание к реалистичным деталям цифровых образов стало обычным явлением, даже когда оно применяется к явно вымышленным персонажам мультфильмов с бровями, которые простираются над шляпами.





СМУРФИКИ: ЗАТЕРЯННАЯ ДЕРЕВНЯ (2017), Sony Pictures Animation.

Пышные детали Запретного леса вокруг смурфиков потребовали создания вычислительно насыщенного графа сцены. В некоторых сценах видно более 60 деревьев, а также много других видов растительности. Одни модели деревьев состояли примерно из 40 тысяч листьев, по 60–90 граней на лист. Другие элементы растительности, такие как вездесущая лоза ползучей Дженни, были процедурно сгенерированы как травяной покров.

Расположение камеры внутри сцены имеет большое значение и влечет за собой множество следствий. Как упоминалось ранее, степень детализации, с которой моделируются объекты, зависит от местоположения этих объектов в сцене. Объекты переднего плана требуют большего количества деталей, чем объекты фона, а само разграничение между передним планом и фоном определяется положением камеры. Если виртуальная сцена используется в контексте, подобном декорациям на театральной сцене, то элементы переднего плана и фона четко определены и объектные модели могут быть построены соответствующим

образом. Однако если контекст требует, чтобы положение камеры изменялось по ходу получения серии изображений, то детальность представления объектов в моделях может потребовать корректировки между созданием отдельных "фотографий". Это область проведения текущих исследований. Уже можно представить себе сцену, состоящую из "интеллектуальных" моделей, уточняющих свои полигональные сетки и другие функции по мере перемещения камеры внутри сцены. Уровень абстракции, требуемый для модели каждого объекта, зависит как от базового объекта, так и от его контекста в сцене.



Основные положения для запоминания

- **Модели могут использовать различные абстракции или уровни абстракции в зависимости от представляемых ими объектов или явлений.**

Интересный пример сценария с движущейся камерой встречается в таких системах виртуальной реальности, в которых человеку разрешается испытывать ощущение движения в воображаемом трехмерном мире. Воображаемый мир представлен графом сцены, и человек смотрит на эту среду с помощью камеры, которая движется внутри сцены в соответствии с движениями человека. В действительности для обеспечения восприятия глубины трех измерений используются две камеры: одна представляет правый глаз человека, а другая — левый. При выводе изображения, полученного каждой из камер, перед соответствующим глазом человек получает иллюзию нахождения в трехмерной сцене, а когда к визуальному опыту добавляются звуковые и тактильные ощущения, иллюзия может стать вполне реалистичной.

В заключение отметим, что построение графа сцены занимает центральное положение в процессе создания 3D-графики. Поскольку он содержит всю информацию, необходимую для создания окончательного изображения, его завершение знаменует собой завершение процесса художественного моделирования и начало вычислительно интенсивного процесса рендеринга изображения. И действительно, после завершения построения графа сцены графическая задача становится задачей вычисления проекций, определения деталей поверхности в определенных точках и моделирования эффектов света, т.е. набора задач, решение которых в значительной степени не зависит от выбора конкретного приложения.

3D-телевидение

Существует несколько технологий для создания 3D-изображений в контексте телевидения, но все они основаны на одном и том же стереоскопическом визуальном эффекте: два слегка различающихся изображения попадают в левый и правый глаза зрителя, что воспринимается его мозгом как глубина. Самые недорогие механизмы для реализации этого эффекта требуют специальных очков с линзами фильтров. Более старые цветные линзы (использовавшиеся в кино в 1950-х годах) или более современные поляризованные линзы отфильтровывают различные аспекты одного изображения с экрана, в результате чего в каждый из глаз попадают разные изображения. Более дорогая технология предполагает использование "активных" очков, которые поочередно закрывают левый и правый объективы синхронно с 3D-телевизором, на экране которого происходит быстрое переключение между левым и правым изображениями. Наконец, разрабатываются 3D-телевизоры, которые не будут требовать специальных очков или головных уборов. Они используют сложные матрицы фильтров или увеличительные линзы на поверхности экрана, чтобы проецировать левое и правое изображения под немного разными углами относительно головы зрителя, а это означает, что его левый и правый глаза будут видеть разные изображения.

10.3. Вопросы и упражнения

1. Ниже приведены четыре точки (закодированные с использованием традиционной прямоугольной системы координат), которые представляют вершины плоской грани. Опишите форму этой грани. (Для тех, кто не имел дела с аналитической геометрией, можно дать следующее пояснение. Каждая тройка чисел указывает, как можно добраться до данной точки, начиная с угла комнаты. Первое число говорит о том, как далеко необходимо пройти вдоль плинтуса между полом и стеной справа от угла. Второе число говорит о том, как далеко от достигнутой точки следует пройти вглубь комнаты в направлении, параллельном стене слева от вас. Наконец, третье число говорит о том, насколько высоко следует подняться вверх от достигнутой на полу точки. Если какое-то из чисел отрицательное, вам придется притворяться, что вы призрак и способны проходить в обратном направлении сквозь стены и полы.)
   (0, 0, 0) (0, 1, 1) (0, 2, 1) (0, 1, 0)
2. Что такое *процедурная модель*?

3. Перечислите некоторые объекты, которые могут присутствовать в графе сцены, используемом для создания изображения в парке.
4. Почему формы всегда представляются полигональными сетками, хотя в некоторых случаях они могут быть более точно представлены геометрическими уравнениями?
5. Что такое *наложение текстуры*?

10.4. Рендеринг

Настало время рассмотреть процесс рендеринга, который включает в себя определение того, как будут выглядеть объекты в графе сцены при их проецировании на проекционную плоскость. Есть несколько способов решения задачи рендеринга. В данном разделе рассматривается традиционный подход, который сегодня используется в большинстве популярных графических систем, представленных на "потребительском рынке" видеоигр, домашних компьютеров и т.д. В следующем разделе рассматриваются две альтернативы этому подходу.

Мы начнем обсуждение с рассмотрения некоторых базовых аспектов взаимодействия света и объектов. В конечном счете внешний вид объекта определяется светом, поступающим от этого объекта, а отсюда следует, что определение внешнего вида объекта в конечном итоге сводится к задаче имитации поведения света.

Взаимодействие света с поверхностью

В зависимости от свойств материала свет, падающий на поверхность объекта, может поглощаться, отражаться от его поверхности как отраженный свет или проходить сквозь его поверхность (с изменением направления) как преломленный свет.

Отражение

Давайте рассмотрим, что происходит с лучом света, который отражается от плоской непрозрачной поверхности. Луч света, двигаясь по прямой линии, падает на поверхность под углом, называемым **углом падения**. Угол, под которым луч отражается от поверхности, всегда равен углу падения. Как показано на рис. 10.5, оба эти угла измеряются относительно линии, перпендикулярной (или **нормальной**) поверхности. (Линия, перпендикулярная поверхности, часто называется просто "нормалью", как в фразе "Угол падения измеряется

относительно нормали к поверхности".) Падающий луч, отраженный луч и нормаль всегда лежат в одной плоскости.

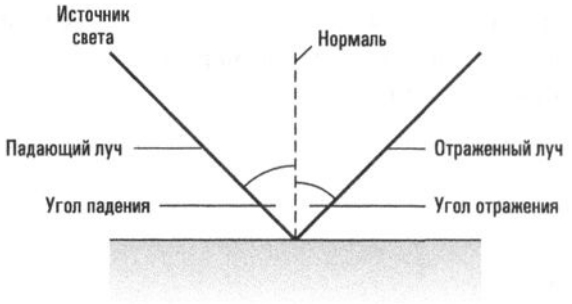


Рис. 10.5. Отражение света от непрозрачной поверхности

Если поверхность является ровной, параллельные световые лучи (например, лучи, приходящие от одного и того же удаленного источника света), падающие на поверхность в одной и той же области, будут отражаться в точности в одном и том же направлении и уходить от объекта как параллельные лучи. Такой отраженный свет называется **зеркальным светом**. Обратите внимание, что зеркальный свет можно наблюдать только в том случае, когда ориентация поверхности и источника света такова, что вызывает отражение света в направлении зрителя. В результате зеркальный свет обычно выглядит, как яркие блики на поверхности объекта. Более того, поскольку зеркальный свет имеет минимальный контакт с отражающей поверхностью, он обычно сохраняет цвет, свойственный исходному источнику света.

Однако в действительности поверхности редко бывают идеально гладкими, и поэтому многие световые лучи могут попадать на поверхность в точках, ориентация которых отличается от ориентации всей поверхности в целом. Кроме того, световые лучи часто проникают сквозь верхнюю границу поверхности и многократно переотражаются между поверхностными частицами, прежде чем окончательно покинут объект в качестве отраженного света. В результате многие лучи будут рассеиваться в разных направлениях. Такой отраженный свет называется **диффузным светом**. В отличие от зеркального света, диффузный свет виден в широком диапазоне направлений. И поскольку ему свойственна тенденция иметь более длительный контакт с поверхностью, диффузный свет более чувствителен к поглощающим свойствам материала и, следовательно, имеет тенденцию окрашиваться в цвет отражающего объекта.

На рис. 10.6 представлен шар, освещенный одним источником света. Яркая подсветка на шаре создается зеркальным светом. Остальная часть полушария, обращенная к источнику света, видна благодаря диффузному свету. Обратите внимание, что полусфера, обращенная в противоположную от первичного

источника света сторону, не видна в лучах этого источника из-за отражения всего его света другими ее частями. Способность видеть эту часть шара обусловлена окружающим светом, который является случайным ослабленным светом, или **рассеянным светом**, не связанным с каким-либо конкретным источником или направлением. Части поверхностей, освещаемых рассеянным светом, часто выглядят как имеющие однородный темный цвет.

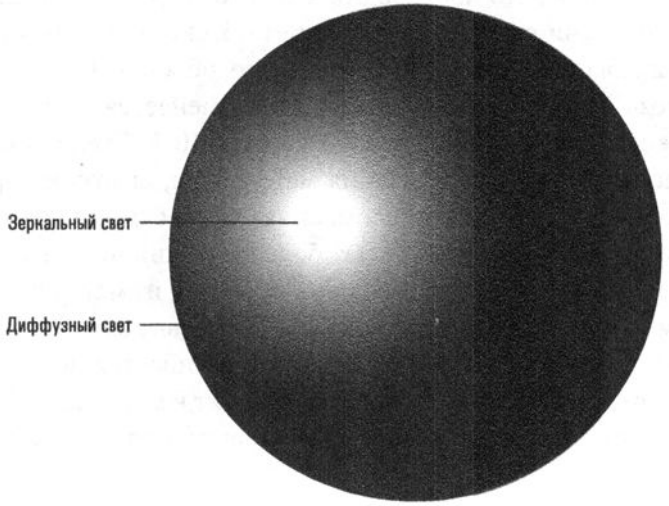


Рис. 10.6. Зеркальный свет и диффузный свет

Большинство поверхностей отражают свет как в виде зеркального, так и в виде диффузного света. Характеристики поверхности определяют пропорции каждого. Гладкие поверхности выглядят блестящими, потому что отражают больше зеркального света, чем диффузного. Грубые поверхности кажутся тусклым, поскольку отражают больше диффузного света, чем зеркального. Кроме того, из-за особых свойств мельчайших элементов некоторых поверхностей соотношение зеркального и рассеянного света может изменяться в зависимости от направления падающего света. В результате свет, падающий на такую поверхность из одного направления, может отражаться главным образом как зеркальный, тогда как свет, падающий на эту поверхность с другого направления, будет отражаться главным образом в виде диффузного света. Как следствие внешний вид такой поверхности при ее вращении будет меняться от блестящего до тусклого. Такие поверхности называются **анизотропными поверхностями**, в отличие от **изотропных поверхностей**, отражательные свойства которых симметричны и не зависят от направления падающего луча света. Примеры анизотропных поверхностей можно найти среди тканей, например ворс сатина меняет внешний вид ткани в зависимости от ее ориентации относительно источника света. Другим характерным примером является травянистая

поверхность футбольных полей, на которых травяной покров (обычно подстригаемый специальными косилками) создает анизотропные визуальные эффекты, представляющие собой узор из светлых и темных полос.

Преломление

А теперь рассмотрим, что происходит, кода свет падает на прозрачный объект. В этом случае лучи света проходят через объект, а не отражаются от его поверхности, как было в случае непрозрачного объекта. Однако когда лучи проникают сквозь поверхность, их направление меняется; это явление называют **преломлением** света, как показано на рис. 10.7. Степень преломления определяется показателями преломления материалов, сквозь которые распространяется луч света. Показатель преломления связан с плотностью материала: более плотные материалы обычно имеют более высокий показатель преломления, чем менее плотные. Когда световой луч проходит из материала с меньшим коэффициентом преломления в материал с более высоким коэффициентом преломления (например, из воздуха в воду), он изгибается по направлению к нормали в точке входа. Если же луч света переходит в материал с более низким показателем преломления, он отклоняется от нормали в точке входа.



Рис. 10.7. Преломление света

Для правильного рендеринга прозрачных объектов программное обеспечение процедуры рендеринга должно знать и учитывать показатели преломления присутствующих в сцене материалов и сред. Но это еще не все. Это программное обеспечение также должно знать, какая сторона поверхности объекта представляет внутреннюю часть объекта, а какая — внешнюю. Иначе говоря, важно, что именно происходит: свет *входит* в объект или *выходит* из него? Методы получения этой информации иногда могут быть довольно тонкими.

Например, если принять соглашение всегда перечислять вершины каждой грани в полигональной сетке в направлении против часовой стрелки, глядя на эту грань снаружи, а не изнутри объекта, то полученный список позволит легко определить, какая сторона грани представляет наружную сторону объекта, а какая внутреннюю.

Отсечение, построчное сканирование и удаление невидимых поверхностей

Теперь пора сосредоточиться на собственно процессе создания изображения из графа сцены. Выбранный здесь подход заключается в строгом следовании всем тем методам, которые используются сейчас в большинстве систем интерактивных видеоигр. В совокупности все эти методы образуют устоявшуюся парадигму, известную как **конвейер рендеринга** (rendering pipeline) или **потоковый рендеринг**. Некоторые плюсы и минусы данного подхода будут обсуждаться в конце этого раздела, а в следующем разделе будут рассмотрены две возможные альтернативы. На данный момент полезно будет лишь дополнительно отметить, что конвейер рендеринга манипулирует непрозрачными объектами, и, следовательно, преломление в этом случае не принимается во внимание. Более того, в этом подходе также игнорируется световое взаимодействие между объектами в сцене, так что пока нас не интересуют зеркальные отражения и тени.

Работа конвейера рендеринга начинается с идентификации в трехмерной сцене той области, которая содержит объекты (или части объектов), "видимые" камерой. Эта область, называемая **визуальным коридором**, представляет собой пространство внутри пирамиды, определяемое прямыми линиями, проходящими от центра проекции через углы окна изображения, как показано на рис. 10.8.

Когда визуальный коридор будет идентифицирован, следующая задача состоит в том, чтобы исключить из рассмотрения все те объекты или их части, которые не попадают в пределы визуального коридора. Совершенно очевидно, что проекция этих частей сцены выйдет за пределы окна изображения и, следовательно, не появится в конечном изображении. На первом этапе данной процедуры из рассмотрения исключаются все объекты, которые полностью находятся за пределами визуального коридора. Для упрощения этого процесса граф сцены может быть организован в виде древовидной структуры, в которой объекты в разных областях сцены хранятся в разных ветвях. В результате исключить из рассмотрения большие участки полного графа сцены можно будет, просто игнорируя целые ветви в дереве.

После идентификации и исключения объектов, которые полностью не попадают в визуальный коридор, к оставшимся объектам сцены применяется процесс, известный как **отсечение**. Его назначение — выделить и исключить из

дальнейшего рассмотрения ту часть каждого оставшегося в сцене активного объекта, которая находится за пределами визуального коридора. Точнее, процедура отсечения — это процесс сравнения каждой отдельной грани в полигональной сетке объекта с границами визуального коридора и отбрасывание той части каждой его грани, которая выходит за пределы этого коридора. Результатом выполнения всей процедуры отсечения является набор полигональных сеток (возможно, с частично обрезанными полигонами), которые полностью заключены в визуальном коридоре.

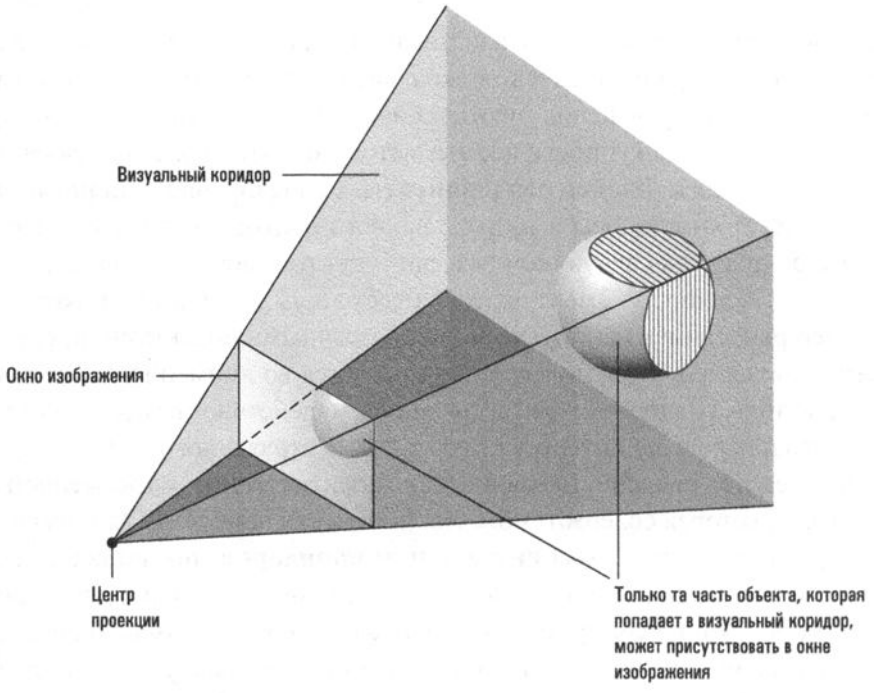


Рис. 10.8. Идентификация области сцены, которая находится внутри визуального коридора

На следующем этапе конвейера рендеринга осуществляется определение тех точек на оставшихся гранях сеток, которые должны быть связаны с позицией каждого отдельного пикселя в конечном изображении. Важно понимать, что только эти пиксели в конечном счете и будут формировать окончательный вид изображения. Если некоторая деталь на объекте попадает между позициями двух соседних пикселей, она не будет представлена собственным пикселем и, следовательно, будет отсутствовать на конечном изображении. Именно поэтому количество пикселей — самая широко рекламируемая характеристика на рынке цифровых фото- и видеокамер. Чем больше пикселей в изображении, тем больше вероятность, что на фотографии будут запечатлены самые мелкие детали.

Алиасинг или ступенчатость линий

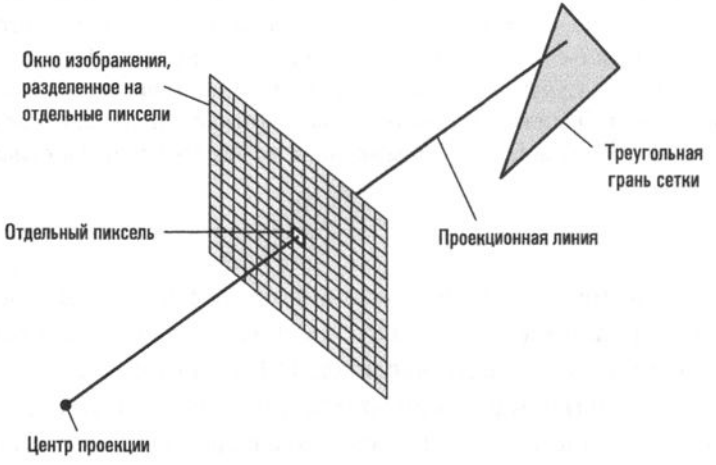
Вы когда-нибудь замечали странный "переливающийся" вид полосатых рубашек или галстуков на экранах телевизоров? Этот эффект — следствие явления, называемого **алиасингом**, которое возникает, когда узор в представленном на экране изображении неправильно соотносится с плотностью пикселей, составляющих изображение. В качестве примера предположим, что часть этого изображения состоит из чередующихся черных и белых полос, но центры всех пикселей по воле случая попадают только на черные полосы. В этом случае объект будет представлен как полностью черный. Но если объект слегка переместится, центры всех пикселей могут оказаться уже на белых полосах, из-за чего цвет объекта внезапно изменится на белый. Существует множество способов компенсировать этот раздражающий эффект. Один из них — визуализировать каждый пиксель как среднее значение небольшой области на изображении, а не как точное представление его отдельной точки.

Процесс связывания пиксельных позиций с точками в сцене называется **построчным сканированием** (поскольку предполагает преобразование граней полигональных сеток в горизонтальные ряды (строки) пикселей, называемые линиями сканирования) или **растеризацией** (поскольку массив пикселей часто называют *растром*). Построчное сканирование выполняется путем проведения прямых линий (проекционных линий) от центра проекции через позицию каждого пикселя в окне изображения с последующим определением точек, в которых эти проекционные линии пересекают грани полигональных сеток. Это те точки на гранях сетки, в которых требуется определить внешний вид данного объекта. И действительно, ведь это именно те точки объектов, которые будут представлены пикселями на конечном изображении.

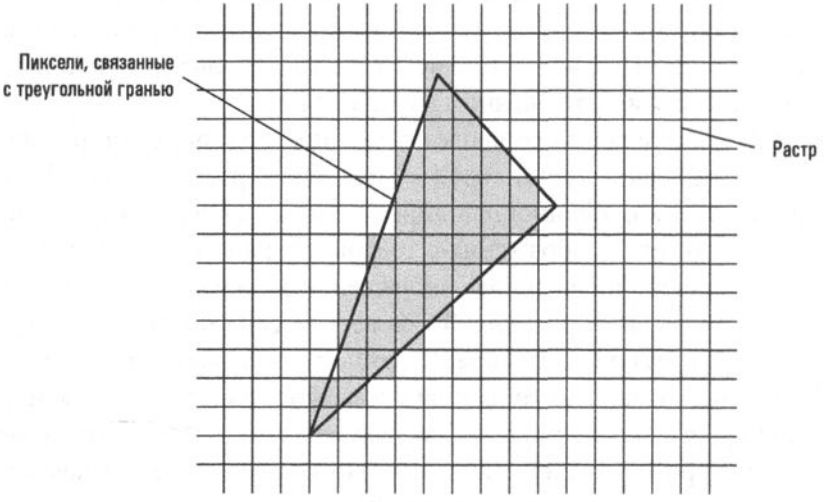
На рис. 10.9 представлена схема процедуры построчного сканирования (или растеризации) для одной треугольной грани некоторого объекта. В части *а* рисунка показано, как проекционная линия используется для того, чтобы связать положение пикселя изображения с точкой на грани полигональной сетки. В части *б* представлено пиксельное изображение этой грани, полученное в результате построчного сканирования. Здесь весь массив пикселей (растр) представлен сеткой, а пиксели, связанные с данной треугольной гранью, заштрихованы. Обратите внимание, что на рисунке можно заметить искажения, часто возникающие при растеризации фигуры, размеры которой достаточно малы по сравнению с размером пикселей. Такие неровные края хорошо знакомы пользователям, работающим за экранами персональных компьютеров.

К сожалению, процедура построчного сканирования для всей сцены (или даже для одного объекта) будет вовсе не такой простой, как растеризация

отдельной грани сетки. Дело в том, что, когда в сцене задействовано множество граней разных объектов, грани одного объекта могут мешать просмотру граней другого. Следовательно, даже если проекционная линия пересекает некоторую грань отдельной полигональной сетки в определенной точке, это вовсе не означает, что данная точка этой грани будет видна и на конечном изображении. Процесс выявления и удаления тех точек в сцене, просмотр которых на конечном изображении блокируется другими объектами, называют **удалением невидимых поверхностей**.



а) Процесс сканирования строк



б) Растр отображает "проецируемую форму" треугольной грани полигональной сетки

Рис. 10.9. Растеризация треугольной грани полигональной сетки

Конкретной версией процедуры удаления невидимых поверхностей является удаление **задних поверхностей**, которое включает в себя отбрасывание из рассмотрения всех участков полигональной сетки, которые представляют "обратную сторону" объекта. Обратите внимание, что эта процедура будет относительно простой, поскольку грани на обратной, невидимой стороне объекта легко могут быть идентифицированы как такие, которые обращены к камере своей внутренней стороной.

Однако полное решение задачи удаления невидимых поверхностей требует гораздо большего, чем простое удаление невидимых сторон объектов. Представьте, например, сцену, в которой перед зданием находится автомобиль. Грани полигональных сеток как автомобиля, так и здания будут проецироваться на одну и ту же область окна изображения. В тех случаях, когда происходит подобное перекрытие, данные пикселей, в конечном итоге сохраняемые в буфере кадров, должны отмечать присутствие объекта на переднем плане (автомобиль), а не отображать объект на заднем плане (здание). Короче говоря, если проекционная линия пересекает более одной грани, пиксель в растрированном изображении должен представлять точку на той из них, которая находится ближе всех к окну изображения.

Упрощенный подход к решению проблемы "передний план/фон", известный как **алгоритм художника**, состоит в том, чтобы сначала расположить объекты в сцене в соответствии с их расстоянием от камеры. Далее, в процессе построчного сканирования, первыми обрабатываются более удаленные объекты, а затем более близкие, — такой подход позволяет результатам сканирования последних переопределить при необходимости любые предыдущие результаты. К сожалению, алгоритм художника не справляется со случаями, когда объекты взаимно накладываются друг на друга. Так, некоторая часть дерева может находиться за другим объектом, тогда как иная часть этого же дерева может находиться перед тем же объектом. Более всеобъемлющее решение проблемы "передний план/фон" можно получить, если сфокусироваться на отдельных пикселях изображения, а не на объектах в целом. Наиболее популярный метод такого типа предполагает использование дополнительной области хранения, называемой **z-буфером** (или **буфером глубины**). В этом буфере для каждого пикселя в изображении (или, что эквивалентно, для каждого пикселя в буфере кадра) имеется отдельная ячейка, предназначенная для хранения данных о расстоянии *вдоль* соответствующей проекционной линии от камеры до точки на грани объекта, информация о которой представлена в данный момент в соответствующей позиции в буфере кадров. В результате с помощью z-буфера проблема "переднего плана/фона" может быть решена посредством вычисления и сохранения внешнего вида пикселя в буфере кадра только в том случае, если для этого пикселя данные в буфере кадра пока отсутствуют или если точка на рассматриваемом в

данный момент объекте находится *ближе*, чем точка ранее визуализированного объекта, что можно определить исходя из информации о расстоянии, записанной для этого пикселя в z-буфере.

Если говорить точнее, то при использовании z-буфера процесс рендеринга может осуществляться следующим образом. Сначала во все ячейки z-буфера помещается значение, представляющее максимальное расстояние от камеры до объектов, которые могут подвергаться рендерингу. Затем, каждый раз при выборе для рендеринга новой точки на определенной грани некоторой сетки нужно будет сначала сравнить расстояние от нее до камеры со значением в той ячейке z-буфера, которая связана с текущей пиксельной позицией. Если это расстояние меньше значения, хранящегося в ячейке z-буфера, следует вычислить внешний вид точки, записать результат в буфер кадра и заменить прежнее значение в ячейке z-буфера расстоянием до только что отрисованной точки. (Обратите внимание: если расстояние до очередной точки *больше* значения, найденного в ячейке z-буфера, то никаких действий предпринимать *не нужно*, — либо эта точка на грани находится слишком далеко, чтобы ее вообще можно было рассмотреть, либо она заблокирована точкой грани, расположенной ближе данной и уже прошедшей рендеринг.)

В конечном счете, поскольку объем вычислений, необходимых для проведения рендеринга по всему графу сцены, может оказаться слишком большим, предварительно применяются различные методы, позволяющие исключить из рассмотрения те элементы объектной модели, которые не могут повлиять на конечный результат. Это общая стратегия, которая может оказаться эффективной и во многих других задачах моделирования и эмуляции, в том числе и вне области компьютерной графики.



Основные положения для запоминания

- **В моделях часто опускаются ненужные свойства тех объектов или явлений, которые моделируются.**

Затенение

Как только процедура построчного сканирования идентифицирует на грани сетки ту точку, которая должна будет появиться в конечном изображении, задача рендеринга сводится к определению внешнего вида грани в данной точке. Этот процесс называется **затенением**. Обратите внимание, что затенение включает в себя вычисление характеристик света, проецируемого на камеру из рассматриваемой точки, которые, в свою очередь, зависят от ориентации

поверхности грани в этой точке. В конце концов, именно ориентация поверхности в конкретной точке определяет степень зеркального, диффузного и рассеянного света, фиксируемого камерой.

Простое решение проблемы с затенением, называемое **постоянным затенением**, состоит в том, чтобы использовать ориентацию плоской грани в качестве ориентации для каждой ее точки, т.е. предполагать, что поверхность каждой грани является абсолютно плоской. Однако в этом случае конечный результат оказывается таким, что полученное изображение выглядит как бы *граненым*, что хорошо видно на рис. 10.10, тогда как предполагается, что оно должны быть достаточно сглаженным, как показано на рисунке 10.6. В определенном смысле постоянное затенение фактически создает изображение самой полигональной сетки, а не того объекта, который этой сеткой моделируется.

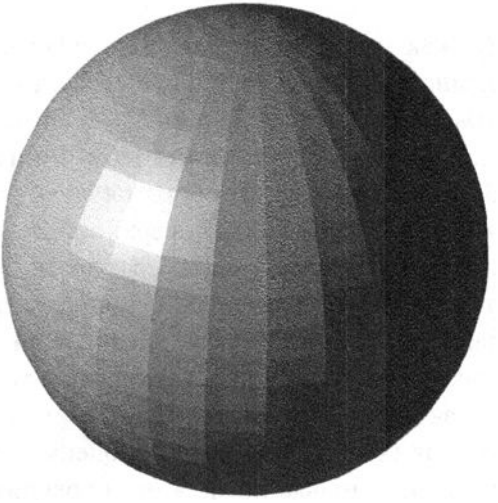


Рис. 10.10. Сфера при рендеринге с постоянным затенением

Чтобы получить более реалистичное изображение, процесс рендеринга должен преобразовать внешний вид отдельных плоских граней сетки в некую гладко изогнутую поверхность. Этот результат достигается путем оценки истинной ориентации исходной поверхности в каждой отдельной отображаемой точке.

Такие схемы оценки обычно начинаются с данных, указывающих ориентацию поверхности в вершинах многоугольной сетки. Существует несколько способов получить эти данные. Один из них заключается в кодировании ориентации исходной поверхности в каждой вершине и присоединении этих данных к полигональной сетке, выполняемом как часть процесса моделирования. В результате получается полигональная сетка со стрелками, называемыми **нормальными векторами**, прикрепленными к каждой вершине. Каждый нормальный вектор направлен наружу в направлении, перпендикулярном исходной

поверхности. В конечном счете создается полигональная сетка, которая может быть представлена так, как показано на рис. 10.11. (Другой подход состоит в вычислении ориентации каждой грани, смежной с вершиной, с последующим нахождением "среднего" для этих ориентаций, которое и будет использоваться в качестве оценки ориентации поверхности в данной вершине.)

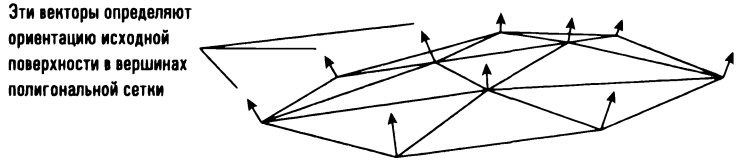


Рис. 10.11. Концептуальный вид полигональной сетки с нормальными векторами в ее вершинах

Независимо от того, как именно определяется ориентация исходной поверхности в вершинах многоугольной сетки, существует несколько стратегий затенения плоскости грани на основе этих данных. К ним, в частности, относятся затенение методом **Гуро** и затенение методом **Фонга**, различие между которыми является достаточно тонким. В обоих случаях работа начинается с использования информации об ориентации поверхности в вершинах грани для аппроксимации ориентации поверхности вдоль ее границ. Далее при затенении по методу Гуро эта информация используется для определения внешнего вида поверхности вдоль границ грани, после чего внешний вид вдоль границ интерполируется для оценки внешнего вида поверхности в точках внутри грани. В отличие от этого при затенении по методу Фонга ориентация поверхности вдоль границ грани сначала интерполируется для оценки ориентации поверхности в точках внутри грани, и только после этого проводится определение ее внешнего вида. (Короче говоря, при затенении по методу Гуро информация об ориентации вдоль границ преобразуется в информацию об их цвете, которая затем интерполируется для определения цвета остальных точек грани. При затенении по методу Фонга информация об ориентации интерполируется до тех пор, пока не будет выяснена ориентация рассматриваемой точки, после чего эта информация об ориентации преобразуется в информацию о цвете.) В конечном итоге считается, что затенение по методу Фонга с большей вероятностью обнаружит зеркальное отражение света во внутренней части грани, поскольку этот метод более чувствителен к изменениям ориентации поверхности. (См. вопрос 3 в конце этого раздела.)

И наконец, следует отметить, что основные методы затенения могут быть расширены с целью наложения текстуры на внешний вид поверхности. Например, метод, называемый **рельефным текстурированием** (bump mapping), по сути является способом генерирования небольших изменений в видимой

ориентации поверхности, так что в конечном счете поверхность будет казаться шероховатой. Точнее говоря, рельефное текстурирование добавляет определенную степень случайности процессу интерполяции, применяемому традиционными алгоритмами затенения, так что в конечном счете создается впечатление, что вся поверхность имеет определенную текстуру, как показано на рис.10.12.

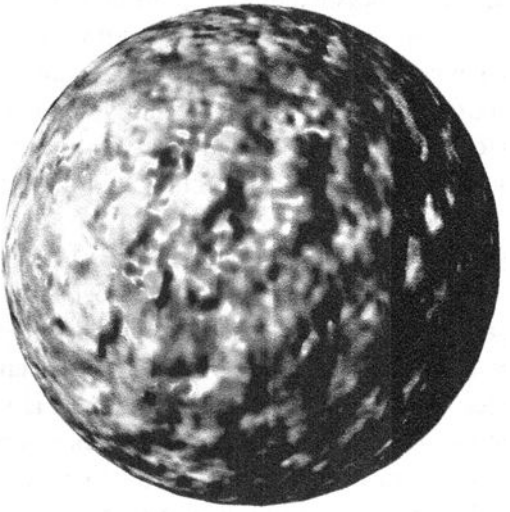


Рис. 10.12. Сфера при рендеринге с использованием рельефного текстурирования

Специализированное оборудование для реализации конвейера рендеринга

Как уже говорилось, вся совокупность процессов отсечения, построчного сканирования, удаления невидимых поверхностей и затенения рассматривается как последовательность этапов процедуры, называемой *конвейером рендеринга*. Поскольку эффективные алгоритмы решения этих отдельных задач хорошо известны, они были реализованы непосредственно в электронных микросхемах, автоматически выполняющих все расчеты, связанные с реализацией конвейера рендеринга в полном объеме. Сегодня даже недорогие образцы подобных микросхем способны выполнять вычисления, необходимые для отображения миллионов плоских граней в секунду.

Большинство компьютерных систем, предназначенных для поддержки графических приложений, в том числе игровые автоматы, имеют в своей структуре подобные микросхемы. В случае компьютерных систем более общего назначения соответствующая электронная подсистема может быть реализована в виде **графической карты** или **графического адаптера**, подключаемого к

шине компьютера в качестве специализированного контроллера (см. главу 2). Такое аппаратное обеспечение существенно сокращает время, необходимое для выполнения процесса рендеринга. Также аппаратная реализация процесса рендеринга позволяет значительно снизить сложность графического прикладного программного обеспечения. По сути, все, что требуется сделать такому прикладному программному обеспечению, — это предоставить графическому адаптеру *граф сцены*. Далее микросхемы аппаратной части самостоятельно выполняют все этапы конвейера рендеринга и помещают полученные результаты в буфер кадра. Таким образом, с точки зрения прикладного программного обеспечения весь конвейер рендеринга сводится к одному этапу с использованием специализированного аппаратного обеспечения в качестве абстрактного инструмента.

В качестве примера давайте снова обратимся к интерактивной видеоигре. При инициализации игры игровое программное обеспечение передает граф исходной сцены в графическое оборудование. Затем сцена визуализируется аппаратными средствами и полученное изображение помещается в буфер кадра, откуда автоматически выводится на экран монитора. По ходу игры игровое программное обеспечение просто обновляет граф сцены, передаваемый графическому адаптеру, таким образом, чтобы отразить изменяющуюся игровую ситуацию, а аппаратное обеспечение графического адаптера многократно воспроизводит сцену, каждый раз помещая обновленное изображение в буфер кадра.

Современный **графический процессор** (GPU), устанавливаемый во многих графических адаптерах, в настоящее время является настолько совершенным абстрактным инструментом, что его можно использовать и для вычислений при проведении других типов моделирования, требующих интенсивных расчетов, — при условии, что они могут быть сформулированы в виде, аналогичном вычислениям, выполняемым в процессе рендеринга. Когда графические процессоры не используются для визуализации изображений, выводимых на экран в режиме реального времени, они вполне могут использоваться для получения результатов числовых вычислений общего назначения. Время, необходимое для рендеринга сложной графической сцены или выполнения расчетов численного моделирования, зависит не только от сложности базовой модели, но и от используемых аппаратных средств и программных систем.



Основные положения для запоминания

- **Время, необходимое для моделирования, зависит от уровня детализации и качества моделей, а также от программного и аппаратного обеспечения, используемого для моделирования.**

Следует, однако, отметить, что возможности и коммуникационные свойства различных типов графического оборудования могут существенно различаться. А это означает, что, если приложение, такое как видеоигра, было разработано для конкретной графической платформы, при переносе в другую среду его потребуется изменить. Чтобы снизить подобную зависимость от специфики конкретных графических систем, были разработаны стандартные программные интерфейсы, выполняющие роль посредника между графическим оборудованием и прикладным программным обеспечением. Эти интерфейсы состоят из особых подпрограмм, преобразующих стандартизированные команды в конкретные инструкции, необходимые для управления определенной графической аппаратной системой. В качестве примера можно привести спецификацию **OpenGL** (сокращение от *Open Graphics Library*), которая представляет собой непатентованную систему, разработанную компанией Silicon Graphics и широко используемую в индустрии видеоигр, или систему **Direct3D**, которая была разработана компанией Microsoft для использования в среде ОС Microsoft Windows.

В заключение следует отметить, что наряду со всеми преимуществами, предоставляемыми использованием конвейера рендеринга, у этого подхода есть и недостатки, самым значительным из которых является тот факт, что конвейер рендеринга реализует только *локальную* модель освещения, т.е. конвейер визуализирует каждый объект независимо от других объектов. Иначе говоря, при использовании модели локального освещения каждый объект в сцене визуализируется относительно источников света так, как если бы он был единственным объектом в этой сцене. В результате любые световые взаимодействия между объектами, такие как тени или отражения, не рассматриваются. Это — основное отличие данного подхода от *глобальной* модели освещения, в которой подобные взаимодействия между объектами обязательно анализируются и учитываются. Два метода реализации глобальной модели освещения будут рассмотрены в следующем разделе, а пока достаточно будет лишь отметить, что оба эти метода все еще лежат за пределами возможностей современных технологий визуализации в реальном масштабе времени.

Однако это не означает, что графические системы, использующие аппаратное обеспечение конвейера рендеринга, вовсе не способны генерировать некоторые глобальные эффекты освещения. В действительности для преодоления некоторых ограничений, накладываемых моделью локального освещения, были разработаны определенные хитроумные методы. В частности, вид **теней**, отбрасываемых объектами сцены на землю, можно смоделировать в контексте модели локального освещения, создав копию полигональной сетки объекта, отбрасывающего тень, "расплющить" ее и поместить на поверхность, представляющую землю, окрасив в темные тона. Другими словами, тень моделируется

так, как если бы она была другим объектом, который затем может быть визуализирован традиционным оборудованием конвейерного рендеринга для создания иллюзии тени. Такие методы популярны как в приложениях "пользовательского уровня", например в интерактивных видеоиграх, так и в приложениях "профессионального уровня", например в имитаторах полета.

10.4. Вопросы и упражнения

1. Обобщите различия, существующие между отраженным зеркальным светом, диффузным светом и рассеянным светом.
2. Дайте определение терминам "отсечение" и "построчное сканирование".
3. Принципы затенения по методу Гуро и затенения по методу Фонга можно сформулировать следующим образом. При затенении по методу Гуро ориентация поверхности объекта вдоль границ грани используется для определения внешнего вида грани вдоль ее границ, а затем эта информация интерполируется на точки внутренней части грани для определения внешнего вида ее рассматриваемых точек. При затенении по методу Фонга ориентация поверхности объекта вдоль границ грани интерполируется с целью вычисления ориентации внутренних точек грани, а затем эта информация используется для определения внешнего вида рассматриваемых точек. Чем может различаться внешний вид одного и того же объекта, в каждом из случаев?
4. В чем состоит основное преимущество, обеспечиваемое реализацией конвейера рендеринга?
5. Опишите, как в локальной модели освещения можно смоделировать зеркальные отражения.

10.5. Моделирование глобального освещения

В настоящее время ведутся интенсивные исследования в отношении двух альтернатив методу конвейера рендеринга, в каждой из которых реализуется глобальная модель освещения, а значит, обеспечивается возможность преодоления ограничений, свойственных локальной модели освещения, присущей традиционному конвейеру. Одна из этих альтернатив — метод трассировки лучей, а другая — метод излучательности. Как будет скоро показано, оба этих метода являются весьма детальными и трудоемкими процессами.

Трассировка лучей

Метод трассировки лучей — это процесс следования по лучу света от экрана назад, к источнику этого луча. Процесс начинается с выбора пикселя для визуализации, определения прямой линии, проходящей через этот пиксель и центр проекции, и отслеживания светового луча, который падает на окно изображения вдоль этой линии. Сам процесс трассировки предполагает следование по лучу вглубь виртуальной сцены, пока он не коснется поверхности некоторого объекта. Если этот объект является источником света, процесс трассировки лучей заканчивается, и пиксель в окне изображения отображается как точка на источнике света. В противном случае оцениваются свойства поверхности найденного объекта с целью определения направления того *входящего* луча света, который был отражен в данной точке с получением *отраженного* луча, который был выбран и отслеживался в обратном направлении в начале процесса. Далее процесс отслеживания движется в обратном направлении по только что найденному входящему лучу с целью обнаружения его источника, и когда он будет найден, процедура повторяется.

Пример трассировки лучей изображен на рис. 10.13. Здесь мы видим луч, прорисованный назад через окно изображения к поверхности зеркала. Оттуда луч трассируется до поверхности блестящего шара, откуда идет обратно к зеркалу и наконец проходит от зеркала к источнику света. Основываясь на информации, полученной в результате этого процесса трассировки, пиксель на изображении должен выглядеть, как точка на шаре, освещаемая источником света, отраженным в зеркале.

Недостатком метода трассировки лучей является то, что в нем отслеживаются только зеркальные отражения. В результате всем объектам, визуализированным с использованием этого метода, свойственна тенденция иметь блестящий вид. Чтобы противостоять этому эффекту, был разработан модифицированный вариант метода трассировки лучей, получивший название **распределенная трассировка лучей**. Разница состоит в том, что вместо трассировки одного луча назад от точки отражения, при распределенной трассировке лучей отслеживается несколько лучей, исходящих из этой точки, каждый из которых идет в несколько ином направлении.

Другой вариант базового метода трассировки лучей применим, когда в сцене присутствуют прозрачные объекты. В этом случае каждый раз, когда луч прослеживается до соприкосновения с поверхностью, необходимо будет учитывать два эффекта. Первый — отражение, а второй — преломление. В этой ситуации задача отслеживания исходного луча разделяется на две задачи: отслеживание отраженного луча в обратном направлении и отслеживание преломленного луча в обратном направлении.

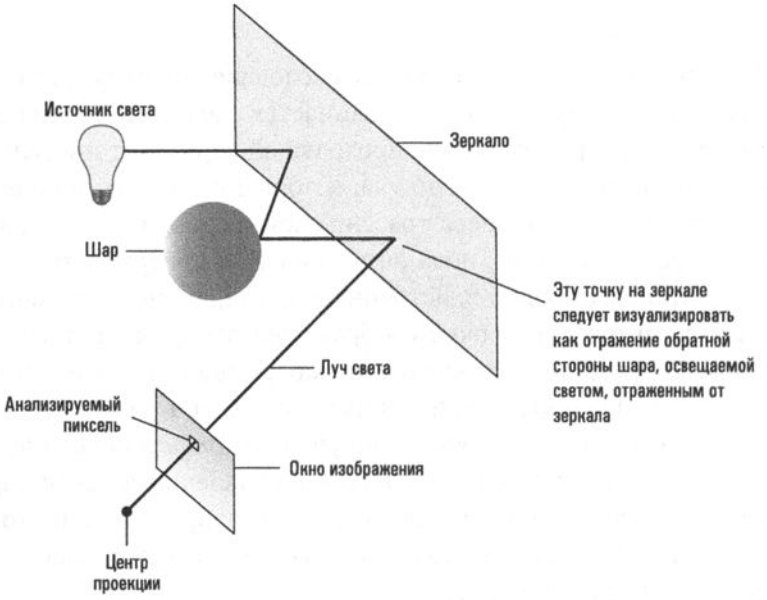


Рис. 10.13. Трассировка лучей

Трассировка лучей обычно выполняется рекурсивно, при этом в каждой активации предлагается прослеживание луча до его источника. Первая активация может проследить луч до блестящей непрозрачной поверхности и выяснить, что он является отражением входящего луча, после чего вычисляется направление данного входящего луча и для его отслеживания инициализируется другая активация. Эта вторая активация будет выполнять аналогичную задачу поиска источника своего луча, — процесс, который может привести к вызову других активаций.

Для прекращения рекурсивной трассировки лучей могут использоваться различные условия. Трассируемый луч может достичь источника света, трассируемый луч может покинуть сцену, так и не встретив на своем пути какой-либо объект, или количество активаций может достигнуть заранее установленного предела. Еще одно условие завершения трассировки может быть построено на анализе поглощающих свойств встречающихся поверхностей. Если поверхность обладает высокой поглощающей способностью — например, темная матовая поверхность, — то любой входящий луч будет иметь лишь незначительное влияние на внешний вид этой поверхности, а значит, процесс трассировки лучей можно будет прекратить. Аналогичный эффект может иметь факт накопления поглощения луча; в этом случае трассировка лучей может прекратиться после посещения нескольких умеренно поглощающих поверхностей.

Основанный на использовании глобальной модели освещения, метод трассировки лучей позволяет избежать многих ограничений, присущих традиционному конвейеру рендеринга. Например, проблемы удаления скрытой поверх-

ности и обнаружения теней естественным образом решаются непосредственно в процессе трассировки лучей. К сожалению, методу трассировки лучей свойственен большой недостаток — он требует больших затрат времени. По мере того как каждое отражение прослеживается до его источника, количество необходимых вычислений возрастает чрезвычайно быстро — проблема, которая лишь усугубляется, если учитывать преломления или применять распределенную трассировку лучей. В результате метод трассировки лучей все еще не реализован в системах реального времени "пользовательского уровня", таких как интерактивные видеоигры, но достаточно часто встречается в приложениях "профессионального уровня", которые не столь нуждаются в поддержке режима реального времени, например в графическом программном обеспечении, используемом на киностудиях.

Метод излучательности

Еще одна альтернатива традиционному конвейеру рендеринга — это **метод излучательности**. Если в методе трассировки лучей используется точечный подход к отслеживанию отдельных лучей света, то в методе излучательности применяется более "пространственный" подход, при котором анализируется общая световая энергия, проходящая между парами плоских единиц поверхности, называемых *патчами*. Эта излучаемая световая энергия является, по существу, диффузным светом. Энергия света, которая излучается от объекта, либо генерируется самим объектом (как в случае источника света), либо отражается от объекта. Внешний вид каждого объекта в конечном счете определяется с учетом той энергии света, которую он получает от других объектов.

Степень, в которой свет, излучаемый одним объектом, влияет на внешний вид другого, определяется параметрами, называемыми **форм-факторами**. В сцене, которая визуализируется, с каждой парой патчей связан свой уникальный форм-фактор. В этих форм-факторах принимаются во внимание геометрические отношения между патчами, такие как разделяющее их расстояние и относительная ориентация. Чтобы определить внешний вид в сцене отдельного патча, определяется количество световой энергии, получаемой им от всех других патчей в сцене, вычисляемое с использованием соответствующего форм-фактора при каждом просчете. Полученные результаты объединяются для получения одного цвета и его интенсивности для каждого патча. Затем, чтобы получить не граненую, а гладкую поверхность объектов в сцене, эти значения интерполируются между соседними патчами с использованием методов, аналогичных тем, которые используются в процедуре затенения по методу Гуро.

Поскольку в расчетах используется множество патчей, для каждого из которых необходимо выполнить полный объем соответствующих вычислений,

методу излучательности свойственна очень большая вычислительная нагрузка. В результате, как и в случае метода трассировки лучей, его требования в этом отношении выходят далеко за пределы возможностей графических систем реального времени, доступных в настоящее время на потребительском рынке. Другая проблема, связанная с использованием метода излучательности, заключается в том, что, поскольку этот метод работает с элементами, представляющими собой плоские патчи, а не отдельные точки, оказывается невозможным уловить детали отображения зеркального света, а это означает, что всем поверхностям, визуализируемым по методу излучательности, свойственна тенденция иметь тусклый вид.

Однако у метода излучательности есть и свои достоинства. Одно из них состоит в том, что определение внешнего вида объектов посредством анализа прохождения световой энергии не зависит от камеры. В результате, как только будут закончены все вычисления по определению прохождения световой энергии в сцене, окончательный ее рендеринг может быть быстро выполнен для различных положений камеры. Другое достоинство заключается в том, что метод излучательности позволяет принять во внимание многие тонкие характеристики света, такие как **смешение цветов**, когда цвет одного объекта влияет на оттенок других объектов вокруг него. Как следствие метод излучательности также находит свое применение. Одна из таких областей — графическое программное обеспечение, используемое при визуализации архитектурных проектов. И действительно, свет в проектируемом здании состоит в основном из диффузного и рассеянного света, так что зеркальные эффекты в данном случае не играют значительной роли, а вот тот факт, что новые положения камеры могут обрабатываться весьма эффективно, имеет очень большое значение, поскольку обеспечивает архитектору возможность быстро просматривать разные помещения с разных точек зрения.

10.5. Вопросы и упражнения

1. Почему в методе трассировки лучей отслеживание отдельных лучей выполняется в обратном направлении, от окна визуализации к источнику света, а не от источника света к окну визуализации?
2. Чем различаются методы прямой трассировки лучей и распределенной трассировки лучей?
3. В чем заключаются два основных недостатка метода излучательности?
4. В чем сходство метода трассировки лучей и метода излучательности? Чем эти методы различаются?

10.6. Анимация

Теперь мы переходим к теме **компьютерной анимации** (CGI — computer-generated imagery), связанной с использованием компьютерных технологий для создания и отображения серии изображений, имитирующих движение. Относительно недорогие, но мощные программные инструменты компьютерной анимации позволили мультипликаторам создавать более визуально точные и сложные произведения, чем могло бы быть экономически выгодным при использовании традиционных методов рисованной анимации. Однако компьютерная анимация получает все большее распространение не только из-за потенциально меньших финансовых затрат, но и потому, что ее мощные инструменты позволяют создателям ставить все более и более амбициозные цели и успешно из реализовывать в своих новых произведениях.



Основные положения для запоминания

- **Рост вычислительной мощности компьютеров упрощает создание и модификацию цифровых художественных произведений с повышенной детализацией и точностью.**
- **Достижения в области вычислительной техники обеспечили появление и стимулировали дальнейший рост творческого потенциала в других областях.**

Основы анимации

Мы начнем с введения некоторых основных концепций анимации.

Кадры

Анимация достигается путем отображения некоторой последовательности изображений, называемых **кадрами**, с достаточно высокой скоростью. На этих кадрах зафиксирован вид изменяющейся сцены через регулярные промежутки времени, и в результате их последовательного отображения возникает иллюзия наблюдения сцены, изменяющейся во времени. Стандартная скорость отображения в киноиндустрии составляет двадцать четыре кадра в секунду. Стандарт в потоковом видео составляет шестьдесят кадров в секунду (хотя, поскольку каждый второй видеокадр создается так, что он переплетается с предыдущим кадром для получения полного детального изображения, такое видео также можно классифицировать как систему, работающую со скоростью тридцать кадров в секунду).

Кинеограф

Кинеограф — это книга с кадрами анимации, которые при быстром перелистывании страниц имитируют движение. Книгу, которую вы сейчас держите в руках, также можно превратить в собственный кинеограф (при условии, что вы пока еще не заполнили ее поля собственными заметками и рисунками). Просто поместите где-нибудь на полях первой страницы точку, а затем, воспользовавшись ее отпечатком на третьей странице, поместите на третьей странице еще одну точку, немного сместив ее относительно точки на первой странице. Далее повторяйте этот процесс на каждой следующей нечетной странице, пока не дойдете до конца книги. Теперь быстро пролистайте страницы и посмотрите, как точка бойко перемещается по их полям. Отлично! Вы сделали собственный кинеограф и, возможно, одновременно сделали первый шаг к карьере аниматора! В качестве эксперимента по кинематике попробуйте вместо простой точки рисовать какую-нибудь фигурку и делать это так, чтобы возникала иллюзия, что фигурка шагает туда и обратно. А теперь поэкспериментируйте с динамикой, создав изображение капли воды, падающей на землю.

Кадры могут быть получены с помощью традиционной фотографии или созданы искусственно с помощью компьютерной графики. Более того, эти два метода вполне могут быть объединены. Например, программное обеспечение для 2D-графики часто используется для изменения изображений, полученных с помощью фотографических методов, с целью удаления из кадра поддерживающих канатов, наложения дополнительных изображений или создания иллюзии **морфинга**, т.е. процесса превращения одного объекта в другой.

Более внимательный взгляд на морфинг дает интересную информацию о процессе анимации. Построение эффекта морфинга начинается с определения пары ключевых кадров, которые в конце работы будут заключать между собой всю морфинговую последовательность. Первый — это последнее изображение объекта перед тем, как должен произойти его морфинг, а второе — это первое изображение объекта после того, как его морфинг завершился. (В традиционном производстве кинофильмов для этого требуется "снять" две последовательности действий: одна ведет к появлению морфа, а другая отражает то, что происходит с ним после морфинга.) В предшествующем морфингу кадре некоторые элементы, такие как точки и линии, принимают как **контрольные точки**, которые затем ассоциируются с аналогичными признаками в кадре после морфинга. Морф создается с применением математических методов, которые постепенно искажают одно изображение и превращают его в другое, используя контрольные точки в качестве направляющих всего процесса. Записывая изображения, получаемые по ходу процесса искажения, можно получить короткую последовательность искусственно созданных изображений, которая заполнит

промежуток между исходными ключевыми кадрами и в конечном счете создаст иллюзию морфинга.

Раскадровка

Типичный анимационный проект начинается с создания **раскадровки**, представляющей собой последовательность двухмерных изображений, рассказывающих полную историю всего сюжета в виде набросков сцен в ключевых точках его презентации. Конечное назначение раскадровки зависит от того, как будет реализован анимационный проект: с использованием методов 2D- или 3D-графики. В проекте, использующем методы 2D-графики, раскадровка обычно преобразуется в окончательный набор кадров фильма почти так же, как это было в студиях компании Disney в 20-х годах прошлого века. В те дни художники, называемые мастерами-аниматорами, превращали раскадровку в детализированные кадры, называемые **ключевыми кадрами**, в которых определялись внешний вид персонажей и декорации через регулярные интервалы времени в процессе анимации. Затем ассистенты аниматора рисовали дополнительные кадры, заполнявшие промежутки между ключевыми кадрами таким образом, чтобы анимация выглядела непрерывной и плавной. Этот процесс восполнения пропусков носил название **фазовки** (in-betweening) или **достройки**.

Основное различие между этим процессом и тем, как это делается сегодня, состоит в том, что теперь для рисования ключевых кадров аниматоры используют специальное программное обеспечение обработки изображений и 2D-графики, а большая часть промежуточного процесса фазовки уже автоматизирована, так что должность ассистента аниматора, по существу, исчезла.

Размытие

В области традиционной фотографии в свое время много усилий было потрачено на получение четких изображений быстро движущихся объектов. В области компьютерной анимации возникает противоположная проблема. Если каждый кадр в последовательности, изображающей движущийся объект, включает этот объект как резкое изображение, то движение в конечном счете может выглядеть прерывистым. Однако высокая четкость изображений является естественным следствием создания кадров как независимых изображений стационарных объектов в графе сцены. В результате аниматоры часто искусственно искажают изображение движущегося объекта в сгенерированном компьютером кадре. Один из методов, называемый *суперсэмплингом*, состоит в том, чтобы создать несколько изображений, в которых движущийся объект смещен незначительно, и затем наложить эти изображения на один кадр. Другой метод заключается в изменении формы движущегося объекта так, чтобы он выглядел вытянутым вдоль направления движения.

3D анимация

Большинство видеоматериалов компьютерных видеоигр и полнофункциональных анимационных фильмов теперь создается с использованием методов 3D-графики. В этом случае проект все так же начинается с создания раскадровки, состоящей из двухмерных изображений. Однако вместо того, чтобы непосредственно включить раскадровку в конечный продукт, как это было в проектах с 2D-графикой, теперь раскадровка используется лишь как руководящий материал для построения трехмерного виртуального мира. Далее этот виртуальный мир многократно "фотографируется", пока объекты в нем перемещаются в соответствии со сценарием или развитием видеоигры.

Возможно, сейчас полезно будет остановиться и уточнить, что, собственно, означает выражение "перемещение объекта в пределах сгенерированной компьютером сцены". Не забывайте, что "объект" на самом деле представляет собой набор данных, хранящихся в графе сцены. Среди данных в этой коллекции есть значения, определяющие местоположение и ориентацию объекта. Следовательно, "перемещение" объекта в сцене осуществляется просто путем изменения этих значений. После внесения требуемых изменений в процессе рендеринга будут использоваться уже новые значения и это означает, что возникнет иллюзия перемещения объекта в конечном двухмерном изображении.

Кинематика и динамика

Степень, в которой движение в трехмерной графической сцене автоматизировано или управляется человеком-аниматором, варьируется в зависимости от приложения. Целью, конечно же, является автоматизация всего процесса. Как следствие множество исследований в этой области было направлено на поиск способов идентификации и моделирования движения различных природных явлений. В этом отношении особенно полезными оказались два раздела механики.

Первым является **динамика**, задача которой — описание движения объекта посредством применения законов физики с целью определения результатов воздействия сил, действующих на этот объект. Например, помимо данных о местоположении, объекту в сцене могут быть назначены направление движения, скорость и масса. Эти данные затем можно будет использовать для определения влияния гравитации или результатов столкновения с другими объектами в сцене, позволив программному обеспечению корректно рассчитать правильное местоположение объекта в следующем кадре.

В качестве примера рассмотрим задачу построения последовательности кадров анимации, изображающей плескание воды в контейнере. В графе сцены для моделирования воды можно было бы использовать систему частиц, в которой каждая частица представляла бы небольшую единицу воды. (Представьте

себе "воду", состоящую из множества больших "молекул" размером с крупную бусину.) Далее можно было бы применить законы физики, чтобы вычислить влияние на эти частицы гравитации, а также учесть взаимодействие между самими частицами, когда контейнер раскачивается из стороны в сторону. Такой подход позволит вычислять местоположение каждой частицы через регулярные промежутки времени, а затем, используя данные о расположении внешних частиц в контейнере в качестве вершин полигональной сетки, можно было бы получить сетку, представляющую поверхность воды. В результате требуемую анимацию можно было бы создать посредством многократного "фотографирования" этой сетки в процессе имитации.

Вторая область механики, используемая при моделировании движения, — **кинематика**. Ее задача состоит в описании движения объекта с точки зрения того, как части этого объекта движутся одна относительно другой. Необходимость применения законов кинематики особенно заметна при анимации сочлененных фигур, когда требуется имитировать перемещение конечностей, таких как руки и ноги. Их движения проще имитировать путем моделирования шаблонов движения суставов, чем посредством вычисления общего эффекта воздействия на эту систему сил, действующих на отдельные мышцы, а также гравитации. Таким образом, в то время как динамика может быть полезна при определении траектории прыгающего мяча, движение руки анимированного персонажа удобнее определять путем применения законов кинематики для вычисления правильных поворотов плеча, локтя или запястья. В результате множество исследований в области анимации живых персонажей фокусируется на вопросах анатомии и на том, как сустав и структура конечности влияют на движение персонажа.

Типичный метод применения кинематики — начать с представления персонажа в виде фигурки из палочек, имитирующей структуру его скелета. Затем каждый элемент этой фигурки, представляющий отдельную кость скелета, покрывается полигональной сеткой, представляющей ту часть поверхности тела персонажа, которая охватывает эту кость, после чего устанавливаются правила, определяющие, каким образом смежные сетки должны соединяться между собой. Теперь фигуркой можно будет манипулировать (со стороны программного обеспечения либо со стороны человека-аниматора), просто изменяя положение костей в суставах скелета, — примерно так, как манипулируют марионеткой. Точки, в которых "нити" (или *контролы*) управления подобной "марионеткой" прикрепляются к модели, называются **локаторами** или **управляющими элементами**.

Множество исследований, проведенных в области применения кинематики к 3D-анимации, были направлены на разработку алгоритмов автоматического вычисления последовательности положений конечностей, имитирующих их естественное движение. Кроме того, в настоящее время уже доступны

алгоритмы, автоматически генерирующие реалистичные последовательности положений персонажей при ходьбе.

Тем не менее бо́льшая часть анимации, основанной на использовании законов кинематики, по-прежнему производится путем проведения персонажа через заданную последовательность положений его костей в суставах. Эти положения могут быть определены творческими усилиями аниматора или получены методом **захвата движения**, предполагающего запись положений живой модели, выполняющей требуемое действие. В последнем случае на стратегические точки тела человека наклеиваются кусочки светоотражающей ленты, а затем его снимают под разными углами, например при выполнении броска бейсбольного мяча. Позднее, отслеживая местоположение этих наклеенных фрагментов ленты в различных кадрах, можно будет установить точную ориентацию рук и ног человека в процессе броска, после чего полученные сведения об ориентации могут быть применены к персонажам в анимации.

Процесс анимации

Конечная цель исследований, проводимых в области анимации, — автоматизировать весь процесс анимации. В этом случае предполагается создание программного обеспечения, которое при заданных конкретных параметрах будет автоматически генерировать требуемую анимационную последовательность. Прогресс в этом направлении демонстрируется тем фактом, что киноиндустрия в настоящее время производит изображения толп людей, батальных сцен или панически бегущих стад животных с использованием отдельных виртуальных "роботов", которые автоматически перемещаются в графе сцены, и при этом каждый из них реализует собственный назначенный только ему сценарий движения. В результате сцены, которые, возможно, потребовали бы участия сотен или даже тысяч актеров-статистов при съемках в реальном мире, оказывается возможным смоделировать виртуально при относительно небольшой стоимости и с полной гарантией того, что ни один из реальных актеров не пострадает от пиротехники или неаккуратного обращения с реквизитом.



Основные положения для запоминания

- **Моделирование позволяет имитировать события реального мира без чрезмерных затрат и потенциальных опасностей, связанных с проведением съемок этих событий в реальном мире.**

Интересный случай произошел при съемках фантастической армии орков и людей в трилогии "Властелин колец". Каждый воин на экране был смоделирован как отдельный "интеллектуальный" объект со своими физическими характеристиками и случайно выбранными личными качествами, которые определяли его склонность атаковать или бежать. В тестовой имитации фрагмента битвы в Хельмовой пади во второй части трилогии у орков оказалась слишком большая склонность к бегству, и они просто убежали при первом же столкновении с воинами-людьми. (Возможно, это был первый случай, когда виртуальные статисты посчитали выполняемую ими работу слишком опасной.)

Конечно, сегодня достаточно большой объем анимации все еще создается аниматорами-людьми. Однако вместо того, чтобы рисовать отдельные двухмерные кадры вручную, как это было в 20-х годах прошлого века, сейчас аниматоры используют программное обеспечение, позволяющее манипулировать трехмерными виртуальными объектами в графе сцены способом, напоминающим управление куклами-марионетками, — как объяснялось в предыдущем разделе при обсуждении применения в анимации законов кинематики. В результате аниматор получает возможность создать серию виртуальных сцен, которые затем будут "сфотографированы" с целью создания анимации. В некоторых случаях этот метод применяется только при создании сцен ключевых кадров, а затем используется дополнительное программное обеспечение, обеспечивающее создание всех промежуточных кадров путем автоматической визуализации сцены. В подобном программном обеспечении законы динамики и кинематики используются для перемещения всех объектов в графе сцены из положений в одном ключевом кадре сцены к положениям, которые они занимают в следующем ключевом кадре.

По мере прогресса в исследованиях, проводимых в области компьютерной графики, и последующего совершенствования используемых технологий процесс анимации, безусловно, будет становиться все более и более автоматизированным. Станет ли в киноиндустрии участие аниматоров-людей, а также реальных актеров и физических декораций когда-нибудь уже устаревшим и излишним, пока еще неизвестно, но многие полагают, что это обязательно произойдет, и причем в не столь отдаленном будущем. В действительности компьютерная 3D-графика может оказать на киноиндустрию существенно большее влияние, чем оказал в свое время переход от немых фильмов к звуковым.

10.6. Вопросы и упражнения

1. Изображение, которое видит человек, обычно задерживается в его восприятии приблизительно на 200 миллисекунд. Исходя из этого приближения, какое минимальное количество изображений должно быть представлено человеку в секунду для получения эффекта анимации? Как это приближение соотносится с тем количеством кадров в секунду, которое используется в киноиндустрии?
2. Что такое *раскадровка*?
3. Что такое *фазовка*?
4. Дайте определение терминам *кинематика* и *динамика*.

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. Какие из следующих приложений являются приложениями 2D-графики, а какие — 3D-графики?

   а. Разработка макета страниц журнала.

   б. Рисование изображений с помощью приложения Microsoft Paint.

   в. Создание изображений из виртуального мира для видеоигры.

2. Что в контексте 3D-графики соответствует каждому из следующих элементов традиционной фотографии? Поясните свои ответы.

   а. Фотопленка.

   б. Прямоугольник в видоискателе.

   в. Фотографируемая сцена.

3. При использовании перспективной проекции при каких условиях сфера в сцене не будет создавать круг на проекционной плоскости?
4. При использовании перспективной проекции может ли изображение отрезка прямой линии в каком-либо случае быть представленным на плоскости проекции изогнутым отрезком? Обоснуйте свой ответ.
5. Предположим, один конец восьмиметрового прямого шеста находится в четырех метрах от центра проекции. Кроме того, предположим, что прямая линия от центра проекции до одного конца шеста пересекает

проекционную плоскость в точке, которая находится на расстоянии одного метра от центра проекции. Если шест параллелен проекционной плоскости, то какого размера будет изображение шеста на проекционной плоскости?

6. Объясните, в чем состоит различие между параллельной проекцией и перспективной проекцией.
7. Объясните, какая связь существует между окном изображения и буфером кадра.
8. В чем состоит важное отличие применения 3D-графики для создания движущегося изображения и применения 3D-графики для создания анимации в интерактивной видеоигре? Поясните свой ответ.
9. Укажите некоторые свойства объекта, которые могут быть включены в модель этого объекта для использования в сцене 3D-графики. Укажите некоторые его свойства, которые, вероятнее всего, не будут представлены в такой модели. Поясните свой ответ.
10. Укажите некоторые физические свойства объекта, которые не фиксируются в модели, содержащей только полигональную сетку. (Следовательно, полигональная сетка сама по себе не представляет собой полной модели объекта.) Объясните, как одно из этих свойств может быть добавлено в модель объекта.
11. Могут ли любые четыре точки в трехмерном пространстве быть вершинами грани в полигональной сетке? Поясните свой ответ.
12. Каждый приведенный ниже набор чисел представляет вершины (с использованием традиционной прямоугольной системы координат) одной грани в полигональной сетке. Опишите форму этой полигональной сетки.

```
Грань 1: (0, 0, 0) (0, 2, 0) (2, 2, 0)(2, 0, 0)
Грань 2: (0, 0, 0) (1, 1, 1) (2, 0, 0)
Грань 3: (2, 0, 0) (1, 1, 1) (2, 2, 0)
Грань 4: (2, 2, 0) (1, 1, 1) (0, 2, 0)
Грань 5: (0, 2, 0) (1, 1, 1) (0, 0, 0)
```

13. Каждый приведенный ниже набор чисел представляет вершины (с использованием традиционной прямоугольной системы координат) одной грани в полигональной сетке. Опишите форму этой полигональной сетки.

```
Грань 1: (0, 0, 0) (0, 4, 0) (2, 4, 0)(2, 0, 0)
Грань 2: (0, 0, 0) (0, 4, 0) (1, 4, 1)(1, 0, 1)
Грань 3: (2, 0, 0) (1, 0, 1) (1, 4, 1)(2, 4, 0)
Грань 4: (0, 0, 0) (1, 0, 1) (2, 0, 0)
Грань 5: (2, 4, 0) (1, 4, 1) (0, 4, 0)
```

14. Создайте полигональную сетку, представляющую прямоугольное тело. Используйте традиционную прямоугольную систему координат, чтобы

закодировать вершины сетки, и нарисуйте эскиз, представляющий ваше решение.

15. Используя не более восьми треугольных граней, создайте полигональную сетку, аппроксимирующую форму сферы единичного радиуса. (При наличии только восьми граней эта полигональная сетка будет лишь очень слабым приближением к сфере, но ваша цель — показать свое понимание того, что такое полигональная сетка, а вовсе не создать достаточно точное представление сферы.) Укажите расположение вершин граней вашей полигональной сетки, используя традиционную прямоугольную систему координат, и нарисуйте ее эскиз.

16. Почему следующие четыре точки не являются вершинами плоской грани?
`(0, 0, 0) (1, 0, 0) (0, 1, 0) (0, 0, 1)`

17. Предположим, что точки `(1, 0, 0)`, `(1, 1, 1)` и `(1, 0, 2)` являются вершинами плоской грани. Какие из следующих линейных сегментов являются нормалями к поверхности этой грани?

а. Отрезок от (1, 0, 0) до (1, 1, 0)

б. Отрезок от (1, 1, 1) до (2, 1, 1)

в. Отрезок от (1, 0, 2) до (0, 0, 2)

г. Отрезок от (1, 0, 0) до (1, 1, 1)

18. Дайте определение двум "типам" процедурных моделей.

19. Сравнивая процессы моделирования и рендеринга, какой из них в каждом случае точнее отвечает приведенным ниже определениям? Обоснуйте свои ответы.

а. Стандартизированная задача.

б. Вычислительно сложная задача.

в. Творческое задание.

20. Что из следующего может быть представлено в графе сцены?

а. Источники света.

б. Неподвижный реквизит.

в. Персонажи или актеры.

г. Камера.

21. В каком смысле создание графа сцены является ключевым шагом в процессе трехмерной графики?

22. Какие сложности вносит тот факт, что камера в графе сцены может изменять свои местоположение и ориентацию?

23. Предположим, что поверхность плоской грани с вершинами `(0, 0, 0)`, `(0, 2, 0)`, `(2, 2, 0)` и `(2, 0, 0)` является гладкой и зеркальной. Если луч света исходит из точки `(0, 0, 1)` и падает на поверхность в точке `(1, 1, 0)`, то через какую из приведенных ниже точек пройдет отраженный луч?

а. `(0, 0, 1)`

б. `(1, 1, 1)`

в. `(2, 2, 1)`

г. `(3, 3, 1)`

24. Предположим, что на буйке с шестом в трех метрах над поверхностью спокойной воды горит сигнальный фонарь. В какой точке на поверхности воды наблюдатель увидит отражение этого фонаря, если наблюдение ведется из точки, находящейся в 4,5 метра от буйка и 1,5 метра над поверхностью воды?

25. Рыба плавает под поверхностью неподвижной воды, а наблюдатель над водой смотрит на нее. Где окажется изображение этой рыбы на поверхности воды относительно ее истинной позиции с точки зрения наблюдателя?

а. Выше и дальше от наблюдателя относительно своего истинного положения.

б. На своем истинном местоположении.

в. Ниже и ближе к наблюдателю относительно своего истинного положения.

26. Предположим, что точки (1, 0, 0) , (1, 1, 1) и (1, 0, 2) являются вершинами плоской грани и перечислены здесь в порядке движения против часовой стрелки, если смотреть на объект снаружи. В каждом приведенном ниже случае укажите, будет ли луч света, исходящий из указанной точки, попадать на поверхность этой грани снаружи или изнутри объекта.

а. `(0, 0, 0)`

б. `(2, 0, 0)`

в. `(2, 1, 1)`

г. `(3, 2, 1)`

27. Приведите пример, в котором объект за пределами визуального коридора все еще может появиться на конечном изображении. Поясните свой ответ.

28. Опишите содержание и назначение z-буфера.

29. При обсуждении методов удаления скрытых поверхностей была предложена процедура решения проблемы "передний план/фон" посредством использования z-буфера. Опишите эту процедуру, используя псевдокод, представленный в главе 5.

30. Предположим, что поверхность объекта покрыта чередующимися оранжевыми и синими вертикальными полосами, каждая из которых имеет ширину один сантиметр. Если объект расположен в сцене так, что положения пикселей связаны с точками на объекте, разнесенными с интервалом в два сантиметра, каковы будут возможные варианты внешнего вида объекта в конечном изображении? Поясните свой ответ.
31. Хотя оба метода — отображение текстуры и рельефное текстурирование — обеспечивают связывание "текстуры" с поверхностью, между ними имеются существенные различия. Сравните эти два метода и укажите на имеющиеся между ними различия.
32. Перечислите четыре этапа конвейера рендеринга и дайте краткое определение каждого из них.
33. Каковы некоторые преимущества, достигаемые за счет использования аппаратной реализации конвейера рендеринга или соответствующего встроенного программного обеспечения?
34. Чем аппаратное обеспечение компьютера, предназначенного для интерактивных видеоигр, отличается от аппаратного обеспечения компьютера общего назначения?
35. Что является существенным ограничением традиционного конвейера рендеринга?
36. Чем различаются между собой модель локального освещения и модель глобального освещения?
37. Какие преимущества имеет метод трассировки лучей в сравнении с традиционным конвейером рендеринга? Какие ему свойственны недостатки?
38. Какие преимущества имеет метод распределенной трассировки лучей в сравнении с традиционным методом трассировки? Какой у него есть недостаток?
39. Какие преимущества имеет метод излучательности по сравнению с традиционным конвейером рендеринга? Какие этому методу свойственны недостатки?
40. Если изображение сцены, полученное с помощью традиционного метода трассировки лучей, сравнить с аналогичным изображением той же сцены, но полученным с помощью метода излучательности, какие различия можно будет заметить на этих двух изображениях?
41. Сколько кадров потребуется для создания анимационного фильма продолжительностью в девяносто минут, если он будет демонстрироваться в кинотеатрах?

42. Опишите, как систему частиц можно использовать для создания анимации мерцающего пламени.
43. Объясните, чем может быть полезно использование z-буфера при создании последовательности кадров анимации, изображающей единственный объект, движущийся внутри сцены.
44. В чем заключаются некоторые различия между задачами, стоящими перед специалистами-аниматорами в наши дни, и задачами, стоявшими перед ними в прошлом?

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также понять, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. Предположим, что компьютерная анимация доходит до того, что в кино- и телевизионной индустрии реальные актеры больше не нужны. Каковы будут последствия? Каков будет эффект "расходящихся кругов на воде" из-за отсутствия "кинозвезд"?
2. С разработкой цифровых камер и соответствующего программного обеспечения возможность изменять или подделывать фотографии стала доступной для широкой общественности. Какие изменения это принесло обществу? Какие этические и правовые вопросы уже возникли или еще могут возникнуть?
3. В какой степени фотографии должны считаться собственностью? Предположим, что человек размещает свою фотографию на веб-сайте, а кто-то ее загружает и изменяет так, чтобы объект оказался в компрометирующей ситуации, после чего распространяет измененную версию в популярном выпуске социальных сетей. Какие права обратиться за помощью в суд должен иметь тот, кто изображен на фотографии? Что если человек на оригинальной фотографии представляет организацию, пользующуюся большим доверием и авторитетом в обществе? Какие возможности правовой защиты должны быть у этой организации или у общественности?

4. В какой степени программист, который помогает разрабатывать видеоигры со сценами жестокости, ответственен за любые последствия, вызванные появлением этой игры? Следует ли ограничивать доступ детей к видеоиграм? Если да, то как и кем? А как насчет других групп в обществе, таких как осужденные преступники?



Основные положения для запоминания

- **Компьютерная программа или результаты запуска программы могут быть быстро переданы большому количеству пользователей и могут оказать существенное влияние на отдельных лиц, организации и общество.**

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Angel E., Shreiner D. *Interactive Computer Graphics. A Top-Down Approach with Shader-Based OpenGL*. 6th ed. — Boston, MA: Addison-Wesley, 2011. (Имеется русский перевод 2-го издания этой книги: Энджел Э. *Интерактивная компьютерная графика. Вводный курс на базе OpenGL*, 2-е изд. — М.: Издательский дом "Вильямс", 2001.)
2. Bowman D.A., Kruijff E., LaViola J.J., Poupyrev I. *3D User Interfaces. Theory and Practice*. — Boston, MA: Addison-Wesley, 2005.
3. Hill F.L., Kelley S. *Computer Graphics Using OpenGL*. 3rd ed. — Upper Saddle River, NJ: Prentice-Hall, 2007.
4. McConnell J.J. *Computer Graphics. Theory into Practice*. — Sudbury, MA: Jones and Bartlett, 2006.
5. Parent R. *Computer Animation. Algorithms and Techniques*. 3rd ed.: — San Francisco, CA: Morgan Kaufmann, 2012.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Джамбруно М. *Трехмерная (3D) графика и анимация*, 2-е изд. — М.: Издательский дом "Вильямс", 2002.
2. Берн Дж. *Цифровое освещение и визуализация*. — М.: Издательский дом "Вильямс", 2003.
3. Ратнер П. *Трехмерное (3D) моделирование и анимация человека*, 2-е изд. — М.: Издательский дом "Вильямс", 2005.

4. Херн Д., Бейкер М.П. *Компьютерная графика и стандарт OpenGL*, 3-е изд. — М.: Издательский дом "Вильямс", 2005.
5. Буске Мишель. *3D Моделирование, снаряжение и анимация персонажей в Autodesk 3ds Max 7*. — М.: Издательский дом "Вильямс", 2006.
6. Брукер Д. *Освещение в трехмерной графике с использованием 3ds Max*, 2-е изд. — М.: Издательский дом "Вильямс", 2008.
7. Осипа Дж. *3D-моделирование и анимация лица: методики для профессионалов*, 2-е изд. — М.: Издательский дом "Вильямс", 2007.
8. Мэрдок К. *Autodesk 3ds Max 2013. Библия пользователя*. — М.: ООО "И.Д. Вильямс", 2013.

Глава 11

В этой главе вы познакомитесь с разделом компьютерных наук, связанным с исследованиями в области создания искусственного интеллекта. Хотя эта область относительно молода, в ней уже достигнуты некоторые удивительные результаты, такие как компьютерные программы, играющие роль противника в компьютерных играх, компьютеры, которые, как кажется, успешно учатся и рассуждают, а также машины, координирующие свою деятельность для достижения общей цели, такой как победа в игре в футбол. В области искусственного интеллекта современная научная фантастика вполне может стать реальностью завтрашнего дня.

Искусственный интеллект

11.1. ИНТЕЛЛЕКТ И МАШИНЫ
- Интеллектуальные агенты
- Методологии исследований
- Тест Тьюринга

11.2. СПОСОБНОСТЬ К ВОСПРИЯТИЮ
- Распознавание изображений
- Обработка языка

11.3. СПОСОБНОСТЬ К РАССУЖДЕНИЮ
- Порождающие системы
- Дерево поиска
- Эвристические методы

11.4. ДОПОЛНИТЕЛЬНЫЕ ОБЛАСТИ ИССЛЕДОВАНИЙ
- Представление знаний и манипулирование ими
- Машинное обучение
- Генетические алгоритмы

11.5. ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ
- Основные свойства
- Обучение искусственных нейронных сетей

11.6. РОБОТОТЕХНИКА

11.7. ОСМЫСЛИВАНИЕ ПОСЛЕДСТВИЙ

Искусственный интеллект — это область компьютерных наук, основная задача которой заключается в отыскании способов создания автономных машин, т.е. таких машин, которые будут способны выполнять сложные задания без вмешательства человека. Такая цель требует, чтобы машины были способны самостоятельно воспринимать окружающий мир и выполнять необходимые рассуждения. Подобные возможности подпадают под категорию той деятельности, которую мы привыкли называть здравым смыслом, явлением, совершенно естественным для человеческого разума, но, несомненно, весьма трудно достижимым для машин. В результате работа по проведению соответствующих исследований продолжает оставаться сложной задачей. В данной главе будут рассмотрены некоторые важные темы в этой обширной области исследований.

11.1. Интеллект и машины

Область искусственного интеллекта довольно обширна и в разных направлениях сливается с другими предметами, такими как психология и неврология, математика и лингвистика, электрическая и механическая инженерия. Чтобы сфокусировать свои мысли на этой обширной области, мы начнем с рассмотрения концепции агента и типов интеллектуального поведения, которое может проявлять агент. И действительно, бо́льшую часть исследований, проводимых в области искусственного интеллекта, можно классифицировать в терминах поведения агента.

Интеллектуальные агенты

Агент — это "устройство", которое реагирует на раздражители из окружающей его среды. Естественно представить агента как отдельную машину, такую как робот, хотя агент может принимать и другие формы, такие как автономный самолет, персонаж в интерактивной видеоигре или процесс, взаимодействующий с другими процессами через Интернет (возможно, как клиент, сервер или узел в одноранговой сети). У большинства агентов есть датчики, с помощью которых они получают данные из окружающей среды, и приводы, с помощью которых они могут оказывать некоторое влияние на окружающую их среду. Примерами датчиков являются микрофоны, видеокамеры, дальномеры и устройства для отбора проб воздуха или почвы. Примерами приводов являются колеса, ноги, крылья, захваты и синтезаторы речи.

Большая часть исследований в области искусственного интеллекта может быть охарактеризована в контексте построения агентов, которые ведут себя разумно в том смысле, что действия исполнительных механизмов этих агентов будут представлять собой рациональную реакцию на данные, получаемые

через их датчики. В свою очередь, появляется возможность классифицировать проводимые исследования, рассматривая различные уровни этих ответных реакций.

Самым простым ответом будет рефлекторное действие, которое является просто предопределенным ответом на входные данные. Для получения более "интеллектуального" поведения необходимы более высокие уровни ответных реакций. Например, можно наделить агента знаниями об окружающей его среде и потребовать, чтобы агент соответствующим образом корректировал свои действия. Процесс броска бейсбольного мяча в основном является рефлекторным действием, но для определения того, как и куда бросить мяч, необходимо знание текущей обстановки, которая раз от раза может существенно изменяться. Как такие знания об окружающем реальном мире могут быть сохранены, обновлены, извлечены и в конечном итоге применены в процессе принятия решений, остается сложным вопросом в области искусственного интеллекта.

Совсем иной уровень ответа потребуется в тех случаях, когда мы хотим, чтобы агент стремился к такой цели, как победа в игре в шахматы или маневрирование в переполненном проходе. Такое целенаправленное поведение требует, чтобы ответ или последовательность ответов агента были результатом сознательного формулирования плана действий или выбора наилучшего действия среди текущих доступных вариантов.

В некоторых случаях ответы агента могут улучшаться с течением времени. Этот процесс может принимать форму развития **процедурных знаний** (обучение "как" — *алгоритмические* знания) или накопления **декларативных знаний** (обучение "что" — *описательные* знания). Получение процедурных знаний обычно включает в себя процесс проб и ошибок, посредством которого агент отбирает подходящие действия, будучи наказанным за плохие, не подходящие случаю действия и вознагражденным за действия хорошие, успешно приводящие к поставленной цели. Следуя этому подходу, были разработаны агенты, которые со временем улучшают свои способности в соревновательных играх, таких как шашки и шахматы. Изучение декларативных знаний обычно принимает форму накопления, расширения или изменения "фактов" в хранилище знаний агента. Например, игрок в бейсбол должен многократно корректировать свою базу данных знаний о возможных ситуациях в игре, на основании которых в дальнейшем он будет отыскивать рациональные ответы на происходящие события.

Чтобы давать рациональные ответы на стимулы, агент должен "понимать" ту информацию, которая поступает через его сенсоры. А это означает, что агент должен быть способен извлечь важные сведения из общего потока данных, поступающего от его датчиков, или, другими словами, агент должен уметь *воспринимать*. В некоторых случаях это довольно простой процесс. Сигналы,

полученные от гироскопа, легко кодируются в формах, совместимых с расчетами, необходимыми для определения соответствующих ответов. Однако в других случаях извлечение информации из входных данных бывает достаточно трудной задачей. К этой категории можно отнести понимание речи и анализ изображений. Аналогичным образом агент должен быть способен сформулировать свои ответы в терминах, совместимых с используемыми исполнительными механизмами. И вновь, это может быть очень простой процесс, но может и потребоваться, например, чтобы агент формулировал свои ответы как законченные устные предложения, а это означает, что он должен генерировать человеческую речь. А отсюда следует, что такие темы, как обработка и анализ изображений, понимание естественного языка и генерация речи, являются важными областями исследований.

Все атрибуты агента, которые мы определили выше, представляют прошлые и нынешние области исследований. Конечно же, они не являются полностью независимыми друг от друга. Хотелось бы разрабатывать агенты, которые обладают ими всеми сразу, получая агенты, понимающие данные, поступающие из их окружения, и разрабатывающие новые модели реагирования за счет процесса обучения, целью которого является максимизация возможностей агента. Тем не менее, отделяя различные типы рационального поведения и анализируя их независимо, исследователи получают ту опору, которая впоследствии может быть объединена с результатами, достигнутыми за счет прогресса в других областях, с целью создания все более и более интеллектуальных агентов.

Мы закончим этот подраздел введением агента, который обеспечит контекст для последующего обсуждения в разделах 11.2 и 11.3. Этот агент предназначен для решения головоломки "Восьмерка", представляющей собой восемь квадратных фишек, помеченных цифрами от 1 до 8. Эти фишки уложены в рамку, способную вместить в общей сложности девять таких фишек, размещенных в три ряда по три штуки (рис. 11.1). Девятая позиция в рамке свободна и в нее может быть перемещена любая из расположенных рядом фишек, что позволяет свободно перемещать фишки в коробке. Задача состоит в том, чтобы переместить произвольным образом помещенные в рамку фишки в их исходное положение, как показано на рис. 11.1.

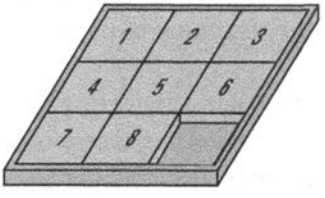


Рис. 11.1. Головоломка "Восьмерка" с расположением фишек, соответствующим правильному решению

Наш агент будет иметь вид металлической коробки, снабженной зажимом, видеокамерой и манипулятором с резиновым наконечником, необходимым для того, чтобы манипулятор не проскальзывал при толкании чего-либо (рис. 11.2). Как только агент включают, его зажим начинает открываться и закрываться, как будто требуя дать ему головоломку. Когда головоломка "Восьмерка" со случайным образом переставленными фишками помещается в зажим, агент захватывает ее и приступает к работе. Манипулятор машины опускается и начинает методично перемещать фишки внутри рамки головоломки до тех пор, пока они не вернутся в изначальное положение, как показано на рис. 11.1. В этот момент зажим отпускает головоломку и машина выключается.

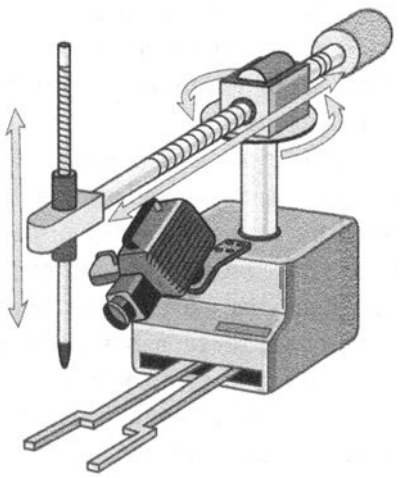


Рис. 11.2. Машина для решения головоломки "Восьмерка"

Данная машина для решения головоломки "Восьмерка" имеет два атрибута агента, идентифицированных нами выше. Во-первых, агент должен быть способен воспринимать происходящее в том смысле, что он должен извлекать информацию о текущем состоянии головоломки, исходя из того изображения, которое он получает от своей камеры. (Подробно проблема распознавания изображений будет обсуждаться разделе 11.2.) Во-вторых, агент должен уметь разрабатывать, а затем воплощать в жизнь план достижения поставленной перед ним цели. (Эти вопросы будут рассматриваться в разделе 11.3.)

Методологии исследований

Чтобы оценить значимость области искусственного интеллекта, очень важно уяснить, что исследования в этой области ведутся по двум основным направлениям. Первым из них является инженерный подход, при котором

исследователи видят свою основную задачу в разработке систем, демонстрирующих то или иное интеллектуальное поведение. Другое направление — теоретические изыскания, когда исследователи пытаются отыскать способы вычислительного представления различных характеристик интеллекта животных и, прежде всего, человека. Это раздвоение становится еще более очевидным при рассмотрении способов, которыми ведутся исследования в каждом из этих двух направлений. Инженерный подход ведет к методологии, ориентированной на демонстрацию определенных действий, поскольку основная цель здесь состоит в создании продукта, демонстрирующего поведение, соответствующее тем или иным поставленным целям. Теоретический подход ведет к методологии, ориентированной на моделирование, поскольку основная цель исследований состоит в том, чтобы расширить наше понимание интеллекта, а это значит, что акцент здесь делается на самих лежащих в его основе процессах, а не на демонстрации их внешних проявлений.

В качестве примера рассмотрим области обработки естественного языка и лингвистики. Эти области тесно связаны и извлекают выгоду из исследований, проводимых в каждой из них, но основные их цели различны. Лингвисты заинтересованы в изучении того, как люди пользуются языком, и следовательно, стремятся к более теоретическим изысканиям. Исследователи в области обработки естественного языка заинтересованы в разработке машин, которые будут способны манипулировать естественным языком, и, следовательно, они придерживаются преимущественно инженерного направления. Таким образом, лингвисты работают в режиме, ориентированном на моделирование, т.е. занимаются построением систем, целью которых является проверка тех или иных теоретических положений. И напротив, исследователи в области обработки естественного языка работают в режиме, ориентированном на демонстрацию, т.е. занимаются созданием систем, предназначенных для выполнения тех или иных задач. Системы, создаваемые в последнем случае (например, системы автоматического перевода или системы, с помощью которых машины реагируют на словесные команды), в значительной степени полагаются на знания, полученные лингвистами, но в них часто применяются некоторые упрощения, в результате чего эти системы могут успешно функционировать лишь в конкретном, четко ограниченном окружении.

В качестве элементарного примера рассмотрим задачу разработки оболочки для операционной системы, которая получает инструкции из внешнего мира посредством устных команд на английском языке. В данном случае оболочке (агенту) не требуется эффективно владеть английским языком в его полном объеме. Скажем, агенту-оболочке нет необходимости различать все возможные значения слова *copy*. (Является это слово существительным или же это глагол? Несет ли оно дополнительный смысл как указание на плагиат?) Вместо

этого оболочка должна просто уметь отличать слово *copy* (скопировать) от других команд, таких как *rename* (переименовать) или *delete* (удалить). При таком подходе оболочка может успешно выполнять свою задачу, просто сопоставляя поступающие входные сигналы с заранее заданными звуковыми шаблонами. Поведение, демонстрируемое такой системой, может быть абсолютно удовлетворительным с точки зрения инженера, но способ, которым оно достигается, едва ли будет эстетически приятен для теоретика.

Тест Тьюринга

В прошлом **тест Тьюринга** (он был предложен Аланом Тьюрингом в 1950 году как средство оценки интеллектуальных возможностей машины) служил важным показателем в измерении прогресса в области искусственного интеллекта. Сегодня значение теста Тьюринга уже не столь велико, но он все еще остается важной частью фольклора в области искусственного интеллекта. Суть теста состоит в следующем: позволить человеку (назовем его опросчиком) общаться с тестируемым субъектом посредством терминала, не уточняя при этом, кем является субъект — человеком или машиной. В данной ситуации поведение машины будет считаться интеллектуальным, если она будет высказываться о событиях настолько разумно, что опросчик не сможет отличить ее от человека. Очевидно, что тест Тьюринга оценивает уровень соответствия поведения машины по отношению к поведению человека. Тьюринг предполагал, что к 2000 году машины смогут с 30-процентной вероятностью продержаться пять минут. Как показало время, эта догадка оказалась на удивление точной.

Происхождение искусственного интеллекта

Исследования по созданию машин, имитирующих человеческое поведение, имеют долгую историю, но большинство авторов соглашаются с тем, что современный этап в исследованиях искусственного интеллекта начался в 1950 году. В этот год Алан Тьюринг опубликовал статью "Computing Machinery and Intelligence", в которой он впервые предположил, что возможно создание машин, запрограммированных для демонстрации разумного поведения. Однако название данной области науки — "искусственный интеллект" — впервые было употреблено лишь несколько лет спустя, в теперь уже ставшем легендарном проекте Джона Мак-Карти (John McCarthy). В нем говорилось о том, что "изучение искусственного интеллекта будет проводиться в течение лета 1956 года в Дартмутском колледже" с целью проверить "гипотезу о том, что каждый аспект механизма обучения или любой другой функции сознания может быть описан настолько точно, что это позволит создать машину для его воспроизведения".

Одна из причин, по которым тест Тьюринга больше не считается значимым показателем интеллекта, заключается в том, что впечатляющее проявление интеллекта может быть **осуществлено** с относительной легкостью. Хорошо известный пример применения сценария теста Тьюринга связан с испытаниями программы DOCTOR (версия более общей системы под названием ELIZA), разработанной в середине 1960-х годов Джозефом Уэйзенбаумом (Joseph Weizenbaum). Эта интерактивная программа была сконструирована для имитации поведения психиатра Рогерианской (Rogerian) школы психоанализа, проводящего психологический опрос. Компьютер играл роль психиатра, а роль пациента была отведена пользователю. В сущности, все выполняемые программой DOCTOR действия сводились к реструктуризации получаемых от собеседника выражений, проводимой по нескольким хорошо известным правилам. Измененные выражения возвращались на экран пользователя в виде ответных реплик. Например, в ответ на выражение "Я сегодня устал" программа DOCTOR могла выдать пользователю следующее сообщение: "Как вы думаете, из-за чего вы сегодня устали?" Если программа не могла распознать структуру полученного сообщения, она просто выводила пользователю малозначащую общую фразу, например "Продолжайте, продолжайте..." или "Очень интересно".

Первоначальная цель Уэйзенбаума при разработке программы DOCTOR состояла в проведении исследований природы языкового общения. С этой точки зрения выбор психотерапии играл второстепенную роль как средства создания нужной окружающей обстановки (или предмета беседы), в которой программа могла бы функционировать. К удивлению Уэйзенбаума, несколько психологов предложили использовать его программу для проведения реальной психотерапии. (Главным тезисом Рогерианской школы является утверждение, что не психиатр, а пациент должен задавать направление беседы в течение терапевтического сеанса. Именно поэтому, как утверждали психологи, компьютер будет вполне способен поддерживать беседу так же хорошо, как и профессиональный терапевт.) Кроме того, программа DOCTOR настолько хорошо имитировала понимание ситуации, что многие "общавшиеся" с ней пациенты чувствовали себя установившими контакт с личностью, имеющей родственные мысли и чувства, и в большинстве случаев, как ни странно, подчинялись машинному диалогу, состоящему из вопросов и ответов. В этом смысле программа DOCTOR прошла тест Тьюринга. В результате возникли новые этические и технические проблемы, а сам Уэйзенбаум стал активным защитником человеческого достоинства в мире прогрессирующих технологий.

Среди более поздних примеров "успешного" выполнения теста Тьюринга можно упомянуть интернет-вирусы, ведущие "интеллектуальные" диалоги с человеком-жертвой с целью обманным путем заставить его отказаться от использования средств защиты от вредоносных программ. Более того, явления, подобные тестам Тьюринга, часто возникают в контексте компьютерных игр,

таких как шахматные программы. Хотя эти программы выбирают ходы, просто применяя методы грубой силы (подобные тем, которые будут обсуждаться в разделе 11.3), люди, играющие с компьютером, часто испытывают ощущение, что машина обладает творческим потенциалом и даже индивидуальностью. Подобные ощущения возникают и в робототехнике, где были созданы машины с физическими атрибутами, демонстрирующими интеллектуальные свойства. Примеры включают игрушечных собак-роботов, которые имитируют трогательное личностное поведение, просто наклоняя голову или поднимая уши в ответ на звуковые сигналы.

11.1. Вопросы и упражнения

1. Укажите несколько типов "интеллектуальных" действий, которые может выполнять агент.
2. Растение, помещенное в темную комнату с единственным источником света, растет по направлению к нему. Является ли такое поведение разумным? Обладает ли растение интеллектом? Каково ваше определение интеллекта?
3. Предположим, что торговый автомат предназначен для отпуска различных товаров в зависимости от того, какая кнопка нажата. Как вы считаете, можно ли сказать, что этот автомат "осознает", какая кнопка была нажата? Каково ваше определение осознанности?
4. Если машина прошла тест Тьюринга, согласитесь ли вы с тем, что она разумна? Если нет, то согласитесь ли вы признать, что она кажется разумной?
5. Предположим, что вы, используя программу обмена текстовыми сообщениями или интерактивный сайт в социальной сети, в течение десяти минут вели содержательный последовательный разговор с неким абонентом. Если позднее выяснится, что вы общались с машиной, то сможете ли вы сделать вывод, что эта машина была разумной? Почему да или почему нет?

11.2. Способность к восприятию

Чтобы разумно реагировать на данные, поступающие от его датчиков, агент должен уметь понимать эти данные. Иначе говоря, агент должен уметь их правильно воспринимать. В этом разделе будут рассмотрены две области

исследований, связанных с восприятием, которые на практике оказались особенно сложными: распознавание изображений и понимание языка.

Распознавание изображений

Давайте обратимся к проблемам, связанным с реализацией машины для решения головоломки "Восьмерка", представленной в предыдущем разделе. Открытие и закрытие зажима нашей машины не представляет собой серьезной проблемы. Обнаружить в зажиме рамку с головоломкой по ходу этого процесса также достаточно просто, поскольку эта прикладная задача не требует большой точности. С проблемой фокусировки камеры на головоломке также легко можно справиться, сконструировав зажим таким образом, чтобы рамка с головоломкой всегда находилась по отношению к камере под определенным углом зрения. Следовательно, первым проявлением разумного поведения, необходимым для нашей машины, является способность получения информации посредством оптических средств.

Важно понимать, что проблема, с которой сталкивается машина при взгляде на головоломку, — это вовсе не просто получение и запоминание изображения. Технически решение подобных задач возможно уже многие годы, например с помощью обычной фотографии или телевидения. В действительности проблема заключается в *понимании* изображения, причем на таком уровне, который позволит извлечь из него сведения о текущем состоянии головоломки (а позднее контролировать перемещение фишек).

В случае нашей машины для решения головоломок существует относительно немного вариантов возможных изображений. Мы предполагаем, что в поле зрения машины всегда находится изображение головоломки, содержащей фишки с целыми числами от 1 до 8, причем в виде хорошо организованной структуры. Проблема, по сути, состоит только в том, чтобы установить взаимное расположение этих чисел. Для решения поставленной задачи представим себе, что изображение головоломки передается машине в закодированном виде — как состояние битов компьютерной памяти, где каждый бит, называемый пикселем, определяет уровень яркости отдельной части картины. Если предположить, что изображение всегда имеет один и тот же размер (головоломка помещается в одно и то же место непосредственно перед камерой), то наша машина сможет определить, какую позицию занимает каждая из фишек головоломки, сравнивая различные участки изображения с предварительно записанными шаблонами. Эти шаблоны представляют собой комбинации битов, соответствующие отдельным нумерованным фишкам головоломки. Как только соответствие между изображением и эталоном установлено, состояние головоломки считается определенным.

Этот метод распознавания изображений является одним из тех, которые используются в оптических считывающих устройствах. Однако он имеет существенный недостаток, заключающийся в требовании определенного единообразия в стиле представления, размере и ориентации символов по отношению к считывающему устройству. В частности, битовый образ физически большего по размеру символа будет отличаться от битового образа того же символа, но меньшего размера, даже если символы имеют в точности одну и ту же форму. А теперь представьте себе, насколько усложняется проблема при обработке рукописного текста.

Другой подход к проблеме распознавания символов основан на определении соответствий в геометрических характеристиках, а не на сравнении образа с точным представлением символа. В этом случае характерной особенностью числа 1 будет одна вертикальная линия, число 2 может характеризоваться незамкнутой изогнутой линией, соединенной внизу с прямой горизонтальной линией, и т.д. Этот метод распознавания символов включает два этапа: первый состоит в определении характерных особенностей изображения, а второй — в сравнении найденных особенностей с характеристиками эталонных изображений. По отношению к методу сравнения шаблонов этот метод распознавания символов значительно сложнее. Например, даже незначительные ошибки в изображении символов могут привести к получению набора совершенно различных геометрических показателей, что весьма затрудняет распознавание близких по начертанию символов, например O и C или, как в случае с нашей головоломкой, цифр 3 и 8.

Можно считать, что нам еще повезло с нашей головоломкой, поскольку здесь не требуется распознавание изображений в общей трехмерной картине. Действительно, то, что образы, которые необходимо распознать (числа от 1 до 8), изолированы в различных частях общей картины, существенно упрощает дело, поскольку это позволяет исключить из рассмотрения возможность появления перекрывающихся изображений, что достаточно типично в случае трехмерных изображений. На обычной фотографии, например, машина сталкивается не только с проблемой распознавания объекта, вид которого может быть показан с различных точек зрения, но и с тем фактом, что некоторые части предмета могут быть вообще скрыты от глаз.

Задача распознавания произвольных изображений обычно решается в два этапа. На первом выполняется **обработка изображения**, направленная на идентификацию характерных особенностей изображения, а на втором проводится **анализ изображения**, т.е. предпринимаются попытки понять, что эти особенности означают. Выше мы уже рассматривали это противопоставление в контексте распознавания символов через особенности их графического воспроизведения. Можно сказать, что в этом случае обработка изображения

представлена процессом идентификации геометрических особенностей изображения, а анализ изображения — процессом идентификации смысла найденных особенностей.

Процесс обработки изображения состоит из многочисленных составляющих. Одной из них является усиление контуров, т.е. применение математических методов для уточнения границ между отдельными частями изображения. Конечно же, в каком-то смысле усиление контуров — это попытка превращения фотографии в чертеж. Другая составляющая процесса анализа изображения состоит в нахождении областей, т.е. определении фрагментов изображения, которые имеют такие общие свойства, как яркость, цвет или текстуру. Эти области, как правило, представляют часть изображения, принадлежащую одному и тому же объекту. (С помощью метода распознавания областей компьютер может раскрашивать мультфильмы и старые черно-белые фильмы.) Еще одной процедурой в процессе обработки изображения является сглаживание, представляющее собой удаление мелких дефектов изображения. Сглаживание не позволяет мелким дефектам изображения оказывать нежелательное влияние на выполнение других процедур обработки изображения. Однако слишком сильное сглаживание может привести к уничтожению важной информации об объектах.

Сильный ИИ против слабого ИИ

Предположение о том, что машины могут быть запрограммированы для демонстрации разумного поведения, известно как **слабый ИИ** (ИИ — искусственный интеллект) и в той или иной степени уже принято широкой аудиторией. Однако предположение, что машина может быть запрограммирована на обладание разумом и фактически искусственным сознанием, известное как **сильный ИИ**, все еще активно дискутируется. Противники сильного ИИ приводят доводы, что машина радикально отличается от человека и, таким образом, никогда не испытает любви, не отличит правду от лжи и не будет думать о себе так, как это делает человек. Однако сторонники сильного ИИ возражают, что человеческий мозг состоит из мельчайших элементов, каждый из которых по отдельности человеком не является и таковым себя не осознает, но все вместе те же элементы формируют личность. Почему же, спрашивают они, то же явление не может происходить и с машинами?

Проблема разрешения споров по проблеме сильного ИИ состоит, как уже отмечалось, в том, что вещи, подобные разуму и сознанию, являются внутренними характеристиками и не могут быть прямо определены. Как указал еще Алан Тьюринг, мы верим, что люди обладают разумом, поскольку они ведут себя разумно, хотя мы и не можем наблюдать их внутреннее умственное состояние. Готовы ли мы проиллюстрировать подобный подход и по отношению к машине, если она продемонстрирует внешние признаки сознания? Почему да или почему нет?

Сглаживание, усиление контуров и нахождение областей — это отдельные этапы процесса идентификации различных частей изображения, тогда как анализ изображения позволяет определить, что представляют собой данные части и в конечном счете что означает все изображение в целом. Здесь мы сталкиваемся с задачей распознавания частично заслоненных объектов, показанных с различных ракурсов. Один из методов анализа изображений заключается в выборе некоторого исходного предположения, чем должен оказаться изображенный объект, с последующей попыткой связать части изображения с предполагаемым объектом. Похоже, что именно так подходит к данной проблеме сам человек. Например, мы часто не в состоянии определить незнакомый предмет в условиях плохой видимости, но стоит нам получить подсказку, на что этот объект может быть похож, как мы легко его распознаем.

Распознавание произвольных изображений связано со многими сложнейшими проблемами, и множество из них еще ждет своего решения. Однако большие успехи, уже достигнутые в анализе изображений, способствовали значительному прогрессу в самых различных областях — от создания роботизированных пылесосов до успешного внедрения автоматической сортировки почты. Однако в этом направлении компьютерных наук предстоит провести еще много исследований. И действительно, анализ изображений — это одна из тех областей, в которых убедительно демонстрируется, как задачи, которые решаются человеческим сознанием быстро и, по-видимому, легко, продолжают бросать вызов возможностям машин.

Обработка языка

Другой проблемой восприятия, которая оказалась весьма сложной, является проблема понимания языка. Успехи, достигнутые в переводе формальных языков программирования высокого уровня на машинный язык (раздел 6.4), привел первых исследователей в этой области к убеждению, что реализация возможности программировать компьютеры для понимания естественного языка является вопросом всего лишь нескольких лет. И действительно, способность компьютеров с легкостью осуществлять перевод программ с одного языка программирования на другой создает иллюзию, что машина действительно понимает переводимый язык. (Вспомните из раздела 6.1 историю, рассказанную Грейс Хоппер о менеджерах, которые думали, что она учит компьютеры понимать немецкий язык.)

Тогда исследователи еще не могли понять всей глубины отличий, существующих между формальными языками программирования и естественными языками, такими как английский, немецкий или русский. Языки программирования построены из хорошо разработанных примитивов, так что каждая

инструкция заключает единственную грамматическую структуру и имеет только одно значение. Напротив, выражение на естественном языке часто может иметь несколько разных значений в зависимости от контекста или даже способа, которым оно передается. Таким образом, для понимания естественного языка люди в значительной степени полагаются на дополнительные знания.

Например, рассмотрим следующие два предложения:

Норман Роквелл изображал людей.

и

У Золушки есть коса.

В обоих случаях корректный перевод не может быть выполнен простым переводом каждого отдельного слова. В действительности, чтобы понять эти предложения, требуется способность понимать контекст, в котором эти предложения присутствуют. В некоторых случаях истинное значение предложения вообще не совпадает с его буквальным переводом. Вот характерный пример:

Ты знаешь, который час?

Чаще всего эта фраза означает "Скажите, пожалуйста, который час?" Однако если говорящий к этому моменту уже достаточно долго ожидал того, кому адресован этот вопрос, то данная фраза может означать совсем другое: "Ты сильно опоздал!"

Таким образом, определение смысла высказывания на естественном языке требует нескольких уровней анализа. Первый — это **синтаксический анализ**, основной составляющей которого является грамматический разбор. Его проведение покажет, что в следующем предложении подлежащим является слово *Мери*:

Мери подарила Джону открытку ко дню рождения.

Аналогично в следующем предложении подлежащим будет слово *Джон*:

Джон получил от Мери открытку ко дню рождения.

Второй уровень анализа высказываний называется **семантическим анализом**. В отличие от грамматического анализа, при котором просто определяется грамматическая роль каждого слова в предложении, в результате семантического анализа требуется установить семантическую роль каждого слова. Цель семантического анализа — определение описываемого действия, агента действия (которым может быть, а может и не быть подлежащее этого предложения) и объекта действия. В результате семантического анализа может быть установлено, что предложения "Мери подарила Джону открытку ко дню рождения" и "Джон получил от Мери открытку ко дню рождения" сообщают об одном и том же.

Третьим уровнем анализа высказываний является **контекстный анализ**. Именно на этом уровне выявляется действительный смысл высказывания. Например, достаточно просто определить грамматическую роль каждого слова в следующем предложении:

Он прикоснулся к бабочке.

Мы даже можем выполнить его семантический анализ, определив действие — прикосновение, агента действия — бабочку и т.д. Но пока мы не знаем контекста этого высказывания, его действительный смысл остается неясным. Действительно, данное высказывание может иметь совершенно разный смысл в контексте сборов на званый ужин и в контексте прогулки на природе. Точно так же, только на контекстном уровне можно установить истинное значение вопроса "Ты знаешь, который час?"

Следует отметить, что различные уровни анализа — синтаксический, семантический и контекстный — не обязательно являются независимыми. Рассмотрим следующее предложение на английском языке, которое может быть переведено в двух вариантах:

Stampeding cattle can be dangerous.

(*Паникующий крупный рогатый скот может быть опасен* или *Пугать крупный рогатый скот может быть опасно.*)

В первом случае подлежащим этого предложения является существительное *cattle* (скот), уточненное прилагательным *stampeding* (паникующий), в контексте разговора о стаде, по какой-то причине впавшем в панику. Во втором случае в этом же предложении подлежащим будет уже отглагольное существительное (герундий) *stampeding* (пугание) с объектом действия *cattle* (скот), — в контексте разговора с человеком, который собирается испугать стадо коров, скажем, ради развлечения. Таким образом, в зависимости от контекста одно и то же предложение может иметь два варианта грамматического анализа.

Еще одно направление исследований в области обработки естественного языка касается всего документа, а не отдельных предложений. Здесь анализируемые проблемы можно разделить на две категории: **информационный поиск** и **извлечение информации**. Информационный поиск — это задача идентификации документов, имеющих отношение к интересующей проблеме. Примером является ситуация, с которой сталкиваются пользователи Интернета при попытке найти сайты, относящиеся к определенной теме. Используемый в данный момент подход заключается в поиске сайтов по ключевым словам, но это часто приводит к целой лавине ложных ссылок, а действительно интересный сайт при этом может быть пропущен, потому что имеет дело с "легковыми машинами" вместо "автомобилей". Очевидно, что нам необходим механизм

поиска, способный понять *содержание* рассматриваемых сайтов. Сложность в достижении такого понимания является той причиной, по которой многие обращаются к таким более сложным методам, как использование XML для создания семантической сети, обсуждавшейся в разделе 4.3.

Извлечение информации заключается в выборке из документов данных с приданием им такой формы, которая сделает эти данные полезными и для использования в других приложениях. Это может означать поиск ответа на конкретный вопрос или запись информации в такой форме, которая позже позволит получить ответы на различные вопросы. Одним из типов подобных форм являются **шаблоны**. По сути, это просто анкета, в которой фиксируются некоторые характеристики. Например, рассмотрим систему для чтения газет. Эта система может использовать набор шаблонов, по одному для каждого из возможных типов газетных статей. Если система опознает статью как сообщение о грабеже, она попытается заполнить строки шаблона, относящегося к грабежам. Данный шаблон, вероятно, будет запрашивать информацию о месте, времени и дате грабежа, о взятых вещах и т.п. В то же время, если система определит статью как сообщение о стихийном бедствии, она будет заполнять соответствующий шаблон, который заставит систему определить тип стихийного бедствия, причиненный ущерб и т.п.

Другой тип формы, в которую может записываться информация, предназначенная для последующего извлечения, называется **семантической сетью**. В сущности, это большая взаимосвязанная структура данных, в которой для представления взаимосвязей между элементами данных используются указатели. На рис. 11.3 приведена часть семантической сети, в которой выделена информация, полученная из следующего высказывания:

Мери ударила Джона.

Искусственный интеллект на вашей ладони

Методы искусственного интеллекта все чаще проявляют себя в приложениях для смартфонов. Например, корпорация Google разработала Google Goggles — приложение для смартфонов, предоставляющее собой визуальный поисковый движок. Просто сделайте снимок книги, ориентира или знака с помощью фотокамеры смартфона, и приложение Goggles выполнит обработку полученного изображения, анализ этого изображения и распознавание текста, а затем запустит поиск в Интернете для идентификации данного объекта. Приложение Google Translate может записывать произнесенные слова на одном языке, переводить их на другой язык и проговаривать для вас этот перевод, работая со многими распространенными сочетаниями языков. Смартфоны, несомненно, станут еще умнее, поскольку ИИ и в дальнейшем будет широко использоваться в различных инновациях.

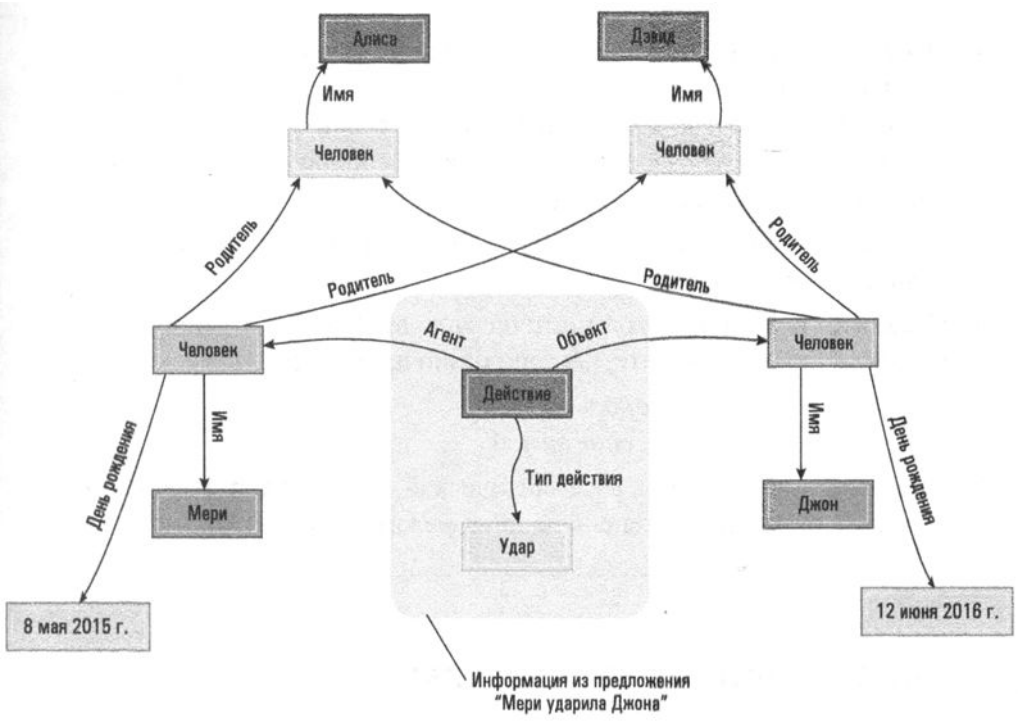
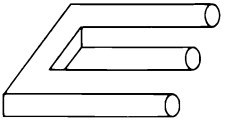


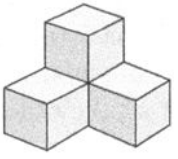
Рис. 11.3. Семантическая сеть

11.2. Вопросы и упражнения

1. Как изменятся требования к видеосистеме робота, если изображения, используемые им для самоконтроля, потребуется посылать человеку, удаленно контролирующему поведение робота?
2. Что подсказывает вам, что следующий рисунок не имеет смысла? Как понимание этого факта может быть запрограммировано в машине?



3. Сколько кубиков изображено на рисунке? Как запрограммировать машину для получения точного ответа на подобный вопрос?



4. Откуда вы знаете, что утверждения "Нет ничего лучше, чем полное счастье" и "Тарелка с холодным супом лучше, чем ничего" не дают оснований прийти к заключению, что "Тарелка с холодным супом лучше, чем полное счастье"? Как ваша способность обнаружить подобные отличия может быть перенесена на машину?
5. Определите двусмысленность, имеющую место при переводе фразы "Они гоночные лошади".
6. Сравните результаты грамматического анализа следующих двух предложений и объясните, чем они различаются семантически.

   Фермер построил изгородь на поле.
   Фермер построил изгородь зимой.
7. Исходя из информации в семантической сети, представленной на рис. 11.3, определите степень родства между Мери и Джоном.

11.3. Способность к рассуждению

А теперь давайте воспользуемся машиной для решения головоломки "Восьмерка", представленной в разделе 11.1, для изучения методов разработки агентов с элементарными способностями к рассуждению.

Порождающие системы

После того как наша машина для разгадывания головоломки "Восьмерка" разберется с расположением фишек на изображении, полученном ее видеокамерой, она должна будет определить, какие фишки следует переместить для решения данной задачи. Первое приходящее на ум решение этой проблемы состоит в предварительном программировании отдельных решений задачи для всех возможных случаев взаимного расположения фишек. В этом случае машине потребуется просто выбрать нужную программу. Однако для этой головоломки существует более 100 тысяч различных возможных конфигураций фишек, поэтому идея предоставления конкретного решения для каждого из этих случаев, безусловно, мало привлекательна. Таким образом, мы вынуждены запрограммировать машину так, чтобы она смогла сама найти решение этой головоломки. Иначе говоря, она должна быть способна самостоятельно принимать решения и делать выводы, т.е выполнять простейшие рассуждения.

Развитие у машин способности к рассуждению уже долгие годы является темой различных исследовательских работ. Одним из результатов подобных

исследований является осознание того факта, что большой класс проблем, связанных с рассуждениями, имеет общие характеристики. Эти общие характеристики были выделены в абстрактный объект, получивший название **порождающая система** и состоящий из трех основных элементов.

1. *Набор состояний.* Каждое **состояние** — это ситуация, которая может возникнуть в прикладной среде. Начальное состояние называется **стартовым** (или исходным). Желаемое состояние (или состояния) именуется **целевым**. (В нашем случае состояние — это конфигурация головоломки, стартовое состояние — это конфигурация головоломки в момент передачи ее машине, а целевое состояние — это конфигурация решенной головоломки, показанная на рис. 11.1.)
2. *Набор порождений (правил или ходов).* **Порождение** — это операция, которую можно выполнить в прикладной среде для перевода системы из одного состояния в другое. Каждое порождение может быть связано с необходимыми условиями, т.е. могут существовать условия, которые должны быть удовлетворены, прежде чем данное порождение может быть выполнено. (В нашем случае порождениями являются перемещения фишек. Каждое такое перемещение имеет необходимым условием существование свободной позиции для перемещаемой фишки.)
3. *Система контроля.* **Система контроля** состоит из логики, способной решить проблему продвижения системы от стартового состояния до целевого. На каждом этапе процесса система контроля должна решить, какое из порождений, удовлетворяющих необходимым условиям, будет применено следующим. (Для примера приведем некоторое состояние нашей головоломки, когда рядом со свободной вакансией находится несколько фишек. Задача системы контроля — решить, какую из них следует перемещать.)

Обратите внимание, что задача, поставленная перед нашей машиной для решения головоломки, вполне может быть сформулирована в контексте порождающей системы. В этих условиях система контроля принимает форму программы. Эта программа проверяет текущее состояние головоломки, идентифицирует последовательность порождений, которая приводит к целевому состоянию, и выполняет эту последовательность. Следовательно, нашей задачей является разработка системы контроля для решения головоломки "Восьмерка".

Важной концепцией в разработке системы управления является **пространство состояний** задачи, представляющее собой совокупность всех возможных состояний, порождений и необходимых условий в порождающей системе. Пространство состояний часто концептуализируется в форме **графа состояний**. Здесь термин *граф* относится к структуре, которую математики назвали

бы **направленным** или **ориентированным графом**, т.е. к набору точек (или **узлов**), соединенных стрелками (или **дугами**). Граф состояний состоит из набора узлов, представляющих состояния в системе, соединенных стрелками, представляющими порождения, переводящие по этой дуге систему из одного состояния в другое. Два узла связываются на графе состояний дугой тогда и только тогда, когда существует порождение, переводящее по этой дуге систему из одного состояния в другое. Необходимые условия по смыслу представлены отсутствием дуги между определенными узлами.

Следует отметить, что подобно тому, как слишком большое количество возможных исходных состояний головоломки не допускает явного предварительного расчета всех возможных вариантов ее решения, эта же проблема размера не позволяет создать и явного представления всего ее графа состояний. Поэтому граф состояний является подходящим средством *осмысления* проблемы, но не позволяет выразить ее во всей полноте. Как бы там ни было, полезно будет рассмотреть (а возможно, и расширить) часть графа состояний нашей головоломки, изображенную на рис. 11.4.

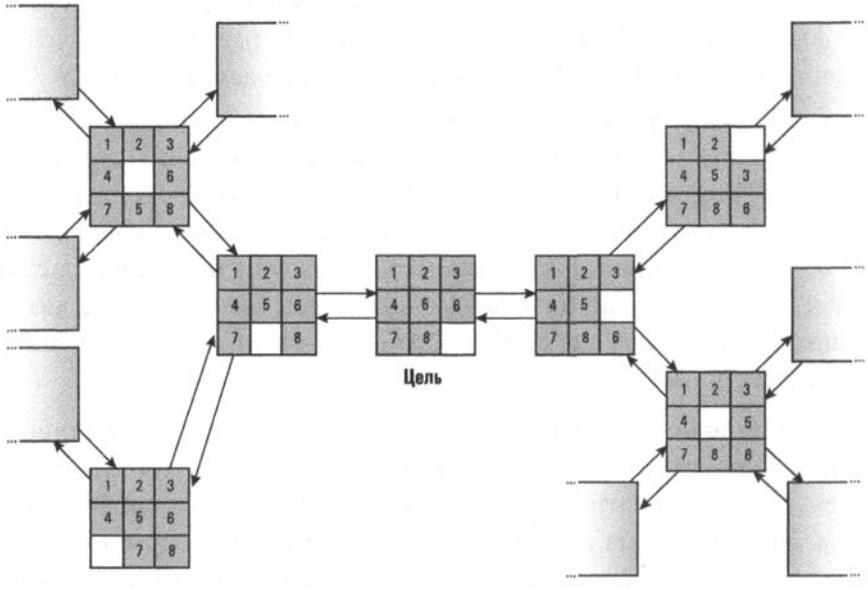


Рис. 11.4. Небольшая часть графа состояний для головоломки "Восьмерка"

Стоящая перед системой контроля проблема, рассмотренная в терминах графа состояний, сводится к нахождению последовательности дуг, по которой система могла бы перейти из начального состояния в целевое. Действительно, подобная последовательность дуг будет представлять ту последовательность порождений, которая позволит решить исходную проблему. Следовательно,

независимо от конкретного приложения, задача системы контроля состоит в нахождении пути по графу состояний. Подобная универсальная точка зрения на систему контроля — важное достижение, полученное нами благодаря выполненному в терминах порождающих систем анализу проблем, требующих для своего решения проведения рассуждений. Таким, образом, если задача может быть охарактеризована в терминах порождающей системы, то ее решение может быть сформулировано в терминах поиска пути в ее графе состояний.

Чтобы подчеркнуть важность этого положения, рассмотрим, как другие задачи могут быть выражены в терминах порождающих систем и, следовательно, как система контроля сможет найти путь через их граф состояний. Одной из групп классических проблем, изучаемых в области искусственного интеллекта, являются игры, например шахматы. Эти игры предполагают некоторые действия умеренной сложности, выполняемые в строго определенном контексте, и поэтому предоставляют собой идеальную среду для тестирования различных теорий. В шахматах состояния соответствующей порождающей системы — это возможные конфигурации положения фигур на доске, порождения — это перемещения фигур, а система контроля воплощена в игроке (человеке или машине). Начальный узел графа состояний представлен доской с фигурами, расставленными на исходных позициях. Разветвлениями этого узла являются дуги, ведущие к тем положениям фигур на доске, которые могут быть получены после выполнения первого хода. Разветвления, выходящие из каждого из этих узлов, — позиция после ответного хода и т.д. При такой формулировке мы можем представить себе игру в шахматы как состязание двух игроков, каждый из которых пытается найти путь через огромнейший граф состояний к конечной вершине, выбранной каждым из них.

Возможно, менее очевидным примером порождающей системы является проблема получения логического заключения из имеющихся фактов. В данной ситуации порождения — это правила логики, называемые **правилами вывода** и позволяющие создавать новые утверждения из уже имеющихся. Например, утверждения "Все студенты старательно учатся" и "Джон является студентом" могут быть скомбинированы, и в результате мы получим утверждение "Джон старательно учится". В такой системе состояния это набор утверждений, которые полагаются верными на данный момент процесса дедукции. Начальным же состоянием системы является набор базовых утверждений (иначе называемых *аксиомами*), на основании которых делаются дальнейшие выводы. Целью является любой набор утверждений, включающий предложенное умозаключение.

Например, на рис. 11.5 показана часть графа состояний, по которому можно проследить вывод утверждения "Сократ смертен" из набора таких утверждений, как "Сократ — мужчина", "Все мужчины — люди" и "Все люди смертны".

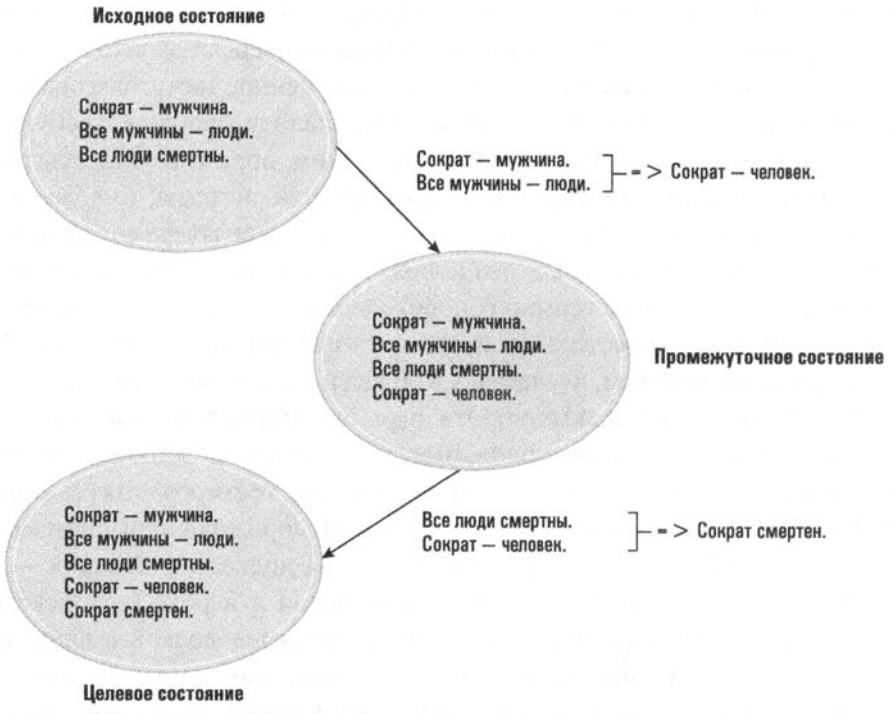


Рис. 11.5. Дедуктивные рассуждения, представленные в контексте порождающей системы

В этом примере мы видим, как совокупность знаний переходит из одного состояния в другое, поскольку в процессе рассуждения применяются соответствующие порождения для генерации дополнительных утверждений. Сегодня такие системы рассуждений, часто реализуемые на языках логического программирования (раздел 6.7), являются основой большинства **экспертных систем**, представляющих собой пакеты прикладного программного обеспечения, предназначенные для имитации причинно-следственных связей, которым обычно следуют эксперты-люди, когда сталкиваются с подобными ситуациями. Например, медицинские экспертные системы используются для диагностики заболеваний или разработки методов лечения.

Дерево поиска

Вы уже знаете, что в контексте порождающих систем задача системы контроля включает анализ графа состояний в целях нахождения пути от стартового узла к целевому. Простейший метод этого поиска состоит в проходе каждой дуги, связанной с начальным узлом и записью достигнутого в каждом случае состояния, с последующим проходом всех дуг новых состояний, также

сопровождающихся записью результата, и т.д. Таким образом, поиск пути к цели "расходится" от начального состояния во всех направлениях, как круги на воде от падающей капли. Этот процесс продолжается, пока одно из полученных состояний не окажется искомой целью. В результате решение считается найденным и системе контроля остается просто реализовать порождения, следуя по обнаруженному пути от начального состояния к целевому.

Результатом применения этой стратегии является построение дерева, называемого **деревом поиска**, которое состоит из части графа состояний, исследованной системой контроля. Корневой узел дерева поиска — это начальное состояние, а дочерние состояния каждого узла — это состояния, которые могут быть достигнуты из этого родительского узла за одно порождение. Каждая дуга между узлами дерева поиска представляет реализацию отдельного порождения, а каждая последовательность переходов от корня до листового узла — путь между соответствующими состояниями на графе состояний.

На рис. 11.7 представлено дерево поиска, которое может быть построено при решении нашей головоломки с начальной конфигурацией, показанной на рис. 11.6. Левая ветвь этого дерева предлагает решить проблему путем перемещения на первом шаге фишки 6 вверх, центральная ветвь представляет вариант с перемещением на первом шаге фишки 2 вправо, а правая отражает ситуацию после перемещения на первом шаге фишки 5 вниз. Более того, дерево поиска показывает, что если мы начнем с перемещения фишки 6 вверх, то единственной доступный следующий ход — это перемещение фишки 8 вправо. (В действительности мы можем вновь переместить фишку 6 вниз, но это вернет систему в первоначальное положение, поэтому мы будем считать такой ход ненужным и в дальнейшем подобные ходы рассматривать не будем.)

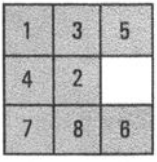


Рис. 11.6. Головоломка "Восьмерка" в выбранном исходном состоянии

Целевое состояние достигается на листовом уровне дерева поиска, изображенного на рис. 11.7. Поскольку это означает завершение процедуры поиска, система контроля завершает поиск пути и приступает к созданию набора пошаговых инструкций, которые будут использованы для решения головоломки во внешней среде. Данный процесс состоит в простом прохождении вверх по дереву поиска, начиная с найденного целевого узла. При этом сведения обо всех порождениях, соединяющих узлы в ветвях дерева, записываются в стек в

том же порядке, в каком они встречаются. Применение этого метода к дереву поиска, показанному на рис. 11.7, приведет к последовательности порождений, показанной на рис. 11.8. Отметим, что система контроля сможет теперь решить головоломку во внешнем мире, просто следуя инструкциям, извлекаемым из стека.

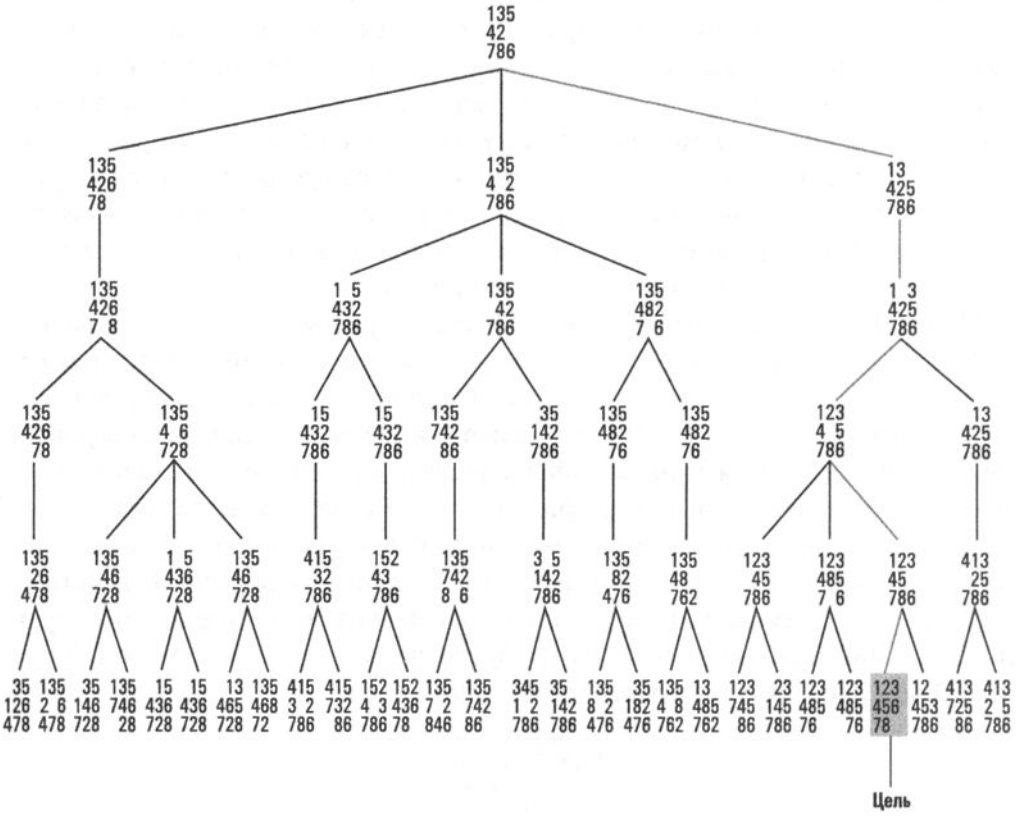


Рис. 11.7. Пример дерева поиска



Рис. 11.8. Последовательность порождений, записанная в стек для последующего выполнения

Есть еще одно наблюдение, которое мы должны здесь отметить. Вспомним, что для построения деревьев, которые мы обсуждали в главе 8, используется система указателей, показывающая путь вниз по дереву, — это позволяет двигаться от родительского узла к дочерним. Однако в данном случае системе контроля требуется передвигаться в обратном направлении, от потомков к их родителям, так как она движется вверх по дереву от конечного состояния системы к начальному. Такие деревья создаются с собственной системой указателей, показывающей чаще вверх, чем вниз (или в некоторых случаях с двумя наборами указателей, что при необходимости позволяет двигаться в обоих направлениях).

Эвристические методы

В нашем примере на рис. 11.7 была выбрана начальная конфигурация, которая породила вполне обозримое дерево поиска. Однако деревья поиска, порожденные при попытке решить более сложную проблему, могут расти намного быстрее. Например, при игре в шахматы возможно 20 вариантов только первого хода, т.е. корневой узел дерева поиска будет иметь 20 ветвей в отличие от 3 ветвей в рассматренном выше случае с головоломкой "Восьмерка". Более того, средняя партия в шахматы состоит из 30–35 пар ходов в отличие от предыдущего примера. Однако даже в случае с головоломкой "Восьмерка" дерево поиска может стать достаточно большим, если решение не будет найдено достаточно быстро. В результате расчет всего дерева вариантов может оказаться таким же непрактичным решением, как и представление всего графа состояний.

Наша стратегия противодействия этой проблеме будет заключаться в изменении порядка, в котором создается дерево поиска. Вместо того чтобы строить дерево посредством **горизонтального поиска** (или поиска в ширину — *breadth-first search*), что означает построение дерева за счет последовательного добавления уровней, мы можем рассмотреть более обещающий путь в глубину и принять во внимание остальные альтернативы только в том случае, если выбранный путь приводит к провалу. Образующаяся вертикальная конструкция дерева поиска (**поиск в глубину** — *depth-first search*) означает, что дерево строится прежде всего по вертикальным связям и лишь затем по горизонтальным. Если говорить точнее, этот подход часто называют поиском **по первому наилучшему совпадению**, признавая тот факт, что вертикальный путь, выбранный для продолжения поиска, является тем, который, как предполагается, предлагает наилучший потенциал.

Вариант поиска по первому наилучшему совпадению больше похож на ту стратегию, которой мы, люди, обычно руководствуемся при решении головоломки "Восьмерка". Человек редко рассматривает сразу несколько альтернатив,



Основные положения для запоминания

- **Для некоторых задач оптимизации, таких как "найти лучшее" или "найти самое маленькое", невозможно найти точное окончательное решение в разумные сроки, но хорошее приближение к оптимальному решению может быть найдено достаточно быстро.**

как в случае поиска по горизонтали. Сталкиваясь с необходимостью выбора, человек, вероятнее всего, предпочтет тот вариант, который покажется ему наиболее обещающим, и последует ему. Подчеркнем, что выше было сказано именно "*покажется* наиболее обещающим". У нас обычно нет уверенности в том, какой из вариантов действительно лучший, и мы просто следуем своей интуиции, которая, безусловно, может и подвести. Несмотря на это использование такой интуитивной информации, похоже, дает человеку преимущество перед методом решения "в лоб", в котором каждому варианту уделяется одинаковое внимание, поэтому может показаться целесообразным применять интуитивные методы и в автоматизированных системах управления.

Но для этого нам нужен способ определения, какое из нескольких состояний представляется наиболее перспективным. Выбранный подход состоит в использовании **эвристики**, которая в нашем случае представляет собой количественную величину, связанную с каждым состоянием и полученную при попытке оценить "расстояние" от этого состояния до ближайшей цели. В некотором смысле подобная эвристика является мерой *прогнозируемой стоимости*. При выборе между двумя состояниями то, у которого значение эвристики будет меньше, можно считать состоянием, из которого, по-видимому, достичь цели удастся с меньшими затратами. Следовательно, это состояние и будет представлять то направление, в котором мы должны продолжать поиск.

Выбираемая эвристика должна отвечать двум требованиям. Во-первых, она должна представлять собой разумную оценку оставшегося объема работы, связанной с поиском решения задачи, когда соответствующее состояние было достигнуто. Очевидно, что только при этих условиях эвристка сможет предоставить нам значимую информацию для выбора из возможных вариантов: чем лучше будет предоставленная эвристикой оценка, тем лучше будут решения, основанные на этой информации. И во-вторых, эвристика должна вычисляться достаточно легко. Иначе говоря, использование эвристики должно приносить пользу процессу поиска, а не превращаться в его бремя. Если вычисление эвристики требует чрезвычайно сложных расчетов, то необходимое для выполнения этих расчетов время можно было бы потратить просто на выполнение поиска в ширину.

В случае головоломки "Восьмерка" простой эвристический метод может предусматривать использование в качестве характеристики "расстояния" от каждого состояния до конечной цели того количества фишек, которое в этом состоянии находится не на своей конечной позиции. Идея состоит в том, что состояние, в котором четыре фишки находятся не на своем месте, будет дальше от конечной цели (и следовательно, менее привлекательно), чем состояние, в котором только две фишки находятся не на своем месте. Однако такая эвристика не учитывает, насколько далеко от своих конечных позиций находятся данные фишки. Если эти две фишки находятся далеко от своих позиций в собранной головоломке, то, чтобы переместить их туда через все игровое поле, может потребоваться много порождений.

Поведенческий интеллект

В ранних работах в области искусственного интеллекта к предмету подходили в контексте явного написания программ, имитирующих интеллект. Однако сегодня многие утверждают, что человеческий интеллект основан не на выполнении сложных программ, а на простых функциях "стимул–отклик", которые развивались на протяжении многих и многих поколений. Эта теория "интеллекта" известна как *поведенческий интеллект*, потому что "интеллектуальные" функции реагирования на стимулы, по-видимому, являются результатом поведения, которое позволило одним людям выживать и размножаться, в то время как другие этого не сделали.

Теория поведенческого интеллекта, похоже, дает ответы на некоторые вопросы в сообществе искусственного интеллекта, например почему машины, основанные на архитектуре фон Неймана, легко превосходят людей в вычислительных навыках, но любые попытки проявления здравого смысла оказываются для них задачей запредельной сложности. В результате положения теории поведенческого интеллекта обещают стать одним из важнейших факторов в исследованиях, проводимых в области искусственного интеллекта. Как указывается в этой главе, методы, основанные на теории поведенческого интеллекта, уже были применены в области искусственных нейронных сетей с целью научить нейроны вести себя желаемым образом. В области генетических алгоритмов они использовались как альтернатива более традиционному подходу к процессу программирования, а в робототехнике — для улучшения эффективности машин за счет использования реактивных стратегий.

Тогда несколько лучший вариант эвристики будет состоять в том, чтобы измерить расстояние, на котором каждая фишка в данном состоянии находится от своего места назначения, а затем сложить полученные значения, чтобы получить одну общую количественную оценку — назовем ее *оценочной стоимостью*. Фишка, непосредственно примыкающая к своей конечной позиции,

будет характеризоваться расстоянием, равным единице, тогда как фишка, один из углов которой касается любой из вершин квадрата, определяющего ее конечное положение, будет характеризоваться расстоянием, равным двум (чтобы она заняла свое положение, ее необходимо переместить по крайней мере на одну позицию по вертикали и на одну позицию по горизонтали). Вычислить такую эвристику несложно и в конечном счете она дает приблизительную оценку количества ходов, необходимых для перевода головоломки из ее текущего состояния в целевое. Например, оценочная стоимость, связанная с конфигурацией, представленной на рис. 11.9, будет равна семи (поскольку фишки 2, 5 и 8 находятся на расстоянии одного хода от их конечного местоположения, а фишки 3 и 6 — на расстоянии двух ходов). И действительно, чтобы привести эту конфигурацию головоломки в конечное состояние, потребуется не менее семи ходов.

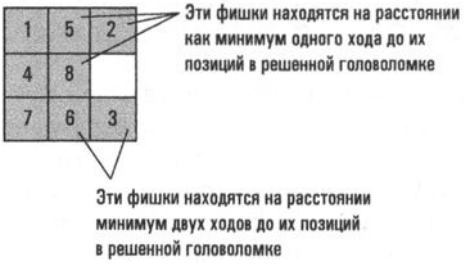


Рис. 11.9. Еще один вариант исходного состояния головоломки

Теперь, когда найден эвристический метод решения головоломки, следующим этапом будет включение его в процесс принятия решений. Напомним, что человек, сталкиваясь с необходимостью принятия решения, чаще всего предпочитает вариант, быстрее всего приводящий к достижению цели. Поэтому наша процедура поиска должна оценивать предполагаемую стоимость всех вариантов при каждом разветвлении процесса и в каждом случае выбирать альтернативу с наименьшей стоимостью. Применение подобной стратегии демонстрируется на рис. 11.10, — на нем представлен алгоритм расчета дерева поиска и выполнения найденного решения.

Применим этот алгоритм для нашей головоломки, выбрав в качестве стартовой конфигурацию, показанную на рис. 11.6. Прежде всего определим это состояние как корень дерева поиска и запишем его оценочную стоимость, которая равна пяти. Затем выполним первый проход по телу цикла `Пока`, в результате чего будут порождены три новых узла, показанных на рис. 11.11. Обратите внимание, что для каждого из этих узлов в круглых скобках показана вычисленная оценочная стоимость.

```
Принимаем начальный узел графа состояний в качестве корня
  дерева поиска и записываем его оценочную стоимость.
Пока (целевой узел не достигнут):
  Выбрать крайний слева листовой узел с наименьшей
    оценочной стоимостью.
  Присоединить к этому выбранному узлу в качестве дочерних все
    узлы, достигаемые от него за одно порождение.
  Записать вычисленную оценочную стоимость для каждого из
    этих новых узлов, добавленных в дерево поиска.
Пройти по дереву поиска вверх от целевого узла к корню дерева,
  помещая в стек сведения обо всех порождениях, представленных
    последовательно проходимыми дугами графа.
Решить исходную задачу, выполнив всю последовательность
    порождений, сведения о которых записаны в стеке.
```

Рис. 11.10. Алгоритм работы системы контроля, использующий эвристический метод

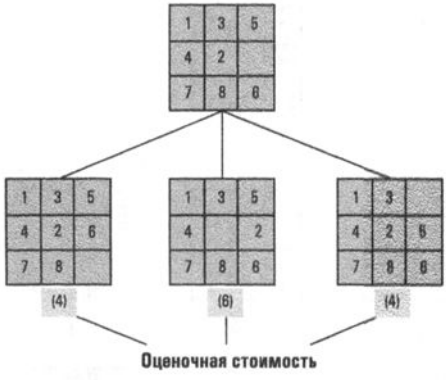


Рис. 11.11. Начало эвристического поиска

Конечный узел еще не достигнут, поэтому цикл Пока выполняется еще раз — для узла, расположенного слева ("Выбрать крайний слева листовой узел с наименьшей оценочной стоимостью"). Полученное в результате дерево поиска представлено на рис. 11.12.

Обратите внимание, что теперь оценочная стоимость для левого узла будет равна пяти, а это указывает, что данный узел уже не является лучшим вариантом продолжения поиска. Алгоритм учитывает это и при следующем проходе цикла Пока требует раскрыть правый узел (именно он теперь является "крайним слева листовым узлом с наименьшей оценочной стоимостью"). Расширенное подобным образом дерево поиска показано на рис. 11.13.

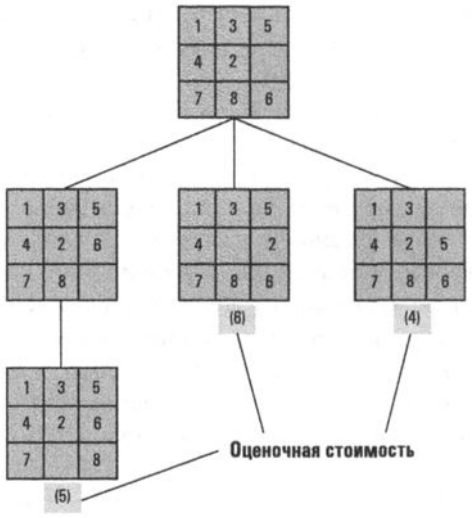


Рис. 11.12. Дерево поиска после двух проходов цикла

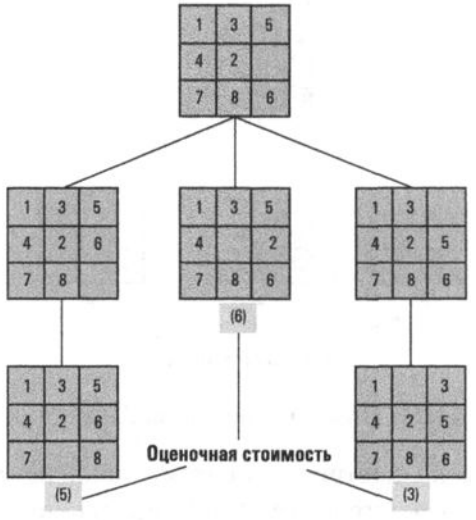


Рис. 11.13. Дерево поиска после трех проходов цикла

Похоже, что наш алгоритм на правильном пути. Поскольку оценочная стоимость вновь раскрытого узла равна всего лишь трем, цикл `Пока` предписывает продолжать путь с этого узла и в конце концов достигает цели с построением дерева поиска, показанного на рис. 11.14. Сравнив это дерево с деревом, представленным на рис. 11.7, можно увидеть, что, даже с учетом неудачно выбранного вначале направления, использование эвристической информации значительно уменьшило размер дерева поиска и позволило реализовать намного более эффективный процесс.

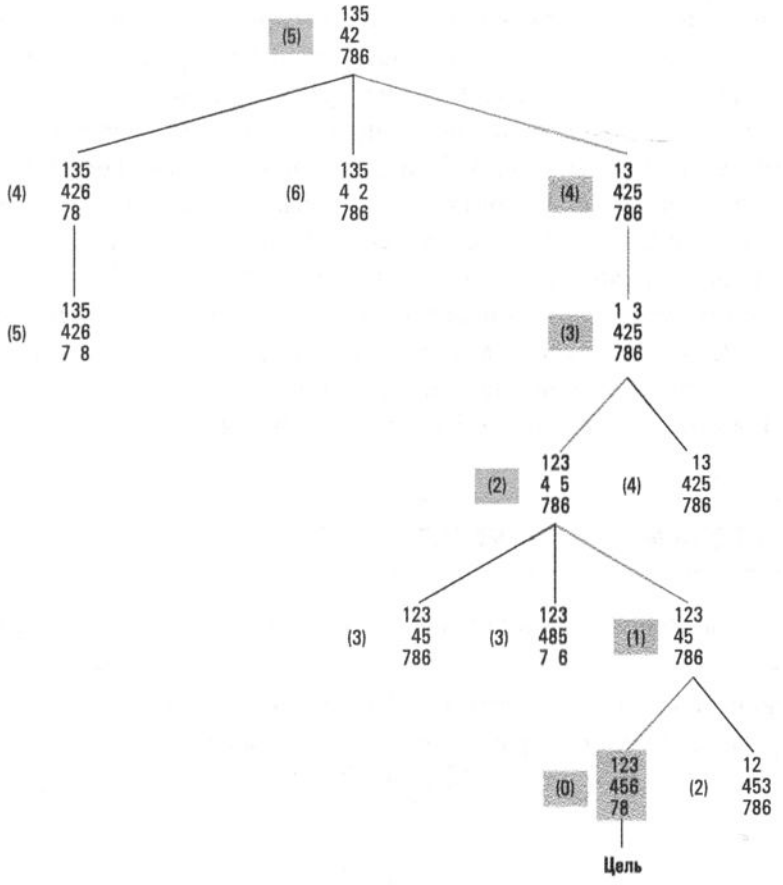


Рис. 11.14. Полное дерево поиска, созданное нашей эвристической системой

По достижении конечного состояния выполнение цикла `Пока` прекращается, и алгоритм переходит к прохождению дерева от пункта назначения до начального состояния, помещая в стек сведения обо всех встреченных порождениях. Полученный стек выглядит точно так же, как и на рис. 11.8.

И наконец алгоритм требует выполнить все порождения строго в порядке их расположения в стеке. В этот момент мы увидим, как машина опустит манипулятор и начнет перемещать фишки.

Приведем еще один последний комментарий в отношении эвристического поиска. Алгоритм, который был предложен в этом разделе и который часто называют алгоритмом поиска по первому наилучшему совпадению, не обязательно найдет наилучшее решение во всех приложениях. Например, при поиске пути к другому городу с использованием имеющейся в автомобиле системы глобального позиционирования (GPS) вы обычно желаете найти *кратчайший*

путь, а не просто какой-либо возможный. **Алгоритм A*** (произносится как "алгоритм А звезда") является модифицированной версией обсуждавшегося выше алгоритма поиска по первому наилучшему совпадению и позволяет найти именно *оптимальное* решение. Основное различие между этими двумя алгоритмами состоит в том, что в дополнение к эвристической оценке, т.е. к нашей оценочной стоимости, алгоритм A* учитывает "накопленную стоимость", оценивающую стоимость достижения каждого листового узла из начального узла при выборе очередного узла для расширения. (В случае GPS-системы автомобиля эта стоимость представляет собой пройденное расстояние, которое GPS-система получает из своей внутренней базы данных.) В результате алгоритм A* строит свои решения на оценках стоимости полных потенциальных путей, а не просто на эвристической оценочной стоимости.

11.3. Вопросы и упражнения

1. Каково значение порождающих систем для теории искусственного интеллекта?
2. Нарисуйте часть графа состояний, окружающую представленный на рисунке узел полного графа состояний головоломки "Восьмерка".

4	1	3
	2	6
7	5	8

3. Используя горизонтальный подход, нарисуйте дерево поиска, которое могла бы создать система контроля машины при решении головоломки "Восьмерка" с приведенным ниже начальным состоянием.

1	2	3
4	8	5
7	6	

4. С помощью карандаша и бумаги, применив горизонтальный подход, попытайтесь построить дерево поиска решения головоломки с приведенным ниже начальным состоянием (успеха вы не добьетесь). С какой проблемой вы столкнулись?

4	3	
2	1	8
7	6	5

5. Какая аналогия может быть прослежена между нашей эвристической системой для решения головоломки "Восьмерка" и пытающимся достигнуть вершины альпинистом, осматривающим только доступную его взору часть ландшафта и всегда направляющимся в сторону наиболее крутого подъема.
6. Используя описанный в данном разделе эвристический подход, примените алгоритм поиска по первому наилучшему совпадению (см. рис. 11.10) к проблеме поиска решения следующей исходной позиции в головоломке "Восьмерка".

1	2	3
4		8
7	6	5

7. Уточните метод вычисления предполагаемой стоимости так, чтобы алгоритм поиска решения, представленный на рис. 11.10, не допускал неправильных выборов, как это имело место в примере, приведенном в тексте раздела. Можете ли вы найти пример, в котором предложенная вами система все-таки сбивалась бы с правильного пути?
8. Нарисуйте дерево поиска, построенное в соответствии с алгоритмом поиска по первому наилучшему совпадению (см. рис. 11.10) при поиске маршрута из города Лисберг в город Бедфорд. Каждый узел в дереве поиска будет представлять город на карте. Начните с узла для города Лисберг. При расширении узла добавляйте только те города, которые непосредственно связаны с расширяемым городом. Запишите в каждом узле расстояние по прямой линии до Бедфорда и используйте его в качестве оценочной стоимости. Какое решение будет найдено при использовании алгоритма поиска по первому наилучшему совпадению? Является ли найденное решение кратчайшим маршрутом?



Расстояние по прямой линии до Бедфорда от городов:

Дейтон	16
Лисберг	34
Стоун	19

9. Алгоритм A* представляет собой модификацию алгоритма поиска по первому наилучшему совпадению, в которой в исходный алгоритм внесены два существенных изменения. Во-первых, в нем

записывается *фактическая стоимость* достижения состояния. В случае маршрута на карте фактической стоимостью является пройденное расстояние. Во-вторых, при выборе узла для расширения выбирается тот узел, сумма фактической стоимости и оценочной стоимости которого является наименьшей. Нарисуйте дерево поиска для условий задачи из упражнения 8, которое будет получено в результате внедрения двух этих изменений. Для каждого узла запишите расстояние, пройденное от исходного узла до данного города, оценочную стоимость достижения цели, а также их сумму. Какое решение будет найдено? Является ли найденное решение кратчайшим маршрутом?

11.4. Дополнительные области исследований

В этом разделе рассмотрены вопросы манипулирования знаниями, обучения и решения других очень сложных проблем, которые продолжают бросать вызов исследователям в области искусственного интеллекта. Эти действия включают в себя умения, которые кажутся совсем простыми для человеческого разума, но, очевидно, оказываются весьма обременительными в отношении возможностей машин. На данный момент значительная часть прогресса в разработке "интеллектуальных" агентов была достигнута, по существу, лишь благодаря предотвращению прямой конфронтации с подобными проблемами — возможно, путем применения остроумных обходных путей или за счет ограничения той области, в которой проблема возникает.

Представление и манипулирование знаниями

При обсуждении восприятия было показано, что понимание изображений требует значительного объема знаний об объектах, присутствующих на изображении, а также что значение того или иного предложения может зависеть от его контекста. Это примеры той роли, которую играют хранилища знаний, часто называемых **знаниями реального мира**, сохраняемых человеческим разумом. По какой-то причине люди хранят в своей памяти огромное количество информации и при необходимости извлекают ее с удивительной эффективностью. Предоставление подобных возможностей машинам является серьезной проблемой в области искусственного интеллекта.

В то же время современные компьютерные системы преуспевают в хранении и извлечении огромных объемов информации, представляющей собой любые данные, которые могут быть дискретизированы, оцифрованы и сохранены

в виде числовых значений, представленных в двоичном коде из единиц и нулей. Точная взаимосвязь между данными и знаниями сложна и неуловима, но по мере роста объемов колоссального количества данных, доступных для компьютеров, расширяются и возможности извлечения из них новой информации и знаний — с использованием так называемых систем "больших данных". Искусственный интеллект, машинное обучение и статистические методы способствуют разработке программных инструментов, которые позволяют обнаруживать в больших наборах данных ранее неизвестные шаблоны или взаимосвязи.



Основные положения для запоминания

- **Большие наборы данных предоставляют новые возможности и ставят сложные задачи в отношении извлечения информации и накопления знаний.**
- **Большие наборы данных предоставляют новые возможности для выявления тенденций и обнаружения взаимосвязей в данных, а также для решения существующих проблем.**
- **Вычислительные инструменты облегчают обнаружение взаимосвязей в информации, хранящейся в больших наборах данных.**

Основная задача в этой области — найти способы представления и хранения знаний. Ее решение осложняется тем фактом, что, как мы уже видели, знания появляются как в декларативной, так и в процедурной формах. Отсюда следует, что представление знаний — это не просто представление фактов или данных, а охват гораздо более широкого спектра задач. Поэтому вполне правомерен вопрос, будет ли вообще в конечном счете найдена единая схема для представления всех форм знаний.

Однако проблема заключается не только в том, как представлять и хранить знания. Знания также должны быть легко доступны, и достижение этой доступности, в свою очередь, также является сложной задачей. В качестве средства представления и хранения знаний сейчас часто используются семантические сети, представленные в разделе 11.2, однако извлечение информации из них может быть проблематичным. Например, смысл утверждения "Мери ударила Джона" в значительной степени зависит от относительного возраста Мери и Джона. (Их возраст 2 года и 30 лет или наоборот?) Подобная информация будет сохранена в полной семантической сети, представленной на рис. 11.3, но для извлечения такой информации в процессе контекстного анализа может потребоваться значительное количество операций поиска в этой сети. При работе с чрезвычайно большими наборами данных использование эффективных алгоритмов поиска или эвристических методов является важным элементом конечных решений независимо от того, как именно организована информация.



Основные положения для запоминания

- **Для эффективного поиска информации очень важно наличие соответствующих инструментов поиска.**

Еще одной проблемой, связанной с доступом к знаниям, является выявление знаний, которые связаны с поставленной задачей неявно, а не совершенно очевидным образом. Вместо того чтобы на вопрос "Артур выиграл гонку?" получать в ответ прямое и грубоватое "Нет", нам нужна система, которая могла бы ответить "Нет, он простудился и не смог принять участие в соревнованиях". Однако задача состоит не в том, чтобы просто извлечь всю связанную информацию. В действительности нам нужны системы, которые будут способны отличать просто связанную информацию от *релевантной* информации. Например, такую фразу, как "Нет, Артур родился в январе и его сестру зовут Лиза", никак нельзя считаться достойным ответом на предыдущий вопрос, даже если представленная информация абсолютно верна и очевидным образом связана с тем, о ком спрашивали. Следовательно, эффективный доступ к знаниям — это проблема не только поиска, но и фильтрации информации или распознавания ее структуры и имеющихся закономерностей.



Основные положения для запоминания

- **Системы фильтрации информации являются важными инструментами при поиске информации и распознавании присутствующих в ней закономерностей.**

Другой подход к разработке более совершенных систем извлечения знаний состоял в том, чтобы включить в процесс извлечения различные формы рассуждений, что привело к появлению того, что сейчас называют **метарассуждением**, т.е. рассуждением о рассуждениях. Пример, первоначально использовавшийся в контексте поиска в базе данных, заключается в применении **предположения о замкнутости мира**, т.е. предположения о том, что утверждение ложно, если не может быть явно получено из доступной информации. Например, предположение о замкнутости мира позволяет базе данных сделать вывод, что Николь Смит не подписалась на определенный журнал, хотя база данных вообще не содержит никакой информации о Николь. Процесс состоит в том, чтобы заметить, что Николь Смит отсутствует в списке подписчиков журнала, а затем применить предположение о замкнутости мира, чтобы сделать вывод, что Николь Смит не подписалась на него.

На первый взгляд, предположение о замкнутости мира кажется тривиальным, но оно имеет следствия, демонстрирующие тот факт, что совершенно безобидные методы метарассуждений могут иметь едва заметные нежелательные последствия. Предположим, например, что единственное знание, которое у нас есть, это одно следующее утверждение:

Микки — мышь ИЛИ Дональд — утка.

Только из этого утверждения мы не можем сделать вывод, что Микки на самом деле является мышью. Таким образом, предположение о замкнутости мира заставляет нас заключить, что следующее утверждение будет ложным:

Микки — это мышь.

Аналогичным образом предположение о замкнутости мира заставляет нас сделать вывод, что следующее утверждение также ложно:

Дональд — это утка.

Таким образом, предположение о замкнутости мира привело нас к противоречивому выводу: несмотря на то что хотя бы одно из этих утверждений должно быть истинным, оба они являются ложными. Понимание всех последствий таких внешне безобидных методов метарассуждений является целью исследований, проводимых в области искусственного интеллекта и баз данных. Оно также еще раз подчеркивает сложности, связанные с разработкой интеллектуальных систем.

Наконец, существует проблема, известная как **проблема фреймов**, касающаяся поддержания сохраненных знаний в актуальном состоянии в изменяющейся среде. Если интеллектуальный агент намеревается использовать имеющиеся знания для определения своего поведения, то эти знания всегда должны быть *актуальными*. Но количество знаний, необходимых для поддержания интеллектуального поведения, может оказаться огромным, и обеспечение актуальности всех этих знаний в изменяющейся среде может оказаться задачей невероятных размеров. Осложняющим фактором здесь является также то, что изменения в окружающей среде часто косвенно изменяют другие элементы информации, и учет таких косвенных последствий может быть сложной задачей. Например, если ваза с цветами была опрокинута и разбилась, ваши знания о ситуации больше не должны содержать тот факт, что в вазе находится вода, хотя то, что при падении вазы вода разлилась, только косвенным образом связано с тем, что ваза разбилась. Таким образом, для решения проблемы фреймов требуется не только способность эффективно хранить и извлекать огромные объемы информации, но также совершенно необходимо, чтобы система хранения должным образом реагировала на возможные косвенные последствия.

Машинное обучение

В дополнение к средствам представления и манипулирования знаниями, хотелось бы предоставить интеллектуальным агентам возможность самим получать новые знания. Всегда можно "обучить" реализованного на базе компьютера агента, написав и установив новую программу или явно добавив новые данные к тем, которые уже были сохранены в его памяти, однако хотелось бы, чтобы интеллектуальные агенты также могли учиться самостоятельно. Желательно, чтобы агенты были способны адаптироваться к изменяющимся условиям и успешно справлялись даже с такими задачами, для решения которых было бы очень сложно написать программы заранее. Робот, предназначенный для работы по дому, в перспективе обязательно столкнется с новой мебелью, новой бытовой техникой, новыми домашними животными и даже новыми владельцами. Автономный самоуправляемый автомобиль должен уметь приспосабливаться к изменениям в ограничительных линиях на дорогах. Игровые агенты должны уметь разрабатывать и применять новые собственные стратегии.

Одним из способов классификации подходов к **машинному обучению** является требуемый уровень вмешательства человека. На первом уровне находится метод обучения посредством **подражания**, когда человек прямо демонстрирует все этапы решения поставленной задачи (возможно, выполнением последовательности требуемых компьютерных операций или же физически перемещая робота для демонстрации последовательности выполняемых движений), а компьютер просто записывает эти этапы. Данная форма обучения многие годы использовалась в прикладных программах, таких как электронные таблицы или текстовые процессоры, когда часто встречающаяся последовательность выполняемых действий записывалась в виде макроса, а затем многократно воспроизводилась выдачей единственной команды. Обратите внимание, что обучение подражанием возлагает на агента лишь незначительную ответственность.

Следующий уровень представляет **обучение с учителем**. При обучении с учителем человек определяет правильный ответ для некоторого ряда примеров, а затем агент обобщает эти примеры для разработки алгоритма, который затем применяет к новым случаям. Серия примеров называется **обучающей выборкой** (или **обучающим набором**). Типичные области применения обучения с учителем включают в себя обучение распознаванию почерка или голоса человека, обучение различению нежелательной и желательной электронной почты и обучение распознаванию заболевания по ряду симптомов.

Третий уровень — это **обучение с подкреплением**. При обучении с подкреплением агенту дается общее правило, позволяющее ему самостоятельно принимать решения, когда он достиг успеха или не справился с заданием в процессе проб и ошибок. Обучение с подкреплением подходит для обучения

игре в шахматы или шашки, так как успех или неудача легко определяется. В отличие от обучения с учителем обучение с подкреплением позволяет агенту действовать автономно, поскольку он учится улучшать свое поведение с течением времени.

Семейство методов машинного обучения, называемых **обучением без учителя**, находится на том же уровне, что и обучение с подкреплением, но используется для других целей. Алгоритмы кластеризации, анализ главных компонентов и неконтролируемые нейронные сети (описанные в следующем разделе) — вот несколько примеров подходов, которые можно использовать для выявления закономерностей или определения значения данных с минимальным вмешательством человека.

Обучение все еще остается сложной областью исследований, поскольку пока не было найдено общего универсального принципа, который охватывал бы все возможные методы обучения. Тем не менее есть множество примеров значительного прогресса. Одним из них является система ALVINN (*Autonomous Land Vehicle In a Neural Net* — автономное наземное транспортное средство в нейронной сети), разработанная в Университете Карнеги–Меллона для обучения системы управления автофургоном с помощью бортового компьютера с использованием видеокамеры для ввода данных. В этом проекте был выбран метод обучения с учителем. Система ALVINN собирала данные, поступающие в процессе управления фургоном человеком-водителем, а затем использовала эти данные для корректировки собственных решений по управлению машиной. После завершения обучения система была способна предсказывать, куда направлять машину, проверять свой прогноз по данным, полученным от водителя, а затем изменять значения параметров таким образом, чтобы приблизиться к выбору, сделанному человеком. Система ALVINN была обучена настолько хорошо, что могла успешно управлять фургоном, движущимся со скоростью семьдесят миль в час. Достигнутый успех способствовал проведению дополнительных исследований, в результате которых были созданы управляющие системы, успешно управлявшие движением автомобиля на скоростных трассах в потоке реального трафика.

В завершение следует рассмотреть еще один феномен, тесно связанный с обучением: *открытие*. Различие между ними состоит в лишь том, что обучение основано на "достижении цели", а открытие — нет. Термин "открытие" используется в отношении чего-то неожиданного, что вовсе не предполагается при обучении. Можно начать изучать иностранный язык или записаться на курсы вождения автомобиля, а затем обнаружить, что эти задачи оказались сложнее, чем ожидалось. И наоборот, исследователь может неожиданно обнаружить большое озеро, хотя изначально он просто хотел посмотреть, что находится там, впереди. Для разработки агентов, способных эффективно совершать

открытия, необходимо, чтобы такой агент мог выявлять потенциально плодотворный "ход мыслей". В этом случае возможность совершения открытия в значительной степени зависит от способности рассуждать и использовать эвристические методы. Более того, многие потенциальные применения открытий требуют, чтобы агент был также способен отличать значимые результаты от незначительных. Например, агент интеллектуального анализа данных не должен сообщать обо всех тривиальных отношениях и взаимосвязях, которые он в них обнаруживает.

Знания в логическом программировании

Важной проблемой при представлении и хранении знаний является то, что это должно быть сделано способом, совместимым с системой, которая должна будет иметь доступ к этим знаниям. Именно в таком контексте часто оказывается полезным логическое программирование (см. раздел 6.7). В подобных системах знания представляются "логическими" утверждениями, подобными следующим:

Дамбо — слон.

и

Если X *— слон, значит* X *— серый.*

Такие утверждения могут быть представлены с использованием систем обозначений, которые удобны для применения правил вывода. В свою очередь, последовательности дедуктивных рассуждений, подобные представленным на рис. 11.5, в этом случае могут быть реализованы простым способом. Следовательно, в логическом программировании представление и хранение знаний хорошо интегрированы с процессом извлечения и применения знаний. Можно сказать, что системы логического программирования обеспечивают "бесшовную" границу между хранимыми знаниями и их использованием.

Примеры успеха в создании компьютерных систем открытий включают, в частности, систему Bacon, получившую свое название в честь философа сэра Фрэнсиса Бэкона, открывшую (или, скорее, "заново открывшую") закон электрического сопротивления Ома, третий закон движения планет Кеплера и закон сохранения импульса. Возможно, еще более убедительным примером является система AUTOCLASS, которая, используя спектральные данные инфракрасного диапазона, обнаружила новые классы звезд, которые ранее были неизвестны в астрономии, — настоящее научное открытие, совершенное с помощью компьютера.

Генетические алгоритмы

Алгоритм A* (представленный в предыдущем разделе) позволяет найти оптимальное решение многих поисковых задач; однако существуют некоторые проблемы, которые слишком сложны для решения с помощью подобных методов (их выполнение требует объемов памяти, превышающих доступные, или их завершение не может быть достигнуто в течение разумного периода времени). Решение таких проблем иногда может быть найдено посредством использования эволюционного процесса, включающего многие поколения пробных решений. Подобная стратегия является основой того, что принято называть **генетическими алгоритмами**. По сути, генетические алгоритмы позволяют найти решение за счет реализации случайного поведения в сочетании с моделированием репродуктивной теории и эволюционного процесса естественного отбора.

Генетический алгоритм начинается с генерации случайного пула пробных решений. Каждое решение — это только предположение. (В случае головоломки "Восьмерка" пробное решение может быть случайной последовательностью перемещений фишек.) Каждое пробное решение называется **хромосомой**, а каждый компонент хромосомы называется **геном** (в случае головоломки "Восьмерка" ген — это перемещение одной фишки).

Поскольку каждая исходная хромосома является случайным предположением, маловероятно, что она будет представлять конечное решение рассматриваемой проблемы. В результате генетический алгоритм продолжает работу, генерируя новый пул хромосом, в котором каждая хромосома является *потомком* (дочерней хромосомой) двух хромосом (*родителей*) из предыдущего пула. Родители выбираются из пула случайным образом, но при этом вероятностное предпочтение отдается тем хромосомам, которые, по-видимому, имеют наилучшие шансы привести алгоритм к решению задачи, тем самым подражая эволюционному принципу выживания наиболее приспособленных. (Определение того, какие хромосомы являются лучшими кандидатами на роль родителя, является, пожалуй, самым проблематичным этапом во всем процессе работы алгоритма.) Каждый потомок получает случайную комбинацию генов от родителей. Кроме того, генерируемое потомство иногда может видоизменяться тем или иным случайным образом. Предполагается, что за счет повторения этого процесса снова и снова пробные решения будут постепенно улучшаться, пока не будет достигнут очень хороший, если не лучший результат. К сожалению, нет полной уверенности в том, что генетический алгоритм в конечном итоге найдет решение, однако проведенные исследования показали, что генетические алгоритмы могут быть достаточно эффективными в решении удивительно широкого круга сложных проблем.

Применительно к задачам разработки программного обеспечения применение генетического алгоритма принято называть **эволюционным программированием**. Здесь цель состоит в том, чтобы разрабатывать программы, позволяя им развиваться, а не явно и жестко писать их код. Исследователи применили методы эволюционного программирования к процессу разработки программ с использованием функциональных языков программирования. Выбранный подход состоял в том, чтобы начать с набора программ, которые содержат большое разнообразие функций. Функции в этой исходной коллекции образуют "генофонд", из которого в дальнейшем и будут создаваться будущие поколения программ. Затем можно позволить эволюционному процессу работать в течение многих поколений, в надежде, что благодаря созданию каждого очередного поколения из лучших исполнителей в предыдущем поколении найденное решение целевой задачи со временем будет успешно развиваться.

11.4. Вопросы и упражнения

1. Что подразумевается под термином "знание реального мира" и каково его значение в искусственном интеллекте?
2. База данных о подписчиках журналов обычно содержит список подписчиков на каждый журнал, но не содержит список тех, кто на этот журнал не подписался. Как же тогда такая база данных позволит определить, что некоторый человек не подписался на определенный журнал?
3. Дайте общее описание проблемы фреймов.
4. Укажите три способа обучения компьютерных систем. Какой из них не предполагает прямого вмешательства человека?
5. Чем эволюционные методы отличаются от более традиционных методов решения проблем?

11.5. Искусственные нейронные сети

Несмотря на все успехи, достигнутые в области искусственного интеллекта, многие проблемы в этой области по-прежнему требуют затрат ресурсов, превосходящих возможности компьютеров, в которых используются традиционные алгоритмические подходы. Последовательности инструкций, по-видимому, не способны воспринимать и рассуждать на уровнях, сопоставимых с уровнем человеческого разума. По этой причине многие исследователи обращаются к

подходам, в которых используются явления, наблюдаемые в природе. Одним из таких подходов являются генетические алгоритмы, представленные в предыдущем разделе. Другой подход — это искусственная нейронная сеть.

Основные свойства

Искусственные нейронные сети предполагают использование такой модели компьютерной обработки, которая имитирует сети нейронов в живых биологических системах. Биологический нейрон — это отдельная клетка с входящими отростками, называемыми *дендритами*, и исходящими отростками, именуемыми *аксонами* (рис. 11.15). Сигнал, передаваемый по аксону клетки, отражает состояние этой клетки, которое может быть заторможенным или возбужденным. Состояние клетки определяется комбинацией сигналов, поступающих через дендриты клетки, которые передают клетке сигналы от аксонов других клеток через небольшие промежутки, называемые *синапсами*. Исследования показывают, что проводимость через отдельные синапсы контролируется их химическим составом. Поэтому именно химическим составом синапса определяется, окажет ли отдельный входной сигнал возбуждающее или тормозящее действие на данный нейрон. Таким образом, считается, что биологическая нейронная сеть собирает информацию (т.е. обучается) посредством регулирования этих химических связей между нейронами.

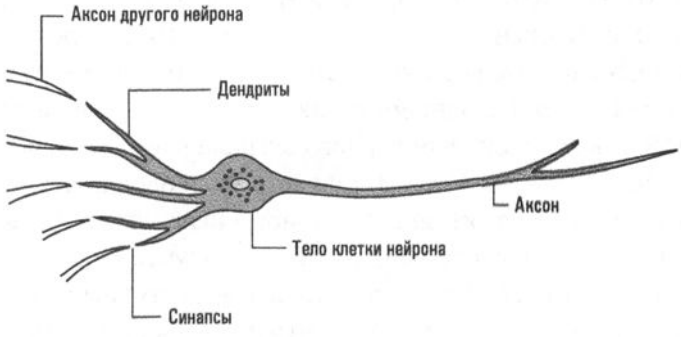


Рис. 11.15. Нейрон живой биологической системы

Нейрон в искусственной нейронной сети — это программная единица, которая имитирует это базовое понимание биологического нейрона. Он выдает на выход значение `1` или `0` в зависимости от того, превышает ли его действующий входной сигнал заданное значение, которое называется **пороговым значением** нейрона. Действующий входной сигнал представляет собой взвешенную сумму фактических входных данных, как показано на рис. 11.16. На этом рисунке нейрон представлен овалом, а связи между нейронами показаны стрелками.

Значения, полученные из аксонов других нейронов (обозначены как v_1, v_2 и v_3), используются в качестве входных данных для приведенного на рисунке нейрона. В дополнение к этим значениям каждому соединению поставлено в соответствие значение, называемое **весовым коэффициентом** (обозначается как w_1, w_2 и w_3). Действующий входной сигнал образуется как сумма значений, получаемых после перемножения каждого входного сигнала на соответствующий весовой коэффициент ($v_1w_1 + v_2w_2 + v_3w_3$). Если эта сумма превышает пороговое значение нейрона, нейрон выдает на выход сигнал `1` (имитирует возбужденное состояние); в противном случае нейрон выдает на выход сигнал `0` (имитируя заторможенное состояние).



Рис. 11.16. Процессы, происходящие внутри искусственного нейрона

Исходя из рис. 11.16, примем соглашение представлять искусственные нейроны в виде овалов. На входящей стороне нейрона для каждого входного сигнала указывается присвоенный ему весовой коэффициент, а пороговое значение данного нейрона записывается в центре овала. В качестве примера на рис. 11.17 представлен нейрон с тремя входными сигналами и пороговым значением, равным 1,5. Весовой коэффициент для первого сигнала равен `-2`, для второго сигнала — `3`, а для третьего — `-1`. Следовательно, если в блок поступают входные сигналы `1`, `1` и `0`, то его эффективный входной сигнал будет равен `1 × (-2) + + 1 × 3 + 0 × (-1) = 1`, а на выходе нейрона будет получено значение `0`. Однако если в нейрон поступят сигналы `0`, `1` и `1`, то его эффективный входной сигнал будет равен уже `0 × (-2) + 1 × 3 + 1 × (-1) = 2`, что превышает пороговое значение. В этом случае на выходе нейрона будет представлено значение `1`.

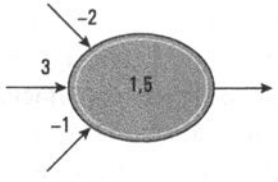


Рис. 11.17. Графическое представление искусственного нейрона

Тот факт, что весовые коэффициенты могут быть как положительными, так и отрицательными, означает, что соответствующие входные сигналы оказывают на нейрон либо возбуждающее, либо тормозящее действие. (Если весовой коэффициент отрицателен, то значение `1` на этом входе уменьшает общую сумму, что удерживает эффективный входной сигнал ниже порогового значения. И наоборот, положительный весовой коэффициент способствует повышению данным входным сигналом общей суммы и как следствие увеличивает шансы на превышение порогового значения.) Сама же абсолютная величина весового коэффициента определяет степень, с которой данный входящий сигнал способен возбуждать или тормозить нейрон. Следовательно, посредством регулирования величин весовых коэффициентов во всей искусственной нейронной сети можно запрограммировать эту сеть так, чтобы она определенным образом реагировала на изменения во входящих сигналах.

Искусственные нейронные сети топологически обычно строятся из нескольких слоев. Входные нейроны сети находятся в первом слое, а ее выходные нейроны — в последнем. Между входным и выходным слоями могут быть включены дополнительные слои нейронов (называемые *скрытыми* слоями). Каждый нейрон одного слоя связан со всеми нейронами в следующем слое. В качестве примера на рис. 11.18, *а* представлена простая нейронная сеть, запрограммированная на выдачу выходного сигнала `1`, если входные сигналы на двух ее входах различаются, и на выдачу выходного сигнала `0` в противном случае. Однако, если изменить весовые коэффициенты в этой сети так, как показано на рис. 11.18, *б*, мы получим нейронную сеть, которая будет выдавать на выходе `1`, если входные сигналы на двух ее входах будут равны `1`, и выдавать на выходе `0` в противном случае.

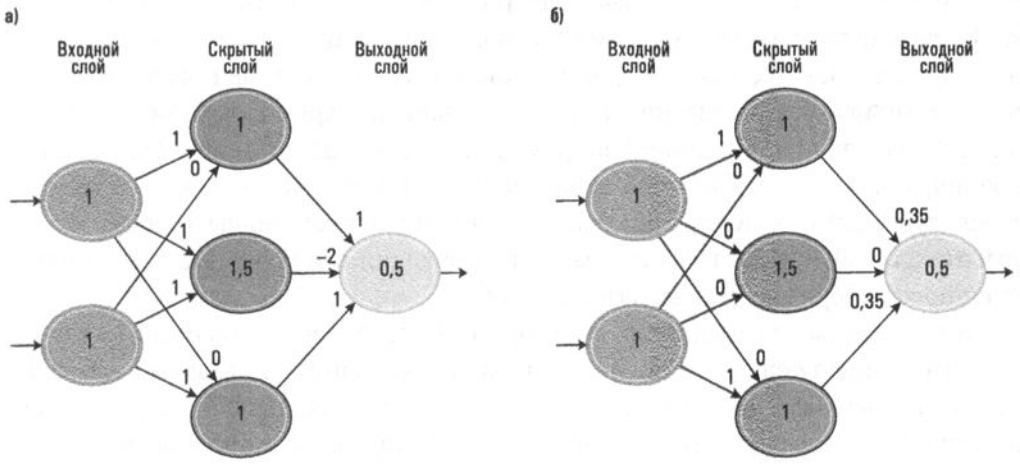


Рис. 11.18. Нейронная сеть с двумя различными программами

Следует отметить, что на рис. 11.18 конфигурация нейронной сети гораздо проще, чем конфигурация реальной биологической сети. Человеческий мозг содержит приблизительно 10^{11} нейронов, причем каждый из них имеет примерно 10^4 синапсов. В действительности дендриты биологического нейрона обычно настолько многочисленны, что производят впечатление волокнистой сетки вместо "букета" из нескольких отдельных щупалец, как было показано на рис. 11.15.

Обучение искусственных нейронных сетей

Важной особенностью искусственных нейронных сетей является то, что они не программируются в традиционном смысле этого слова, а *обучаются*. Иначе говоря, программист не определяет предварительно значения весовых коэффициентов, необходимых для решения конкретной проблемы, чтобы затем "вписать" эти значения в сеть. Вместо этого искусственная нейронная сеть подбирает правильные значения весовых коэффициентов в процессе обучения с учителем (раздел 11.4), предусматривающим многократно повторяющуюся процедуру, в которой к сети применяются входные данные из обучающей выборки, а затем ее весовые коэффициенты корректируются небольшими приращениями таким образом, чтобы сеть демонстрировала действия, приближающиеся к желаемому поведению.

Интересно отметить, что к задаче обучения искусственных нейронных сетей также применялись методы генетического алгоритма. В частности, для обучения нейронной сети можно случайным образом сгенерировать несколько наборов весовых коэффициентов для всех элементов сети, после чего каждый из этих наборов будет служить в качестве хромосомы для генетического алгоритма. Затем шаг за шагом элементам сети могут назначаться весовые коэффициенты, представленные каждой хромосомой, и в каждом случае поведение сети будет тестироваться на различных входных данных. Хромосомы, показавшие наименьшее количество ошибок по результатам проведенного процесса тестирования, могут быть с большей вероятностью выбраны в качестве родителей для следующего поколения хромосом. В многочисленных проведенных экспериментах такой подход в конечном итоге приводил к получению достаточно успешного набора весовых коэффициентов.

Давайте рассмотрим пример, в котором обучение искусственной нейронной сети с целью решения проблемы оказалось успешным и, возможно, более продуктивным, чем попытки отыскать соответствующее решение с помощью традиционных методов программирования. Исследуемая проблема является одной из тех, с которыми может столкнуться робот при попытках разобраться в окружающей среде с помощью информации, поступающей от его видеокамеры.

Предположим, например, что робот должен научиться различать стены комнаты, выкрашенные в белый цвет, и пол, выкрашенный черным. На первый взгляд, это может показаться легкой задачей: просто классифицируйте белые пиксели как элементы стены, а черные пиксели — как элементы пола. Однако, поскольку робот может смотреть в разные стороны и перемещаться по комнате, изменение условий освещения может привести к тому, что в одних случаях стена будет выглядеть серой, тогда как в других случаях серым может выглядеть пол. А это означает, что робот должен научиться четко различать стены и пол при самых разных условиях освещения.

Чтобы решить эту задачу, можно было бы создать искусственную нейронную сеть, входные данные которой состоят из значений, определяющих цветовые характеристики отдельного пикселя в изображении, а также значения, указывающего общую яркость изображения в целом. Затем можно было бы провести обучение этой сети, предоставив ей многочисленные примеры пикселей, представляющих отдельные части стен и полов при различных условиях освещения.

Помимо простых задач обучения (таких, как классификация пикселей), искусственные нейронные сети использовались и для изучения сложного интеллектуального поведения, о чем свидетельствует проект ALVINN, упоминавшийся в предыдущем разделе. И действительно, система ALVINN представляла собой искусственную нейронную сеть, строение которой было на удивление простым (рис. 11.19). На ее вход поступали данные от матрицы датчиков размером 30 × 32, каждый из которых отслеживал собственную часть видеоизображения дороги впереди и передавал свои данные каждому из четырех нейронов на скрытом слое. (В результате каждый из этих четырех нейронов имел 960 входов.) Выход каждого из этих четырех нейронов скрытого слоя был подключен к каждому из тридцати выходных нейронов, выходные значения которых определяли направление, в котором прилагались управляющие воздействия. Возбуждение нейрона на одном конце ряда из тридцати нейронов указывало на необходимость резкого поворота налево, тогда как возбуждение нейрона на другом конце этого ряда указывало на необходимость выполнить резкий поворот направо.

Нейронная сеть системы ALVINN обучалась посредством "наблюдения" за движениями человека; при этом она принимала собственные управляющие решения, сравнивала их с решениями человека, а затем вносила небольшие изменения в весовые коэффициенты, чтобы приблизить свои решения к решениям человека. Однако здесь возникла интересная побочная проблема. Хотя, следуя этой простой технике, система ALVINN научилась успешно управлять фургоном, эта система не нашла способа восстанавливаться после совершения ошибки. По этой причине данные, собранные при записи действий человека, были

искусственно дополнены так, чтобы включить и ситуации восстановления. (Один из подходов к такой тренировке по восстановлению, который рассматривался первоначально, заключался в том, чтобы человек разворачивал транспортное средство ошибочным образом, чтобы затем система ALVINN могла наблюдать за действиями человека, направленными на возвращение к нормальному движению, и следовательно, учиться самостоятельно их выполнять. И хотя система ALVINN деактивировалась в тот момент, когда водитель-человек выполнял неправильный разворот, она все же научилась и выполнять такие развороты, и возвращаться к нормальному движению после них — это особенность, явно нежелательная в данном проекте.)

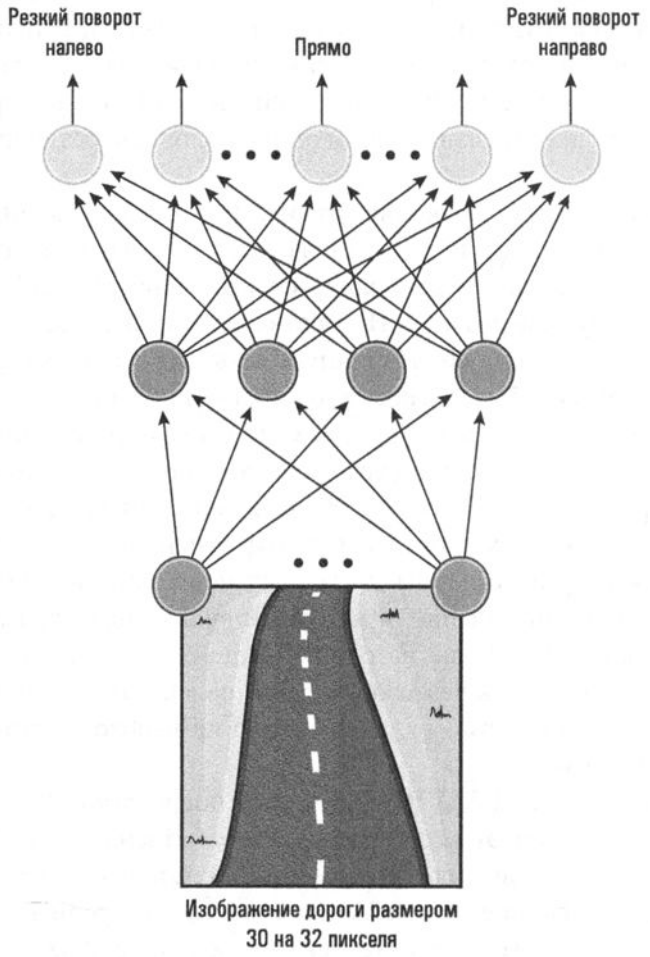
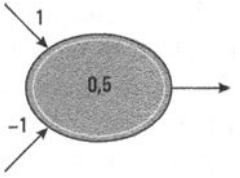


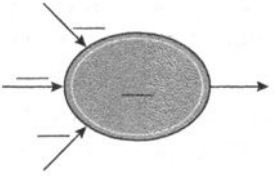
Рис. 11.19. Структура нейронной сети системы ALVINN

11.5. Вопросы и упражнения

1. Каков будет выходной сигнал приведенного ниже нейрона, если оба его входных сигнала будут равны `1`? А каким будет выходной сигнал, если на входы будут поданы сочетания сигналов `0`, `0`; `0`, `1` и `1`, `0`?



2. Отрегулируйте весовые коэффициенты входов и пороговое значение приведенного ниже нейрона так, чтобы его выходной сигнал был равен `1` тогда и только тогда, когда по крайней мере два из его входов равны `1`.



3. Определите проблему, которая может возникнуть при обучении искусственной нейронной сети.

11.6. Робототехника

Робототехника — это область искусственного интеллекта, изучающая возможности и способы создания физических автономных агентов, проявляющих разумное поведение. Как и все агенты, роботы должны быть способны воспринимать, рассуждать и действовать в своей среде. Следовательно, исследования в области робототехники охватывают фактически все прочие области искусственного интеллекта, а также в значительной степени опираются на механику и электротехнику.

Чтобы взаимодействовать с миром, роботам нужны механизмы для манипулирования объектами и перемещения самих себя. В ранний период развития робототехники эта область была тесно связана с разработкой манипуляторов, чаще всего механических рук с локтями, запястьями, ладонями или инструментами. Исследования касались не только того, как такие устройства могли бы манипулировать своими элементами, но и того, как можно было бы сохранять

и использовать знания об их расположении и ориентации. (Вы можете закрыть глаза, а затем без затруднений прикоснуться пальцем к кончику носа, поскольку ваш мозг постоянно контролирует, где находятся ваши нос и палец.) Со временем руки роботов становились все более и более ловкими, и сейчас, пользуясь чувством осязания, основанным на силовой обратной связи, они способны успешно манипулировать даже яйцами и бумажными стаканчиками.

В последнее время разработка более быстрых и легких компьютеров привела к расширению исследований в области создания мобильных роботов, способных самостоятельно перемещаться. Возможность достижения необходимой мобильности привела к бурному росту творческих замыслов. Исследователи в области передвижения роботов достигли значительных успехов в разработке роботов, которые плавают, как рыбы, летают, как стрекозы, прыгают, как кузнечики, и ползают, как змеи.

Колесные роботы также очень популярны, поскольку их относительно легко проектировать и строить, однако им свойственны ограничения в типе местности, по которой они способны перемещаться. Целью многих проводящихся сейчас исследований является преодоление этого ограничения за счет использования различных комбинаций колес и гусениц для подъема по лестницам или перекатывания через камни. Например, в марсоходах NASA специально разработанные колеса используются для перемещения по пересеченной местности с каменистой почвой.

Роботы с конечностями-ногами предлагают бо́льшую мобильность, но при этом их конструкция значительно сложнее. Например, двуногие роботы, разрабатываемые так, чтобы ходить как люди, должны постоянно контролировать и корректировать свое положение в пространстве, иначе они просто упадут. Тем не менее такие трудности вполне преодолимы, примером чему служит двуногий человекоподобный робот по имени Асимо, разработанный в корпорации Honda, — он может подниматься по лестнице и даже бегать.

Несмотря на большие успехи в создании манипуляторов и средств передвижения, большинство роботов по-прежнему автономны лишь в очень ограниченной степени. Руки промышленного робота обычно жестко запрограммированы для выполнения конкретного задания и работают без датчиков (при условии, что детали будут предоставлены им точно в предусмотренных заранее положениях). Другие мобильные роботы, такие как марсоходы НАСА и военные беспилотные летательные аппараты, в значительной степени полагаются на людей-операторов при необходимости проявления разумной деятельности.

Преодоление этой зависимости от людей является основной целью современных исследований. Один из аспектов проблемы состоит в том, что автономный робот должен иметь необходимые знания об окружающей среде, а также о том, в какой степени ему нужно заранее планировать свои действия. Один из подходов заключается в создании роботов, которые ведут подробные записи о

своем окружении, содержащие перечень объектов и данные об их местоположении, на основании которых они разрабатывают точные планы своих будущих действий. Исследования в этом направлении в значительной степени зависят от прогресса в представлении и хранении знаний, а также от совершенствования методов проведения рассуждений и разработки планов.

Альтернативный подход заключается в разработке лишь реагирующих (или, как говорят, *реактивных*) роботов, которые, вместо того чтобы вести сложные записи и прилагать огромные усилия для составления подробных планов предстоящих действий, просто применяют простые наперед заданные правила взаимодействия с окружающим миром для определения своего поведения в каждый очередной момент времени. Сторонники реактивной робототехники утверждают, что, планируя длительную поездку на автомобиле, люди заранее не составляют всеобъемлющих, детальных планов. Вместо этого они просто выбирают основные дороги, оставляя такие вопросы, как, где поесть, где отдохнуть и как поступить в случае необходимости объезда, для рассмотрения по мере их возникновения. Аналогичным образом реактивный робот, которому необходимо перемещаться по многолюдному коридору или переходить из одного здания в другое, заранее не разрабатывает очень подробный план своего движения; вместо этого он применяет простые правила преодоления возникающих препятствий по мере их появления. Такой подход используется и в самом продаваемом роботе в истории — в пылесосе Roomba компании iRobot, который перемещается по полу в реактивном режиме, не заботясь о запоминании детальных сведений о расположении мебели и других возможных препятствий. В конце концов, домашнее животное в следующий раз, вероятно, уже не будет находиться в том же самом месте.

Конечно, ни один из упомянутых подходов, по-видимому, нельзя считать лучшим для всех возможных ситуаций. По-настоящему автономные роботы, скорее всего, будут использовать несколько уровней рассуждения и планирования, применяя высокоуровневые методы для постановки и достижения основных целей и используя низкоуровневые реактивные системы для достижения произвольно возникающих промежуточных подцелей. Пример такого многоуровневого мышления можно найти в устройствах, участвующих в конкурсе Robocup — международном соревновании команд по футболу среди роботов, — который одновременно служит и форумом для исследований по разработке команды роботов, способных победить команды мирового уровня по футболу среди людей к 2050 году. Здесь акцент делается не только на создании отдельных экземпляров мобильных роботов, которые могут "бить" по мячу, но и на создании *команды* роботов, которые взаимодействуют друг с другом для достижения общей цели. Эти роботы должны не только двигаться и рассуждать о своих действиях, но и уметь рассуждать о действиях своих товарищей по команде и своих противников.

Роботы в доме



Роботизированные пылесосы, такие как популярная линия Roomba от компании iRobot, уже стали широко доступными потребительскими устройствами. Используя комбинацию оптических и инфракрасных датчиков, а также датчиков удара и падения, усовершенствованная модель может перемещаться по всему этажу дома, запоминая, где она находилась при обходе вокруг ножек мебели, лестничных клеток и домашних животных. Программное обеспечение автономного агента, управляющее роботом, обеспечивает достаточный уровень разумности, позволяющий ему вернуться на зарядную станцию, когда встроенный пылесборник заполнится или заряд батареи будет исчерпан. В навигационном алгоритме используются развертки по длине комнаты, обеспечивающие покрытие больших открытых участков, в комбинации со вторым проходом, предназначенным для исследования границ с целью уборки вокруг препятствий и возможного поиска недоступных мест.

Приложения для смартфонов, обеспечивающие связь с этим устройством, можно использовать для отслеживания состояния робота, планирования автоматических запусков и даже для создания окончательной карты очищенной площади с отмеченными особенно загрязненными участками.

Другим примером исследований в области робототехники является область, известная как "эволюционная робототехника", в которой теории эволюции применяются для разработки новых схем как в отношении правил реагирования

низкого уровня, так и в отношении рассуждений высокого уровня. Теория выживания наиболее приспособленных здесь используется для разработки таких, например, устройств, которые на протяжении нескольких поколений приобретают собственные средства поддержания баланса или мобильности. Во многих исследованиях в этой области проводится различие между системой внутреннего контроля робота (в основном программным обеспечением) и физической структурой его тела. Например, система управления плавающим роботом-головастиком была перенесена на аналогичного робота с ногами. Затем в отношении этой системы управления были применены эволюционные методы, чтобы получить робота, который мог бы ползать. В других случаях эволюционные методы применялись к физическому телу робота, например, с целью определения оптимального положения датчиков при выполнении конкретной задачи. В более сложных исследованиях анализировались возможности развития программных систем управления роботов одновременно с совершенствованием физических структур их тела.

Перечислить все впечатляющие результаты исследований в области робототехники было бы непосильной задачей. Конечно, наши нынешние роботы еще очень далеки от мощных роботов, фигурирующих в фантастических фильмах и романах, но они уже достигли впечатляющих успехов в решении конкретных задач. У нас есть роботы, которые могут управлять автомобилем в дорожных пробках, вести себя, как домашние любимцы, собаки, и даже успешно доставлять боеголовки к назначенным целям в любых условиях. Однако, по достоинству оценив все эти успехи, мы все же должны отметить, что привязанность, которую мы испытываем к искусственной собаке, или удивительная мощь умного оружия поднимают серьезные социальные и этические вопросы, бросающие вызов обществу. Наше будущее — это то, что мы сами себе уготовили.

11.6. Вопросы и упражнения

1. Чем реактивный подход к поведению робота отличается от более традиционного варианта поведения, основанного на четком планировании?
2. Какие из текущих тем исследований в области робототехники вам известны?
3. На каких двух уровнях эволюционные теории могут применяться к разработке роботов?

11.7. Осмысливание последствий

Несомненно, что прогресс, достигнутый в области исследований искусственного интеллекта, предполагает значительную потенциальную возможность принести пользу человечеству, поэтому очень легко поддаться энтузиазму, вызванному этой возможностью. Но нельзя забывать и о таящейся в будущем потенциальной угрозе, последствия которой могут быть такими же разрушительными, как и размеры полученной выгоды. Различие довольно часто заключается всего лишь в точке зрения или в положении, занимаемом человеком в обществе: что для одного выгодно, для другого может оказаться губительным. Будет весьма полезно остановиться на некоторое время и взглянуть на успехи технологии с противоположной точки зрения.

Одни смотрят на бурное развитие технологий как на подарок человечеству, подразумевая освобождение человека от утомительных, традиционных обязанностей и прокладывание пути к более удобному и приятному образу жизни. Однако другие рассматривают это же явление как проклятье, лишающее граждан ремесел и перераспределяющее доходы в пользу сильных мира сего. Приблизительно так звучало письмо истинного гуманиста Махатмы Ганди из Индии. Он неоднократно приводил доводы в пользу того, что для Индии будет лучше, если заменить большие текстильные фабрики прялками, расположенными в домах сельских жителей. Таким образом, как утверждал он, централизованное серийное производство, предоставляющее работу лишь избранным, будет заменено рассредоточенной системой массового производства, приносящей выгоду б*о*льшему количеству людей.

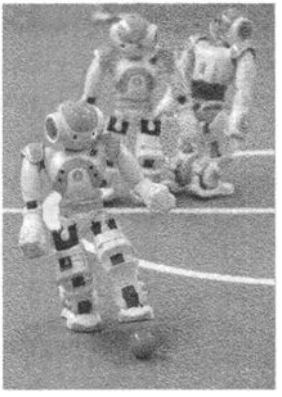
История полна революций, причиной которых являлись диспропорции в распределении богатств и привилегий. Если нынешняя бурно развивающаяся технология будет способствовать усилению подобных различий, последствия могут оказаться катастрофическими.

Но последствия построения все более и более "разумных" машин существенно глубже (и более фундаментальны), чем последствия борьбы классов и различных группировок в обществе. Эта проблема поразила человеческое самомнение в самое сердце. В XIX веке общество было приведено в смятение теорией Чарльза Дарвина об эволюции и мыслью о том, что человек мог произойти от низших форм жизни. И как должно реагировать общество, поставленное перед фактом возможности неудержимой атаки со стороны машин, умственные способности которых бросают вызов человеку?

В прошлом технологии развивались медленно, давая время на сохранение нашего самомнения за счет пересмотра существовавших понятий об интеллекте. Наши древние предки наделили бы механические устройства XIX века сверхъестественным интеллектом, а мы сейчас отказываем этим машинам в каких бы то ни было признаках сознания. Но как же отреагирует человечество,

Роботы делают историю

а) Футбольный робот бьет по мячу во время проведения турнира RoboCup German Open 2010 15 апреля 2010 года в Магдебурге, Восточная Германия.
(© Jens Schlueter/SFP/gettyimages/Newscom)



б) Робот "Босс" от компании Tartan Racing, победитель турнира Urban Challenge — конкурса, спонсируемого организацией DARPA с целью создания транспортных средств, способных самостоятельно совершать поездки в условиях городской среды.
(© DARPA)



в) Один из роверов управления НАСА — робот-геолог для проведения исследований поверхности Марса.
(Публикуется с разрешения NASA/JPL-Caltech)



если машины действительно бросят вызов интеллекту человека или, что более вероятно, если возможности машин будут развиваться быстрее, чем наши способности приспосабливаться к этому?

Можно получить представление о потенциальной реакции человечества на машины, бросающие вызов нашему интеллекту, обратившись к реакции общества на внедрение тестов IQ в середине XX века. Считалось, что эти тесты определяют уровень интеллекта ребенка. Дети в Соединенных Штатах часто классифицировались по результатам этих тестов и соответственно направлялись в те или иные образовательные программы. В результате широкие образовательные возможности были открыты для тех детей, которые показали хорошие результаты по этим тестам, тогда как дети, которые показали плохие результаты, направлялись лишь на упрощенные, коррекционные программы обучения. Короче говоря, при наличии шкалы, позволяющей измерять интеллект человека, общество, как правило, игнорирует возможности тех, кто оказался в нижней части этой шкалы. Как же тогда общество справится с ситуацией, когда "интеллектуальные" возможности машин станут сопоставимыми или хотя бы просто *покажутся* сопоставимыми с возможностями людей? Откажется ли общество от тех, чьи способности считаются "низшими" по сравнению с машинами? Если да, то каковы будут последствия для этих членов общества? Должно ли достоинство человека зависеть от результатов сравнения его способностей с возможностями машины?



Основные положения для запоминания

- **Инновации, обеспечиваемые использованием компьютерной техники, вызывают юридические и этические проблемы.**

Мы уже видим, как в некоторых областях интеллектуальная мощь человека ставится машинами под сомнение. Машины уже способны обыграть мастеров игры в шахматы, компьютеризированные экспертные системы способны давать медицинские советы, а управление инвестициями некоторым программам удается даже лучше, чем профессиональным брокерам. Как повлияют подобные системы на вовлеченных в эти занятия людей и их представление о самих себе? Как влияет на человеческое чувство собственного достоинства тот факт, что во все большем количестве отраслей машины уже превосходят человека?

Многие возражают, что интеллект, которым обладают машины, всегда будет радикально отличаться от человеческого, поскольку человек имеет биологическое происхождение, а машина — нет. Поэтому, как утверждают они, машина никогда не сможет воспроизвести процесс принятия решений человеком.

Машины могут прийти к тем же выводам, что и человек, но эти умозаключения будут сделаны не на том основании, на котором их делает человек. Но тогда как оценить, до какой степени различаются эти два интеллекта и будет ли этично для общества следовать по пути, указанному нечеловеческим интеллектом?

В своей книге *Computer Power and Human Reason* Джозеф Уэйзенбаум[1] (Joseph Weizenbaum) следующим образом возражает против неконтролируемого применения искусственного интеллекта.

> Компьютеры могут выносить судебные вердикты, компьютеры могут принимать решения в отношении психиатрических диагнозов. Они могут подбрасывать в воздух монеты более изощренно, чем самые упорные из людей. Все дело в том, что им нельзя давать такие задания. Они даже могут прийти к "правильному" выводу в некоторых случаях, но всегда и непременно на таких основаниях, которые человек не пожелал бы принять во внимание.
>
> В прошлом уже имело место множество дебатов относительно проблемы "Компьютер и разум". Главный вывод, который я сделал, состоит в том, что соответствующая проблема не является ни технологической, ни даже математической — она *этическая*. Она не может быть устранена с помощью вопросов, начинающихся со слов "Может ли..." Пределы области применения компьютеров в итоге могут быть установлены исключительно в терминах долженствования. Наиболее простое понимание проблемы заключается в том, что, поскольку мы не имеем на данный момент возможности сделать компьютер мудрым, мы и не должны давать ему заданий, требующих наличия мудрости.

Вы можете возразить, что большая часть этого раздела граничит с научной фантастикой, а не с компьютерными науками. Однако не так давно многие отвергали вопрос "Что произойдет, если компьютеры захватят общество?" по причине подобного же отношения: "Этого никогда не произойдет!" Тем не менее во многих отношениях этот день *уже* наступил. Если компьютеризированная база данных ошибочно сообщит, что у вас плохой кредитный рейтинг, судимость или имеется перерасход по кредитной карточке, что в конечном счете будет принято во внимание — ложные утверждения компьютера или ваше заявление о невиновности? А если неисправная навигационная система показывает, что покрытая туманом взлетно-посадочная полоса якобы находится совсем не в том месте, где должен приземлиться самолет? Если машина используется для прогнозирования реакции общественности на различные политические решения, то какое решение примет политик? Сколько раз всевозможные служащие и

[1] Joseph Weizenbaum, "Computer Power and Human Reason: From Judgment to Calculation". W.H. Freeman, 1976.

работники не смогли вам помочь, потому что их компьютерная система "зависла"? Кто (или что) в подобных случаях должен нести ответственность? Разве мы уже не сдали наше общество машинам?

11.7. Вопросы и упражнения

1. Какая часть современного общества выживет, если убрать все машины и механизмы, появившиеся за последние 100 лет? А если за последние 50 или 20 лет? В каких местах нашей планеты будут находиться те, кто выживут?
2. В какой степени вашу жизнь контролируют машины? Кто контролирует машины, влияющие на вашу жизнь?
3. Откуда поступает информация, на которой основываются ваши решения, принимаемые в повседневной жизни? А что можно сказать о самых важных для вас решениях? Насколько вы доверяете точности этой информации? Почему?

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. Как было продемонстрировано в разделе 11.2, люди могут употреблять вопросы для целей, отличных от опрашивания. Вот один из подобных вопросов: "Вы знаете, что у вас шина спустила?" Обычно его целью является информирование человека, а не пустое любопытство. Приведите примеры вопросов, используемых для убеждения, предостережения, осуждения.
2. Проанализируйте автомат продажи газированной воды, рассматривая его в качестве агента. Что собой представляют его датчики? Каковы его исполнительные механизмы? Какой уровень реакции (рефлекторная, основанная на знаниях, направленная на достижение цели) он демонстрирует?
3. Определите тип ответной реакции объекта в каждом из приведенных ниже случаев как рефлекторный, основанный на знаниях или направленный на достижение цели. Обоснуйте свои ответы.
   а. Компьютерная программа для перевода текстов с немецкого языка на английский.

б. Термостат, включающий систему обогрева, когда температура в помещении падает ниже текущего установленного значения.

в. Пилот, благополучно посадивший самолет на взлетно-посадочную полосу аэродрома.

4. Если исследователь использует компьютерную модель для изучения способностей к запоминанию человеческого мозга, должны ли разработанные для этой цели программы обязательно использовать память компьютера на пределе его возможностей? Объясните свою точку зрения.

5. Приведите несколько примеров декларативных знаний. Приведите несколько примеров процедурных знаний.

***6.** В контексте парадигмы объектно-ориентированного программирования какие части объектов используются для хранения декларативных знаний? Какие части объектов предназначены для хранения процедурных знаний?

7. Какие из перечисленных ниже действий, по вашему мнению, являются ориентированными на выполнение, а какие — на имитацию?

а. Проектирование автоматизированной системы трансфера (часто используется в аэропортах между терминалами).

б. Разработка модели, предсказывающей путь урагана.

в. Проектирование базы данных системы веб-поиска, предназначенной для создания и поддержки индекса документов, хранящихся в Сети.

г. Разработка модели экономики государства для проверки экономических теорий.

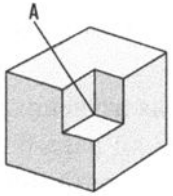
д. Разработка программы слежения за основными показателями состояния больного.

8. Сегодня некоторые телефонные звонки в различных организациях и учреждениях обрабатываются автоматическими автоответчиками, в которых для общения с абонентом используют программы распознавания речи и синтеза человеческого голоса. Как вы считаете, пройдут ли эти системы тест Тьюринга? Поясните свой ответ.

9. Определите небольшой набор геометрических свойств, которые могут быть использованы для различения символов F, E, L и T.

***10.** Опишите сходство между технологией идентификации символов посредством сравнения их с образцами-шаблонами и кодом с исправлением ошибок, обсуждавшимся в главе 1.

11. Опишите две трактовки приведенного ниже чертежа, основанные на том, является ли обозначенный буквой A угол выпуклым или вогнутым.



12. Сравните значения предложных оборотов в следующих двух предложениях (они различаются всего лишь одним словом). Как можно запрограммировать машину, чтобы она воспринимала подобные различия?

The pigpen was built by the barn (Хлев был выстроен рядом с сараем.)
The pigpen was built by the farmer (Хлев был выстроен фермером.)

13. Чем будут различаться результаты грамматического анализа двух приведенных ниже предложений? Чем будут различаться результаты их семантического анализа?

An awesome sunset was seen by Andrea. (Потрясающий закат был увиден Андреа.)
Andrea saw an awesome sunset. (Андреа увидела потрясающий закат.)

14. Чем будут различаться результаты грамматического анализа двух приведенных ниже предложений? Чем будут различаться результаты их семантического анализа?

Если $X < 10$, то вычесть 1 из X, иначе прибавить 1 к X.
Если $X > 10$, то прибавить 1 к X, иначе вычесть 1 из X.

15. В тексте этой главы кратко обсуждалась проблема понимания естественных языков в сравнении с формальными языками программирования. В качестве примера сложностей, возникающих при работе с естественным языком, укажите ситуации, в которых один и тот же вопрос "Ты знаешь, который час?" будет иметь различные значения.

16. Изменения в контексте выражения могут изменить как его смысл, так и значимость. Исходя из контекста, представленного на рис. 11.3, покажите, как изменится значимость предложения "Мери ударила Джона" при изменении дат рождения с середины 2010-х на конец 2000-х. А если одну дату изменить на 1980-е, а другую — на конец 2000-х?

17. Нарисуйте семантическую сеть, представляющую информацию из следующего абзаца.

Дон бросил мяч Джеку, который послал его в центр поля. Центральный полевой игрок попытался поймать его, но мяч отскочил рикошетом от стенки.

18. Иногда способность ответить на вопрос зависит от знания пределов осведомленности, поскольку это само по себе является дополнительным фактом. Например, предположим, что существуют две базы данных — **А** и **Б**. Обе содержат полный список работников, принимающих участие в программе страхования здоровья компании, но только в базе **А** есть сведения о том, что этот список *полный*. Какой вывод позволяет сделать база **А** (и не позволяет сделать база **Б**) относительно человека, не внесенного в список?

19. Приведите пример, в котором предположение о замкнутости мира приводит к противоречию.

20. Приведите два примера, в которых обычно используется предположение о замкнутости мира.

21. В контексте порождающих систем какие различия существуют между графом состояний и деревом поиска?

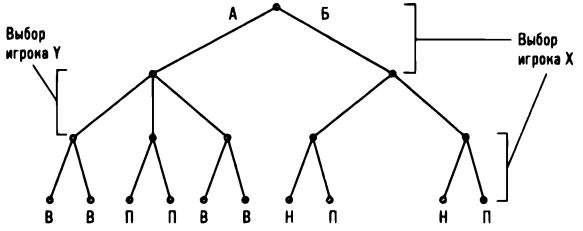
22. Проанализируйте задачу сборки кубика Рубика в терминах порождающих систем. (Что здесь является состоянием, порождением и т.д.?)

23. **а.** Предположим, что дерево поиска представляет собой двоичное дерево и для достижения цели требуется восемь порождений. Какое наибольшее количество узлов может быть в дереве при достижении целевого состояния, если дерево построено по принципу горизонтального поиска?

б. Объясните, как можно уменьшить общее количество узлов, анализируемых во время поиска, путем одновременного проведения двух поисков: один начинается из исходного состояния системы, а другой ведется в обратном направлении начиная от ее целевого состояния. Оба эти поиска продолжаются до тех пор, пока они не встретятся. (Предположим, что дерево поиска, записывающее состояния, найденные при обратном поиске, также является двоичным деревом и что оба поиска выполняются с одинаковой скоростью.)

24. В тексте главы упоминалось, что порождающая система часто используется как технологический прием для того, чтобы делать выводы из известных фактов. Состояния системы — это факты, гарантированно истинные на каждом этапе процесса рассуждения, а порождения — это правила логики, используемые для оперирования известными фактами. Определите несколько правил логики, необходимых для вывода утверждения "Джон — высокий" из следующих известных фактов: "Джон — баскетболист", "Баскетболисты не низкие", "Джон либо высокий, либо низкий".

25. Приведенное ниже дерево представляет возможные ходы в некоторой игре с противником и показывает, что игрок *X* в настоящий момент имеет выбор между ходами А и Б. После хода игрока *X* игрок *Y* выбирает свой ход, после чего игрок *X* делает последний ход в игре. Листовые узлы дерева помечены буквами В, П и Н в зависимости от того, заканчивается игра выигрышем, проигрышем или ничьей для игрока *X*. Какой ход выберет игрок *X* — А или Б? Почему? Чем выбор порождения в игре с противником отличается от подобного выбора в одиночной игре, такой как головоломка "Восьмерка"?



26. Проанализируйте игру в шашки как порождающую систему и опишите эвристику, которая может быть использована для определения того, какое из двух состояний ближе к цели. Как система контроля в этих условиях будет отличаться от игры с единственным игроком, такой как "Восьмерка"?

27. Если рассматривать правила алгебраических преобразований как порождения, то проблема упрощения алгебраических выражений может быть автоматизирована в контексте порождающих систем. Определите набор алгебраических порождений, которые позволили бы упростить исходное уравнение $3/(2x - 1) = 6/(3x + 1)$ до вида $x = 3$. Какие эмпирические (эвристические) правила необходимо использовать для выполнения подобного алгебраического упрощения?

28. Нарисуйте дерево поиска, порожденное горизонтальным поиском при попытке решить головоломку "Восьмерка", для приведенного ниже исходного состояния, причем без использования какой бы то ни было эвристической информации.

	1	3
4	2	5
7	8	6

29. Нарисуйте дерево поиска, которое будет порождено алгоритмом поиска по первому наилучшему совпадению, представленным на рис. 11.10, при попытке решить головоломку "Восьмерка" для начального состояния, показанного в задании 28, считая эвристической величиной количество фишек, находящихся не на своих местах.

30. Нарисуйте дерево поиска, которое будет порождено алгоритмом поиска по первому наилучшему совпадению, представленным на рис. 11.10, при попытке решить головоломку "Восьмерка" для показанного ниже начального состояния, используя эвристическую оценку, описанную в разделе 11.3.

1	2	3
5	7	6
4		8

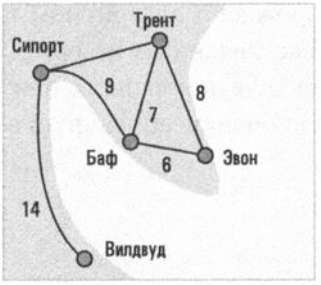
31. Почему при решении головоломки "Восьмерка" лучше в качестве эвристической величины использовать параметр, описанный в разделе 11.3, а не число фишек, находящихся не на своих местах?

32. Каковы различия между методом принятия решения о том, какую из частей списка следует рассматривать при выполнении двоичного поиска (см. раздел 5.5), и методом выбора ветви при эвристическом поиске?

33. Следует отметить, что если состояние на графе состояний порождающей системы имеет очень малую эвристическую величину по сравнению с остальными состояниями и существует порождение, переводящее это состояние само в себя, то приведенный на рис. 11.10 алгоритм может попасть в бесконечный цикл и рассматривать это состояние снова и снова. Покажите, что бесконечного зацикливания процесса поиска можно избежать, если установить стоимость выполнения любого порождения в системе равной единице. При этом предполагаемую стоимость следует вычислять как сумму эвристической величины и стоимости достижения данного состояния по всему пройденному пути.

34. Какие эвристические показатели можно использовать при поиске маршрута между двумя городами на большой карте дорог?

35. Нарисуйте дерево поиска с четырьмя уровнями, которое будет создано при использовании алгоритма поиска по первому наилучшему совпадению, представленному на рис. 11.10, в процессе поиске маршрута из города Трента в город Вилдвуд. Каждый узел в дереве поиска будет представлять город на карте. Начните с узла для города Трент. При расширении дерева от текущего узла добавляйте в него узлы только тех городов, которые непосредственно связаны с городом, представленным текущим узлом. В каждом узле запишите расстояние по прямой от данного города до Вилдвуда и используйте это значение в качестве эвристического. Имеет ли алгоритм поиска по первому наилучшему совпадению определенный недостаток в своей работе? Если да, то как можно откорректировать его работу?



Расстояние по прямой линии до Вилдвуда от городов:

Звон	10
Баф	8
Трент	15
Сипорт	13

36. Алгоритм A* представляет собой модификацию алгоритма поиска по первому наилучшему совпадению, в которой в исходный алгоритм внесены два существенных изменения. Во-первых, в нем записывается *фактическая стоимость* достижения состояния. В случае маршрута на карте фактической стоимостью является пройденное расстояние. Во-вторых, при выборе узла для расширения выбирается тот узел, сумма фактической стоимости и оценочной стоимости которого является наименьшей. Нарисуйте дерево поиска для условий задания 35, которое будет получено в результате внедрения двух этих изменений. Для каждого узла запишите расстояние, пройденное от исходного узла до данного города, оценочную стоимость достижения цели, а также их сумму. Каким будет путь из Трента в Вилдвуд?

37. Укажите два свойства, которыми должны обладать эвристические показатели, чтобы они могли оказаться полезными в порождающих системах.

38. Предположим, что у вас есть два ведра: одно емкостью 3 литра, другое — 5 литров. Вы можете переливать воду из одного ведра в другое, опорожнять ведро или наполнять его. Ваша задача — налить 4 литра воды в 5-литровое ведро. Опишите, как решение этой задачи может быть представлено в виде порождающей системы.

39. Предположим, что вашей обязанностью является наблюдение за погрузкой двух грузовиков, каждый из которых может перевозить до 14 т груза. Груз состоит из набора ящиков общей массой 28 т, но масса каждого ящика разная (она указана на нем). Какой эвристический показатель можно было бы использовать при распределении ящиков между машинами?

40. Что из следующего является примерами метарассуждений?

а. Он ушел давно, поэтому он должен уйти уже далеко.

б. Поскольку я обычно принимаю неправильное решение, а последние два решения были правильными, я отменю свое следующее решение.

в. Я устал, поэтому я, вероятно, не могу ясно мыслить.

г. Я устал, поэтому, думаю, я вздремну.

41. Поясните, как способность человека успешно решать проблему фреймов помогает людям находить потерянные предметы.

42. **а.** В каком смысле обучение подражанием похоже на обучение с учителем?

б. В каком смысле обучение подражанием отличается от обучения с учителем?

43. Для искусственной нейронной сети, приведенной на рис. 11.18, подберите такие значения весовых коэффициентов и пороговых величин, чтобы она давала на выходе значение 1, когда оба входных сигнала одинаковы (два 0 или две 1), и значение 0, когда они различны (один сигнал равен 1, другой — 0).

44. Начертите диаграмму, подобную изображенной на рис. 11.5 и представляющую процесс упрощения алгебраического выражения $7x + 3 = 3x - 5$ до вида $x = -2$.

45. Дополните ваш ответ на предыдущее задание с целью представления других путей, которыми сможет следовать система контроля при попытках решить поставленную в этом задании задачу.

46. Начертите диаграмму, подобную изображенной на рис. 11.5 и представляющую процесс рассуждений, которые позволят сделать заключение "Полли может летать" исходя из следующих известных фактов. "Полли — попугай", "Попугай — птица" и "Все птицы умеют летать".

47. В противовес принятому в предыдущем задании утверждению некоторые птицы, такие как страус или скворец с перебитым крылом, летать не могут. Однако идея создания дедуктивной системы, в которой будут явно перечислены все исключения из утверждения "Все птицы умеют летать", не кажется нам разумной. Как же тогда человек принимает решение, может ли определенная птица летать?

48. Объясните, как семантическое значение выражения "Я представил мужа тучной леди" может зависеть от контекста.

49. Объясните, как проблема путешествия из одного города в другой может быть описана в терминах порождающей системы. Что здесь является состояниями, а что — порождениями?

50. Предположим, что вы должны в произвольном порядке (но не одновременно) выполнить три задания — А, Б и В. Опишите эту проблему в терминах порождающей системы и начертите для нее граф состояний.

51. Как изменится граф состояний из предыдущего задания, если задание В должно быть выполнено раньше, чем задание А?

52. **а.** Пусть запись (i, j), где i и j — целые положительные числа, используется для обозначения правила "если число, находящееся на i-й позиции

списка, больше, чем находящееся на *j*-й, то следует поменять их местами". Какая из следующих двух последовательностей действий лучше справится с задачей сортировки списка из трех чисел?

(1, 3) (3, 2)
(1, 2) (2, 3) (1, 2)

б. Следует отметить, что если представлять последовательности чередований таким образом, то последовательности могут быть разбиты на подпоследовательности, которые затем могут повторно соединяться для формирования новых последовательностей. Используя такой подход, разработайте генетический алгоритм для создания программы, сортирующей список из десяти чисел.

53. Предположим, что каждый член группы роботов должен быть оснащен парой датчиков. Каждый датчик может обнаружить объект непосредственно перед ним на расстоянии до двух метров. Каждый робот имеет форму круглого мусорного ведра и может двигаться в любом направлении. Разработайте последовательность экспериментов, проводимых с целью определения, где должны располагаться датчики робота, чтобы он мог успешно толкать баскетбольный мяч по прямой линии. В какой степени предлагаемую вами последовательность экспериментов можно сравнить с эволюционной системой?

54. Как вы склонны принимать решения: в реактивном или в строго планируемом режиме? Будет ли ваш ответ зависеть от того, решаете ли вы, что съесть на обед или как успешнее сделать карьеру?

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также понять, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. Насколько исследователи в области ядерной энергетики, генной инженерии и искусственного интеллекта ответственны за использование результатов их работы? Ответственен ли ученый за знания, полученные в результате его исследований? Что если полученные знания привели к неожиданным последствиям?

2. Как бы вы определили различие между интеллектом и кажущимся интеллектом? Вы уверены, что это разные вещи?
3. Предположим, что компьютеризированная медицинская экспертная система приобрела в медицинском сообществе хорошую репутацию благодаря высокому качеству даваемых ею советов. До какой степени врач может позволить этой системе изменять его решение относительно методов лечения его пациентов? Если врач выберет метод лечения, противоположный рекомендациям системы, а впоследствии окажется, что права была система, виновен ли человек в злоупотреблении доверием больного? И вообще, если экспертная система становится широко известной в своей области, то до какой степени она будет скорее повышать, чем снижать возможности человеческих экспертов, когда те выносят собственное суждение?
4. Многие согласятся с тем, что действия компьютера — это просто последствия того, как он был запрограммирован, а следовательно, компьютер не имеет свободы воли. А это означает, что компьютер не несет ответственности за совершаемые им действия. Является ли компьютером человеческий мозг? Заложена ли в него программа уже при рождении человека? Программируются ли люди окружающей их средой? Ответственны ли люди за свои действия?
5. Имеются ли области, в которых наука не должна работать, даже если и есть такая возможность? Например, если станет возможным создание машины, ощущения и мыслительные возможности которой сравнимы с человеческими, будет ли уместным создание такой машины? Какие предметы для спора вызовет возможность существования такой машины? Какие споры кипят уже сейчас в отношении достижений в других областях науки?
6. Истории известны многочисленные примеры влияния на работу ученых и художников политических, религиозных и других общественных институтов того времени. Какими способами сейчас оказывается подобное влияние на работу ученых и на работу специалистов по программированию в частности?
7. Сегодня многие организации и общественные структуры берут на себя хотя бы часть ответственности за переобучение тех, чье поле деятельности было сокращено в связи с развитием технологии. Что должно и что может сделать общество, если технологии все больше и больше будут выходить за пределы наших возможностей?
8. Предположим, что вы получили для оплаты счет на 0,00 долларов. Что вы будете делать? Предположим, что вы ничего не сделали и через 30 дней

получили второе извещение о поступлении к оплате счета на 0,00 долларов. Что вы теперь будете делать? Предположим, что вы опять ничего не сделали и через следующие 30 дней получили новое извещение о необходимости оплатить счет на сумму 0,00 долларов с примечанием, что если вы немедленно его не оплатите, то будут приняты меры. Кто ответствен за подобное?

9. Бывают ли случаи, когда вы ассоциируете со своим персональным компьютером некое подобие личности? Бывают ли моменты, когда эта личность кажется вам мстительной или упрямой? Вы когда-нибудь злитесь на свой компьютер? В чем разница между злостью на компьютер и злостью на результаты, полученные на этом компьютере? Ваш компьютер когда-нибудь злился на вас? Есть ли у вас похожие отношения с другими объектами, такими как автомобили, телевизоры или шариковые авторучки?
10. Исходя из ваших ответов на вопрос 9, в какой степени люди готовы ассоциировать поведение объекта с наличием у него интеллекта и осведомленности? До какой степени людям позволено создавать такие ассоциации? Возможно ли, чтобы разумное существо раскрыло свой разум каким-либо другим способом, кроме своего поведения?
11. Многие считают, что способность машины пройти тест Тьюринга не означает, что она интеллектуальна. Одним из аргументов является то, что интеллектуальное поведение само по себе не подразумевает интеллект. Тем не менее теория эволюции основана на выживании наиболее приспособленных, что в действительности является тестом на основе поведения. Означают ли положения теории эволюции, что разумное поведение является предшественником интеллекта? Означает ли для машин способность пройти тест Тьюринга то, что они находятся на пути к тому, чтобы стать интеллектуальными?
12. Возможности медицины сейчас достигли такого уровня, что многие части человеческого тела теперь можно заменить искусственными элементами или частями тела от доноров. Вполне возможно, что в этот перечень когда-нибудь смогут попасть и части человеческого мозга. Какие этические проблемы возникнут при появлении такой возможности? Если нейроны пациента по одному будут заменяться искусственными нейронами, останется ли этот пациент той же личностью? Сможет ли пациент когда-нибудь заметить различия? Будет ли такой пациент оставаться человеком?
13. В автомобиле GPS-устройства уведомляют водителя о предстоящих поворотах и других необходимых действиях спокойным, дружеским голосом. Если водитель совершает ошибку, устройство автоматически вносит

необходимые коррективы и дает указания, как вернуться на маршрут без каких-либо лишних эмоций. Считаете ли вы, что GPS-устройства позволяют снизить уровень стресса у водителя при въезде в новое для него место назначения? Каким образом GPS-устройства могут способствовать усилению стресса водителя?

14. Приложения для смартфонов теперь могут обеспечивать базовый голосовой перевод на другие языки. При каких обстоятельствах вы будете чувствовать себя комфортно, используя эту функцию? Полагаетесь ли вы на то, что подобный перевод будет правильно передавать исходный смысл сообщения? Сталкивались ли вы при использовании подобных приложений с какими-либо проблемами?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Banzhaf W.P., Nordin R., Deller E., Francone F.D. *Genetic Programming: An Introduction.* — San Francisco, CA: Morgan Kaufmann, 1998.
2. Lu J., Wu J. *Multi-Agent Robotic Systems.* — Boca Raton, FL: CRC Press, 2001.
3. Luger G.F. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 6th ed. — Boston, MA: Addison-Wesley, 2008. (Имеется русский перевод 4-го издания этой книги: Люгер Дж.Ф. *Искусственный интеллект: стратегии и методы решения сложных проблем*, 4-е изд. — М.: Издательский дом "Вильямс", 2003.)
4. Mitchell M. *An Introduction to Genetic Algorithms.* — Cambridge, MA: MIT Press, 1998.
5. Negnevitsky M. *Artificial Intelligence: A Guide to Intelligent Systems*, 2nd ed. — Boston, MA: Addison-Wesley, 2005.
6. Nilsson N. *Artificial Intelligence: A New Synthesis.* — San Francisco, CA: Morgan Kaufmann, 1998.
7. Nolfi S., Floreano D. *Evolutionary Robotics.* — Cambridge, MA: MIT Press, 2000.
8. Rumelhart D.E., McClelland J.L. *Parallel Distributed Processing.* — Cambridge, MA: MIT Press, 1986.
9. Russell S., Norvig P. *Artificial Intelligence: A Modern Approach*, 3rd ed. — Upper Saddle River, NJ: Prentice-Hall, 2009. (Имеется русский перевод 2-го издания этой книги: Рассел С., Норвиг П. *Искусственный интеллект: современный подход*, 2-е изд. — М.: Издательский дом "Вильямс", 2005.)
10. Shapiro L.G., Stockman G.C. *Computer Vision.* — Englewood Cliffs, NJ: Prentice-Hall, 2001.

11. Shieber S. *The Turing Test.* — Cambridge, MA: MIT Press, 2004.
12. Weizenbaum J. *Computer Power and Human Reason.* — New York: W. H. Freeman, 1979.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Джоши П. *Искусственный интеллект с примерами на Python.* — СПб.: ООО "Диалектика", 2019.
2. Форсайт Д.А., Понс Дж. *Компьютерное зрение. Современный подход.* — М.: Издательский дом "Вильямс", 2004.
3. Братко И. *Алгоритмы искусственного интеллекта на языке PROLOG*, 3-е изд. — М.: Издательский дом "Вильямс", 2004.
4. Джексон П. *Введение в экспертные системы*, 3-е изд. — М.: Издательский дом "Вильямс", 2001.

Глава 12

В этой главе мы рассмотрим теоретические основы компьютерных наук. В некотором смысле именно материал данной главы придает всей этой области исследований и экспериментов статус настоящей науки. Хотя в целом эта совокупность знаний несколько абстрактна по своей природе, она, тем не менее, находит много практических применений. В частности, мы рассмотрим влияние положений теории вычислений на мощь языков программирования, а также увидим, как теоретические исследования привели к созданию системы шифрования с открытым ключом, которая сейчас широко используется при обмене данными через Интернет.

Теория вычислений

12.1. ФУНКЦИИ И ИХ ВЫЧИСЛЕНИЕ

12.2. МАШИНЫ ТЬЮРИНГА
- Понятие машины Тьюринга
- Тезис Черча–Тьюринга

12.3. УНИВЕРСАЛЬНЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ
- Простейший язык программирования
- Программирование на простейшем языке
- Универсальность простейшего языка

12.4. НЕВЫЧИСЛИМЫЕ ФУНКЦИИ
- Проблема остановки
- Неразрешимость проблемы остановки

12.5. СЛОЖНОСТЬ ЗАДАЧ
- Измерение сложности задачи
- Задачи полиномиального и неполиномиального типов
- NP-задачи

*12.6. КРИПТОГРАФИЯ С ОТКРЫТЫМ КЛЮЧОМ
- Модульная арифметика
- RSA-криптография с открытым ключом

** Звездочкой отмечены разделы, рекомендованные для факультативного изучения.*

В этой главе будут рассмотрены некоторые вопросы относительно того, что компьютеры могут делать и чего не могут. Вы узнаете, как простые машины, известные как машины Тьюринга, могут использоваться для определения границы между проблемами, которые решаются машинами, и проблемами, решить которые они неспособны. В частности, будет рассмотрена конкретная проблема, известная как проблема остановки, решение которой выходит за рамки возможностей алгоритмических систем, а значит, и за пределы возможностей как современных, так и будущих компьютеров. Более того, будет показано, что даже среди потенциально решаемых компьютерами задач есть такие, решить которые настолько сложно, что они являются фактически неразрешимыми с чисто практической точки зрения. В завершение главы будет показано, как знания из области теории сложности вычислений можно использовать для построения эффективной системы шифрования с открытым ключом.

12.1. Функции и их вычисление

Основное назначение этой главы — исследование возможностей компьютеров. Необходимо понять, что машины могут делать и чего не могут, а также установить, какие функциональные возможности необходимы машинам, чтобы они могли полностью реализовать свой потенциал. В предыдущих главах было приведено много примеров функций на языке Python, однако здесь мы начнем обсуждение с более общей концепции вычисления математических функций.

Функция в ее математическом смысле представляет собой соответствие между множеством возможных входных значений и множеством выходных значений, так что каждое возможное входное значение связывается с некоторым выходным значением. В качестве примера можно привести функцию, которая преобразует расстояние, выраженное в футах, в метры. Для каждого расстояния, выраженного в ярдах, эта функция возвращает то значение, которое получится, если то же расстояние измерить в метрах. В качестве другого примера можно привести функцию `sort()`, которая ставит в соответствие любому входному списку числовых значений соответствующий выходной список, содержащий те же самые числовые значения, что и во входном списке, но расположенные в порядке возрастания их значений. Еще одним примером является функция `addition()`, входные данные которой представлены парами числовых значений, а выходные значения представляют собой сумму каждой пары входных значений.

Процесс определения того выходного значения, которое функция назначает заданному входному значению, называется *вычислением* функции. Способность вычислять функции очень важна, поскольку именно за счет вычисления функций у нас появляется возможность решать различные задачи. Чтобы

решить задачу сложения двух чисел, достаточно вычислить соответствующее значение функции `addition()`; чтобы отсортировать список, необходимо вычислить функцию `sort()`, передав ей этот список как входное значение. В свою очередь, фундаментальная задача компьютерных наук — найти методы вычисления тех функций, которые лежат в основе проблем, для которых требуется найти решение.

Теория рекурсивных функций

Ничто так не дразнит человеческую природу, как заявление о том, что что-то сделать невозможно. Как только одни исследователи начали выявлять проблемы, которые нельзя решить в том смысле, что у них нет алгоритмических решений, другие исследователи начали пристально изучать эти проблемы, пытаясь понять, в чем состоит их сложность. Сегодня эта область исследований является основной частью предмета, известного как "теория рекурсивных функций", и об этих сверхсложных проблемах многое уже известно. Действительно, точно так же, как математики разработали системы счисления, раскрывающие "количественные" уровни за пределами бесконечности, теоретики рекурсивных функций обнаружили несколько уровней сложности в задачах, которые находятся далеко за пределами возможностей алгоритмов.

Например, рассмотрим систему, в которой выходные значения функции определены заранее и записаны в таблицу вместе с соответствующими им входными значениями. Всякий раз, когда требуется найти значение функции, мы просто находим в таблице входную величину и определяем соответствующее ей выходное значение. Подобные системы удобны, но их возможности ограничены, поскольку существует множество функций, которые не могут быть представлены в табличном виде полностью. Пример подобной функции приведен на рис. 12.1; здесь предпринята попытка отобразить функцию, преобразующую измерения в футах в эквивалентные измерения в метрах. Поскольку список возможных пар входных и выходных значений неограничен, эта таблица обречена быть неполной.

Более мощный подход к вычислению функций состоит в том, чтобы следовать указаниям, представленным в виде алгебраической формулы, а не пытаться отобразить все возможные комбинации входных и выходных значений в таблице. Например, для описания, как рассчитать суммарный доход от инвестиций на сумму P при процентной ставке r за n лет, можно использовать следующую алгебраическую формулу:

$$V = P(1 + r)^n$$

Футы (вход)	Метры (выход)
1	0,3048
2	0,6096
3	0,9144
4	1,2192
5	1,5240
.	.
.	.
.	.

Рис. 12.1. Попытка отобразить функцию, которая преобразует измерения в футах в метры

Однако выразительная мощность алгебраических формул также имеет свой предел. Существуют функции, в которых взаимосвязь между входными и выходными значениями настолько сложна, что не может быть описана посредством алгебраических манипуляций входными значениями. Типичным примером являются тригонометрические функции, такие как синус и косинус. Чтобы вычислить синус угла в 38°, следует нарисовать соответствующий треугольник, измерить его стороны и вычислить требуемое отношение; это процесс, не имеющий выражения в терминах алгебраических манипуляций числом 38. Тем не менее любой карманный калькулятор успешно решает проблему вычисления синуса угла в 38°. Фактически же в этом случае изощренные математические методы применяются для получения лишь очень хорошего *приближения* к значению синуса угла в 38°, которое и выдается в качестве ответа.

Как видите, по мере рассмотрения функций со все возрастающей сложностью приходится применять все более и более сложные алгоритмы их вычисления. Вопрос заключается в том, всегда ли будет возможно найти систему, способную вычислять функции независимо от их сложности. Ответ — *нет*. Поразительным результатом математических исследований является обнаружение существования функций настолько сложных, что для них не существует четкого пошагового процесса определения выходных значений, исходя из заданных входных значений. Как следствие вычисление этих функций оказывается за пределами возможностей любой алгоритмической системы. Эти функции называются **невычислимыми**, в отличие от тех функций, выходные значения которых могут быть алгоритмически определены на основании их входных значений и которые по этой причине называют **вычислимыми**. Даже если мы ограничим наше обсуждение функциями, которые выдают на выходе только два значения, `1` или `0` (*да* или *нет*), вся совокупность возможных в этом случае задач также разделится на два лагеря: **решаемые** функции, для которых можно построить правильный алгоритм для любых возможных входных

значений, и **неразрешимые** функции, для которых не может быть построен алгоритм, всегда возвращающий правильный ответ.

Различие между вычислимыми и невычислимыми функциями очень важно в компьютерных науках. Поскольку машины могут выполнять только те задания, которые можно описать алгоритмами, изучение вычислимых функций — это изучение пределов возможностей машин. Если мы сможем определить тот набор функциональных средств, который позволит машине вычислять все существующее множество вычислимых функций, а затем построим машины, обладающие этими функциональными средствами, то получим уверенность, что созданные нами машины настолько мощные, насколько это вообще возможно. Аналогичным образом, если обнаружится, что решение проблемы требует вычисления невычислимой функции, можно с уверенностью заключить, что решение данной проблемы лежит за пределами возможностей машин.



Основные положения для запоминания

- **Решаемой является проблема, при которой алгоритм может быть сконструирован таким образом, чтобы давать ответ "да" или "нет" для всех возможных значений входных данных. (Пример подобной проблемы — "Является ли указанное число четным?").**
- **Неразрешимой является проблема, для которой не может быть построен алгоритм, всегда приводящий к получению правильного ответа "да" или "нет".**

12.1. Вопросы и упражнения

1. Приведите несколько примеров функций, которые могут быть представлены в табличной форме полностью.
2. Приведите несколько примеров функций, выходное значение которых может быть описано как алгебраическое преобразование их входного значения.
3. Приведите пример функции, которая не может быть описана в терминах алгебраических формул. Является ли эта функция вычислимой?
4. Древнегреческие математики для рисования фигур использовали только линейку и циркуль. Они разработали методы нахождения средней точки на отрезке прямой линии, построения прямого угла и рисования равностороннего треугольника. Тем не менее существовали и такие "вычисления", которые их "вычислительная система" выполнить не могла. Приведите соответствующие примеры.

12.2. Машины Тьюринга

В попытках понять возможности и ограничения машин многие исследователи предлагали и изучали различные вычислительные устройства. Одним из таких устройств является *машина Тьюринга*, которая была предложена Аланом М. Тьюрингом в 1936 году и до сих пор используется в качестве инструмента для изучения возможностей алгоритмических процессов.

Понятие машины Тьюринга

Машина Тьюринга состоит из блока управления, который считывает и записывает символы на ленте с помощью головки считывания/записи, как схематически показано на рис. 12.2. Лента неограниченно простирается в обоих направлениях и поделена на ячейки, каждая из которых может содержать один произвольный символ из конечного набора. Этот набор символов называется машинным алфавитом.

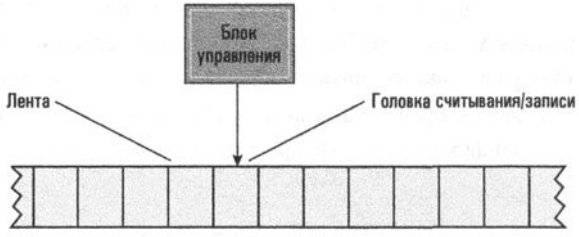


Рис. 12.2. Компоненты машины Тьюринга

В любой момент вычислений машина Тьюринга должна находиться в одном из возможных положений (число которых конечно), называемых *состояниями*. Вычисления машины Тьюринга начинаются в специальном состоянии, называемом стартовым, и прекращаются, когда машина переходит в другое специальное состояние, называемое состоянием останова.

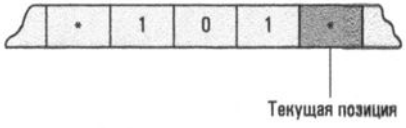
Вычисления машины Тьюринга состоят из последовательности этапов, выполняемых ее блоком управления. Каждый этап включает считывание символа в текущей ячейке ленты (находящегося под считывающей головкой), запись символа в ту же ячейку, возможное перемещение головки в соседнюю ячейку слева или справа с последующим изменением состояния. Конкретное выполняемое действие зависит от программы, сообщающей блоку управления, что делать, исходя из состояния машины и содержания текущей ячейки ленты.

Давайте рассмотрим какой-либо конкретный пример машины Тьюринга. Для этого представим ее машинную ленту как длинную полоску, поделенную на ячейки, в которые можно записывать символы машинного алфавита. Текущее

Происхождение машин Тьюринга

Алан Тьюринг разработал модель своей машины еще в 1930-х годах — задолго до того, как развитие технологии позволило изготовить те вычислительные машины, с которыми мы имеем дело сегодня. Фактически исходной концепцией Тьюринга явилось представление о выполнении человеком вычислений с помощью карандаша и бумаги. Целью работы Тьюринга являлась разработка модели, которая позволила бы изучать предельные возможности "вычислительных процессов". Машина Тьюринга появилась вскоре после публикации в 1931 году знаменитой статьи Гёделя (Gödel), демонстрирующей ограниченность вычислительных систем, после чего многие ученые-исследователи направили свои усилия на понимание этих ограничений. В том же 1936 году, когда Тьюринг представил свою модель, Эмиль Пост (Emil Post) создал другую модель (в настоящее время известную как порождающая система Поста), имеющую те же возможности, что и машина Тьюринга. Обе модели до сих пор используются в качестве инструментов для изучения возможностей современной вычислительной техники, что служит доказательством проницательности упомянутых исследователей.

положение головки считывания/записи машины будем отмечать специальным указателем над текущей ячейкой. В нашем примере машинный алфавит будет состоять из символов `0`, `1` и `*`. Лента нашей машины может выглядеть следующим образом.



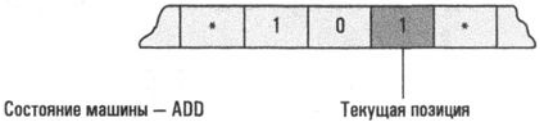
Интерпретируя строку символов на ленте машины как двоичное представление чисел, разделенных звездочками, можно прийти к заключению, что данная часть ленты содержит число `5`. Назначение нашей машины Тьюринга состоит в увеличении числа на ленте на единицу. Точнее говоря, предполагается, что в стартовом положении головка находится над звездочкой, расположенной справа от строки из нулей и единиц, а задача заключается в том, чтобы изменить комбинацию битов, расположенную левее, так, чтобы она представляла собой следующее по величине целое число.

Состояниями нашей машины являются `START` (*Старт*), `ADD` (*Прибавить*), `CARRY` (*Перенести*), `NO CARRY` (*Не переносить*), `OVERFLOW` (*Переполнение*), `RETURN` (*Возврат*) и `HALT` (*Останов*). Действия, относящиеся к каждому из этих состояний, и содержание текущей ячейки описаны в таблице, представленной на рис. 12.3. Будем предполагать, что в начале работы машина всегда находится в состоянии `START`.

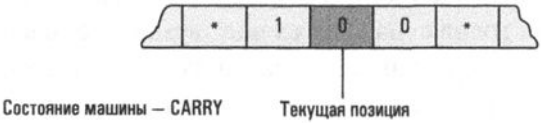
Текущее состояние	Содержание текущей ячейки	Записываемое значение	Направление перемещения	Следующее состояние
START	*	*	Влево	ADD
ADD	0	1	Вправо	RETURN
ADD	1	0	Влево	CARRY
ADD	*	*	Вправо	HALT
CARRY	0	1	Вправо	RETURN
CARRY	1	0	Влево	CARRY
CARRY	*	1	Влево	OVERFLOW
OVERFLOW	(Игнорируется)	*	Вправо	RETURN
RETURN	0	0	Вправо	RETURN
RETURN	1	1	Вправо	RETURN
RETURN	*	*	Нет перемещения	HALT

Рис. 12.3. Состояния машины Тьюринга, предназначенной для увеличения числа на единицу

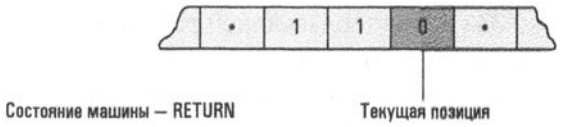
Запустим машину, установив на нее изображенную выше ленту, содержащую число `5`. Обратите внимание: когда в состоянии `START` текущая ячейка содержит звездочку (как в нашем примере), инструкция в таблице требует записать в эту ячейку символ `*`, переместить головку считывания/записи на одну ячейку влево, а затем перевести машину в состояние `ADD`. В результате возникает следующая ситуация.



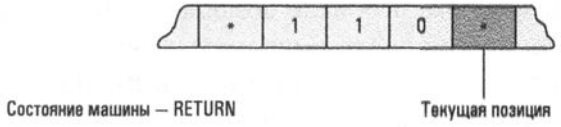
Для продолжения работы вновь заглянем в таблицу, чтобы узнать, что нужно делать, когда машина находится в состоянии `ADD`, а в ячейке содержится символ `1`. Таблица указывает, что необходимо заменить символ `1` символом `0`, переместить головку считывания/записи на одну ячейку влево, а затем перевести машину в состояние `CARRY`. Новая ситуация выглядит следующим образом.



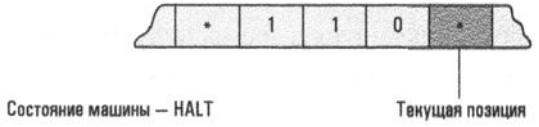
Вновь обращаемся к таблице, чтобы узнать, что нужно делать, когда машина находится в состоянии `CARRY`, а в текущей ячейке содержится символ `0`. Оказывается, следует заменить символ `0` символом `1`, переместить головку считывания/записи на одну ячейку вправо, а затем перевести машину в состояние `RETURN`. После выполнения указанной последовательности действий получаем следующее состояние машины.



Далее, согласно таблице следует заменить символ 0 в текущей ячейке тем же символом 0, после чего переместить головку считывания/записи на одну ячейку вправо, оставив машину в прежнем состоянии RETURN. В результате будет получена ситуация, показанная ниже.



В этом случае согласно таблице требуется перезаписать символ звездочки в текущей ячейке и перевести машину в состояние HALT. В результате машина завершает работу в следующем положении (теперь совокупность символов на ленте представляет число 6, что и требовалось получить).



Тезис Черча–Тьюринга

Машина Тьюринга в предыдущем примере может использоваться для вычисления функции, известной как функция следования, которая присваивает каждому неотрицательному целочисленному входному значению n выходное значение $n + 1$. Для этого достаточно поместить на ленту входное значение, представленное в двоичной форме, запустить машину в работу и, дождавшись ее останова, считать с ленты полученное выходное значение. Функция, выходные значения которой могут быть вычислены с помощью некоторой машины Тьюринга, называется **вычислимой по Тьюрингу**.

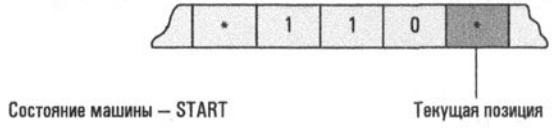
Гипотеза Тьюринга заключалась в тождественности понятий функции, вычислимой по Тьюрингу, и просто вычислимой функции. Иными словами, он предположил, что вычислительная мощность машин Тьюринга не уступает мощности любых алгоритмических систем. Это эквивалентно тому, что (в отличие от подходов, использующих таблицы или алгебраические формулы) концепция машины Тьюринга предоставляет среду, в которой могут быть описаны все вычислимые функции. В наше время эта гипотеза обычно упоминается как тезис **Черча–Тьюринга**, в честь Алана Тьюринга и Алонзо Черча (Alonzo

Church). С момента выхода основополагающей работы Тьюринга было собрано множество фактов в поддержку этого тезиса, и сейчас тезис Черча–Тьюринга можно считать принятым широкими научными кругами. Поэтому мы можем утверждать, что вычислимые функции и функции, вычислимые по Тьюрингу, следует рассматривать как эквивалентные понятия.

Важность этой гипотезы состоит в том, что она предоставляет конкретный подход к оценке возможностей и ограничений, свойственных вычислительной технике. Говоря точнее, она устанавливает набор вычислимых по Тьюрингу функций в качестве пробного набора, с помощью которого сравнивается вычислительная мощность различных вычислительных систем. Если вычислительная система способна вычислить все вычислимые по Тьюрингу функции, то она считается настолько мощной, насколько вообще может быть мощной любая вычислительная система.

12.2. Вопросы и упражнения

1. Запустите в работу описанную выше машину Тьюринга (см. рис. 12.3) при условии, что ее лента находится в следующем исходном состоянии.



2. Опишите машину Тьюринга, заменяющую произвольную строку из нулей и единиц единственным нулем.
3. Опишите машину Тьюринга, уменьшающую любое значение, которое больше нуля, значением `1` и не изменяющую значение, если оно равно нулю.
4. Приведите примеры типичных ситуаций из повседневной жизни, в которых требуется выполнение вычислений. Какую аналогию можно провести между этими ситуациями и машиной Тьюринга?
5. Опишите машину Тьюринга, которая обязательно завершит свою работу для одних входных значений, но никогда не остановится при других входных значениях.

12.3. Универсальные языки программирования

В главе 6 рассматривалось множество функций, которые можно найти в языках программирования высокого уровня. В этом разделе мы попробуем применить только что полученные знания о вычислимости функций для определения, какие из них действительно необходимы. Забегая вперед, можно сразу сказать, что большинство функций, существующих в современных языках высокого уровня, по сути, просто повышают удобство использования этих языков и не вносят никакого дополнительного вклада в их фундаментальную мощность.

Наш подход заключается в том, чтобы описать простой императивный язык программирования, который будет достаточно богат, чтобы позволить создавать с его помощью программы для вычисления всех функций, вычислимых по Тьюрингу (и следовательно, всех вычислимых функций вообще). Таким образом, если будущий программист обнаружит, что проблема не может быть решена с помощью этого языка, причина будет вовсе не в его недостаточной функциональности. В действительности причина будет заключаться в том, что алгоритма решения этой проблемы просто не существует. Язык программирования с такими свойствами называют **универсальным языком программирования**.

Вас может удивить тот факт, что универсальный язык программирования вовсе не должен быть сложным. В действительности язык, о котором здесь пойдет речь, довольно прост, и по этой причине мы будем называть его *простейшим*. Однако этот язык полностью отвечает минимальному набору требований, предъявляемых к любому универсальному языку программирования.

Простейший язык программирования

Давайте начнем знакомство с нашим простейшим языком программирования с рассмотрения переменных, которые можно найти в других языках программирования. Переменные позволяют программистам мыслить в терминах структур данных и типов данных (таких, как массивы числовых величин и строки буквенных символов), несмотря на то что сама машина не ассоциирует такую трактовку с теми комбинациями двоичных разрядов, которые представляют эти объекты. Но прежде чем представить машине для выполнения команду высокого уровня, манипулирующую сложными типами данных и структурами, необходимо будет преобразовать (оттранслировать) ее в набор машинных команд, которые и обеспечат выполнение всех операций с битовыми комбинациями, необходимых для реализации запрашиваемых действий.

Для удобства мы можем интерпретировать эти битовые комбинации как числовые значения, представленные в двоичной форме. Следовательно, все

вычисления, выполняемые компьютером, могут быть выражены в виде числовых вычислений с использованием неотрицательных целых чисел, — это все, что в действительности сможет увидеть наблюдатель. Более того, языки программирования могут быть упрощены и за счет выдвижения требования, согласно которому программисты должны выражать алгоритмы исключительно в терминах комбинаций двоичных разрядов (хотя для программистов это будет тяжелым бременем).

Поскольку наша цель при разработке простейшего языка состоит в том, чтобы разработать максимально простой язык, в нем будет использован именно такой подход. Будем считать, что все переменные в нашем простейшем языке представляют собой битовые комбинации, которые для удобства интерпретируются как неотрицательные целые числа в двоичной нотации. Следовательно, переменная, которой присваивается битовая комбинация `10`, с нашей точки зрения, будет содержать значение "два", тогда как переменная, которой присваивается битовая комбинация `101`, будет содержать значение "пять".

Благодаря этому соглашению все переменные в программе на простейшем языке будут иметь один и тот же тип, "комбинация двоичных разрядов", поэтому в нем не требуются операции для преобразования данных различных типов, как и декларативные операторы описания имен переменных и связанных с ними свойств. При написании программ на простейшем языке, как и в языке Python, программист может просто начать использовать новое имя переменной, когда это необходимо, подразумевая, что оно ссылается на комбинацию битов, интерпретируемую как неотрицательное целое число в двоичном коде.

Конечно, транслятор нашего простейшего языка должен уметь отличать имена переменных от других выражений. Для этого необходимо разработать такой синтаксис языка, чтобы роль каждого терма была понятна из его контекста. В соответствии с этим установим, что имена переменных должны начинаться только с латинских букв, за которыми могут следовать любые комбинации из латинских букв и цифр (от 0 до 9). Поэтому строки `XYZ`, `B747`, `abcdefghi` и `X5Y` могут использоваться в качестве имен переменных, тогда как строки `2G5`, `%o` и `x.y` — не могут.

Теперь обратимся к процедурным операторам нашего простейшего языка. В нем будут использоваться три оператора присваивания и одна управляющая структура, представляющая цикл. В соответствии с соглашением следования синтаксису языка Python установим требования записывать только один оператор в каждой строке и использовать отступы для определения тела циклической структуры.

Каждый из трех операторов присваивания тем или иным образом модифицирует значение указанной в этом операторе переменной. Первый оператор

позволяет связать с именем переменной строку двоичных нулей. Он имеет следующий синтаксис:

```
clear <имя>
```

Здесь параметр <имя> может быть любым допустимым именем переменной.

Оставшиеся два оператора присваивания взаимно противоположны по выполняемым действиям и имеют следующий синтаксис:

```
incr <имя>
```

и

```
decr <имя>
```

И в этом случае параметр <имя> также представляет собой любое допустимое имя переменной. Первый из представленных операторов выполняет увеличение связанного с указанной переменной значения на единицу. Например, рассмотрим выполнение приведенного ниже оператора, полагая, что исходным значением переменной Y является комбинация битов 101.

```
incr Y
```

Поскольку к исходному значению переменной Y прибавляется единица, после выполнения этого оператора с переменной Y будет связана комбинация битов 110, представляющая число 6.

В противоположность этому оператор decr используется для уменьшения значения, связанного с указанной переменной, на единицу. Исключением является случай, когда связанное с переменной значение уже равно нулю. В этом случае данный оператор не изменяет значения переменной. Рассмотрим выполнение следующего оператора.

```
decr Y
```

Если исходным значением переменной Y будет комбинация битов 101, то после выполнения данного оператора с переменной Y будет связана комбинация битов 100, представляющая число 4. Однако, если до выполнения этого оператора с переменной Y было связано нулевое значение, оно останется неизменным и после его выполнения.

Наш простейший язык содержит всего одну управляющую структуру, представленную оператором while. Приведенная ниже последовательность операторов

```
while <имя> not 0:
   .
   .
   .
```

(здесь параметр <имя> представляет собой любое допустимое имя переменной) вызовет повторение любой последовательности операторов, помещенной после оператора while, пока значение переменной <имя> не станет равным нулю. Точнее говоря, когда в процессе выполнения программы встречается структура while, то прежде всего проверяется, равно ли нулю текущее значение переменной <имя>. Если это так, то вся структура пропускается и начинается выполнение операторов, следующих за телом цикла, выделенного отступами. Если же значение данной переменной отлично от нуля, то выполняется последовательность операторов в теле цикла, выделенного отступами, после чего управление возвращается оператору while, где вновь выполняется указанное сравнение. Отметим, что ответственность за предупреждение зацикливания программы возлагается исключительно на программиста, который в теле цикла должен явно описать действия, необходимые для изменения значения его управляющей переменной таким образом, который позволит избежать его бесконечного выполнения. Например, рассмотрим последовательность операторов.

```
incr X
while X not 0:
  incr Z
```

Этот цикл будет выполняться бесконечно, поскольку значение переменной X никогда не будет равно нулю. Этого нельзя сказать о последовательности операторов.

```
clear Z
while X not 0:
  incr Z
  decr X
```

Выполнение данного цикла обязательно завершится, причем в переменную Z будет помещено исходное значение переменной X.

Отметим, что операторы while могут появляться и среди операторов, образующих тело цикла другого оператора while. В подобных случаях для определения, какие именно операторы принадлежат к телу вложенных операторов while, используется несколько уровней отступов.

Как завершающий пример рассмотрим последовательность операторов, представленную на рис. 12.4. Она предназначена для вычисления произведения значений переменных X и Y, которое присваивается переменной Z. Однако эта программа имеет побочный эффект, заключающийся в уничтожении любого исходного значения переменной X, отличного от нуля. (Исходное значение переменной Y восстанавливается в структуре while, управляемой переменной W.)

```
clear Z
while X not 0:
  clear W
  while Y not 0:
    incr Z
    incr W
    decr Y
  while W not 0:
    incr Y
    decr W
  decr X
```

Рис. 12.4. Программа перемножения значений переменных X и Y, написанная на простейшем языке

Программирование на простейшем языке

Не забывайте, что описание простейшего языка программирования здесь дается лишь для обеспечения возможности исследовать, что *возможно*, а вовсе не того, что *удобно*. Поэтому использование этого простейшего языка для каких-либо прикладных целей, вероятнее всего, окажется весьма затруднительным. С другой стороны, вы скоро увидите, что этот простой язык успешно играет роль универсального языка программирования и при этом абсолютно лишен каких-либо излишеств. А сейчас пора просто продемонстрировать, как можно использовать этот простейший язык программирования для выражения некоторых элементарных операций.

Прежде всего отметим, что посредством комбинирования операторов присвоения с заданной переменной может быть связано практически любое значение (любое неотрицательное целое). Например, приведенная ниже последовательность операторов присваивает битовую комбинацию `11` (двоичное представление числа 3) в качестве значения переменной X, что достигается посредством обнуления ее предыдущего значения и последующего троекратного применения оператора увеличения текущего значения на единицу:

```
clear X
incr X
incr X
incr X
```

Еще одним типичным действием в программах является перемещение данных из одного места в другое. В терминах простейшего языка это означает возможность присвоения текущего значения одной переменной другой переменной. Эта операция может быть выполнена посредством предварительного обнуления переменной назначения и последующего увеличения ее значения

соответствующее число раз. Фактически эти действия уже были реализованы в рассмотренном выше примере:

```
clear Z
while X not 0:
  incr Z
  decr X
```

В результате выполнения данной последовательности операторов переменной `Z` присваивается значение переменной `X`. Однако выполнение этой последовательности операторов имеет побочный эффект, заключающийся в уничтожении исходного значения самой переменной `X`. Во избежание этого введем промежуточную переменную, в которую предварительно переместим пересылаемое значение из его исходного положения. Затем используем эту промежуточную переменную как источник данных для восстановления исходного значения с одновременным присвоением того же значения целевой переменной. С помощью данного метода перемещение значения переменной `Today` (*сегодня*) в переменную `Yesterday` (*вчера*) может быть выполнено с помощью последовательности операторов, представленной на рис. 12.5.

```
clear Aux
clear Tomorrow
while Today not 0:
  incr Aux
  decr Today
while Aux not 0:
  incr Today
  incr Tomorrow
  decr Aux
```

Рис. 12.5. Реализация на простейшем языке команды `copy Today to Tomorrow`

Для сокращенного представления показанной на рис. 12.5 последовательности операторов можно принять следующий синтаксис:

```
copy <имя1> to <имя2>
```

Здесь параметры `<имя1>` и `<имя2>` представляют имена соответствующих переменных. Таким образом, хотя простейший язык сам по себе и не имеет явной команды присвоения, мы часто будем писать программы, полагая, что он допускает подобные действия. При этом следует помнить, что для преобразования подобной программы в текст на действительном простейшем языке следует повсеместно заменить оператор `copy` эквивалентной структурой `while`, обратив внимание на то, чтобы используемая в ней промежуточная переменная больше нигде в программе не употреблялась.

Универсальность простейшего языка

В качестве примера значения тезиса Черча–Тьюринга применим его для подтверждения сделанного выше заявления о том, что наш простейший язык является универсальным языком программирования. Прежде всего отметим, что любая программа, написанная на этом языке, может рассматриваться как вычисление значений функции. Входные значения функции представляют собой те числовые величины, которые были присвоены определенным переменным до выполнения программы, а выходными значениями являются числовые значения определенных переменных программы после завершения ее выполнения. Для вычисления функции мы просто выполняем программу, предварительно позаботившись о том, чтобы входные переменные содержали требуемые значения, а затем считываем значения выходных переменных, которые они будут иметь после завершения выполнения программы.

В этом контексте рассмотрим следующую программу:

```
incr X
```

Ее выполнение приводит к вычислению той же функции (функции следования), которая вычисляется машиной Тьюринга, описанной в разделе 12.2. Действительно, она увеличивает помещенное в переменную `X` значение на единицу. Аналогичным образом мы можем рассматривать содержимое переменных `X` и `Y` в качестве входных значений, а конечное содержимое переменной `Z` — как выходное значение для следующей программы:

```
copy Y to Z
while X not 0:
  incr Z
  decr X
```

Нетрудно убедиться, что эта программа реализует функцию сложения.

Исследователи показали, что наш простейший язык программирования можно использовать для выражения алгоритмов вычисления всех функций, вычислимых по Тьюрингу. Объединение этих результатов с тезисом Черча–Тьюринга подразумевает, что любая вычислимая функция может быть вычислена программой, написанной на простейшем языке. Следовательно, наш простейший язык программирования является универсальным языком программирования в том смысле, что если существует алгоритм для решения проблемы, то эта проблема может быть решена с помощью некоторой программы на данном простейшем языке. В свою очередь, наш простейший язык программирования теоретически может использоваться как язык программирования общего назначения.

Мы говорим *теоретически*, поскольку данный язык, безусловно, не настолько удобен, как язык Python или языки высокого уровня, описанные в главе 6. Тем не менее каждый из этих языков в качестве своего ядра непременно содержит все функции, входящие в состав нашего простейшего языка. Фактически именно это ядро обеспечивает универсальность каждого языка, все же прочие функции различных языков включены в них исключительно для удобства.

Хотя это будет не практично в среде прикладного программирования, такие языки, как наш простейший язык, находят применение в теоретических исследованиях в области компьютерных наук. Например, в приложении Д простейший язык программирования используется в качестве инструмента для решения вопроса об эквивалентности итеративных и рекурсивных структур, поднятого в главе 5. В этом приложении демонстрируется, что наше подозрение об их эквивалентности было на самом деле вполне оправданным.

12.3. Вопросы и упражнения

1. Покажите, что оператор `invert X;` (устанавливающий значение переменной `X` в нуль, если ее исходное значение отличалось от `0`, или присваивающий ей значение `1` в противном случае) может быть смоделирован программой на простейшем языке.
2. Покажите, что даже наш простейший язык содержит больше операторов, чем это действительно необходимо, продемонстрировав, что оператор `clear` может быть заменен комбинацией других операторов языка.
3. Покажите, что структура `if-else` может быть смоделирована на простейшем языке, т.е. напишите программу, воспроизводящую действие следующего выражения:

   ```
   if X not 0:
     S1
   else:
     S2
   ```

   Здесь параметры `S1` и `S2` представляют произвольные последовательности операторов.
4. Покажите, что каждый из операторов простейшего языка может быть выражен в терминах машинного языка, описанного в приложении В. (Следовательно, наш простейший язык может быть использован в качестве языка программирования для подобной машины.)

5. Как в представленном в этом разделе простейшем языке можно обрабатывать отрицательные числа?
6. Охарактеризуйте функцию, вычисляемую приведенной ниже программой на простейшем языке, предполагая, что входным значением является переменная X, а выходным — переменная Z.

```
clear Z
while X not 0:
  incr Z
  incr Z
  decr X
```

12.4. Невычислимые функции

В этом разделе мы ознакомимся с функцией, невычислимой по Тьюрингу, которая согласно тезису Черча–Тьюринга, по общепринятому мнению, является невычислимой и в общем смысле. Иначе говоря, это функция, вычисление которой лежит за пределами возможностей компьютеров.

Проблема остановки

Невычислимая функция, о которой пойдет речь ниже, связана с проблемой, известной как **проблема остановки**, которая (в неформальном смысле) представляет собой попытку заранее предсказать, завершится (или остановится) ли программа, если она будет запущена при определенных условиях. В качестве примера рассмотрим следующую программу на простейшем языке:

```
while X not 0:
  incr X
```

Если начать выполнение этой программы с исходным значением переменной X, равным 0, то цикл выполняться не будет, и программа сразу же остановится. Однако если мы начнем выполнение программы при любом другом исходном значении переменной X, то цикл будет выполняться вечно, и эта программа никогда не остановится.

Следовательно, в этом случае легко можно прийти к заключению, что выполнение программы будет остановлено только тогда, когда она запускается со значением переменной X, равным нулю. Однако по мере перехода к более сложным примерам задача прогнозирования поведения программы усложняется. На самом деле в некоторых случаях, как будет показано ниже, решить эту задачу невозможно. Но сначала необходимо формализовать используемую терминологию и точнее сфокусировать свои мысли.

Приведенный выше пример показал, что остановка программы в конечном итоге может зависеть от начальных значений ее переменных. Следовательно, если мы надеемся предсказать, завершится ли выполнение программы, следует проявлять точность в отношении этих начальных значений. Выбор, который мы собираемся сделать в отношении этих значений, на первый взгляд, может показаться довольно странным, однако не стоит отчаиваться. Наша цель — использовать преимущество метода, называемого **самосоотносимостью**, когда объект ссылается на самого себя. Эта уловка часто приводила к поразительным результатам в математике, начиная с информационных курьезов, подобных выражению "Это утверждение неверно", и заканчивая более серьезными парадоксами, представленными вопросом "Содержит ли набор всех наборов сам себя?" Таким образом, то, что мы собираемся сделать, заключается в подготовке почвы для серии рассуждений, аналогичных следующей логической цепочке: "Если это верно, то это неверно; но если это неверно, то это верно".

В нашем случае самосоотносимость будет достигнута путем присвоения переменным в программе начального значения, которое представляет саму эту программу. В этом отношении обратите внимание на то, что любую программу, написанную на нашем простейшем языке, можно рассматривать как одну длинную битовую комбинацию в формате "один байт на символ" с использованием кодировки ASCII, которая затем может быть интерпретирована как двоичное представление для (довольно большого) неотрицательного целого числа. Именно это целочисленное значение будет назначаться в качестве исходного значения переменным в программе.

Давайте рассмотрим, что произойдет, если применить подобный подход в случае следующей простой программы:

```
while X not 0:
  incr X
```

Необходимо установить, что произойдет, если запустить эту программу, исходно присвоив переменной `X` целочисленное значение, представляющее саму эту программу (рис. 12.6). В данном случае ответ очевиден. Поскольку переменная `X` будет иметь ненулевое значение, программа войдет в бесконечный цикл и никогда не завершится. Иной результат будет получен, если выполнить аналогичный эксперимент с программой

```
clear X
while X not 0:
  incr X
```

В этом случае программа немедленно остановится, поскольку в начале ее выполнения переменной `X` присваивается нулевое значение, и цикл никогда не выполняется независимо от исходного значения этой переменной.

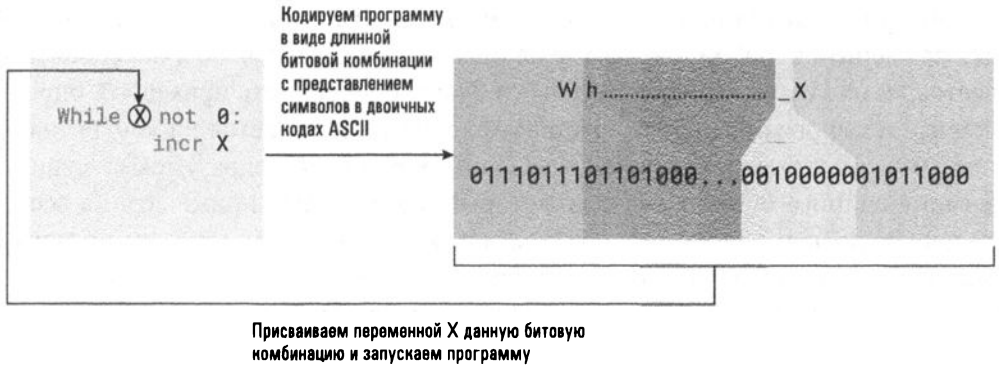


Рис. 12.6. Тестирование программы на самоостанавливаемость

Итак, давайте сделаем следующее определение: "Программа на простейшем языке является **самоостанавливающейся**, если выполнение этой программы в случае, когда все ее переменные инициализированы собственным закодированным представлением данной программы, приводит к завершению процесса". Более неформально это положение можно сформулировать так: "Программа является самоостанавливающейся, если ее выполнение прекращается, когда она запускается сама с собой в качестве входных данных. Вот та самая *самосоотносимость*, которая была обещана выше.

Отметим, что вопрос, является ли программа самоостанавливающейся, не имеет никакого отношения к задаче, для решения которой она была написана. Это просто свойство, которым любая программа на простейшем языке либо обладает, либо не обладает. Иначе говоря, любая написанная на простейшем языке программа является либо самоостанавливающейся, либо не самоостанавливающейся.

Теперь мы можем точно описать проблему остановки. Это проблема определения того, являются ли программы на простейшем языке самоостанавливающимися. Скоро будет показано, что не существует алгоритма для получения ответа на этот вопрос в целом. То есть не существует единого алгоритма, который, если на вход ему дадут какую-либо программу на простейшем языке, будет способен определить, является ли эта программа самоостанавливающейся. А отсюда следует, что решение проблемы остановки лежит за пределами возможностей компьютеров.

Тот факт, что мы, казалось бы, успешно решили проблему остановки в приведенных выше примерах, а теперь утверждаем, что проблема остановки является неразрешимой, может показаться вам противоречивым, поэтому давайте сделаем паузу для необходимых пояснений. Наблюдения, которые мы использовали в приведенных выше примерах, были уникальными для этих конкретных случаев и неприменимы во всех возможных ситуациях. То, что требуется для

решения проблемы остановки, — это единый общий алгоритм, который можно будет применить к любой программе на простейшем языке, чтобы определить, является ли она самоостанавливающейся. Наша способность применять определенные изолированные идеи для определения того, является ли конкретная программа самоостанавливающейся, никоим образом не подразумевает существования единого общего подхода, который можно будет применять во всех возможных случаях. Короче говоря, мы могли бы построить машину, которая может решить конкретную проблему остановки, но мы не можем построить единственную машину, которую можно было бы использовать для решения любой возможной проблемы остановки.



Основные положения для запоминания

- **У неразрешимой проблемы вполне могут быть конкретные варианты, имеющие алгоритмическое решение, однако для нее не существует единственного алгоритмического решения, позволяющего решить эту проблему во всех возможных ситуациях.**

Неразрешимость проблемы остановки

Теперь мы хотим продемонстрировать, что решение проблемы остановки лежит за пределами возможностей машин. Наш подход состоит в том, чтобы показать, что для решения проблемы потребуется алгоритм для вычисления невычислимой функции. Входами рассматриваемой функции будут закодированные версии программ на простейшем языке, а ее выходы ограничены значениями `0` и `1`. Если говорить точнее, мы определяем искомую функцию так, чтобы при входном значении, представляющем собой самоостанавливающуюся программу на простейшем языке, она генерировала выходное значение `1`, а для входного значения, представляющего не самоостанавливающуюся программу, — выходное значение `0`. Для краткости будем называть эту функцию *функцией остановки*.

Наша задача — показать, что функция остановки является невычислимой. Доказательство проведем методом от противного, т.е. покажем, что исходное предположение о вычислимости данной функции приводит к противоречию, вследствие чего мы будем вынуждены сделать вывод о ее невычислимости. Итак, покажем, что утверждение "функция остановки вычислима" не может быть верным. Все необходимые для этого аргументы приведены на рис. 12.7.

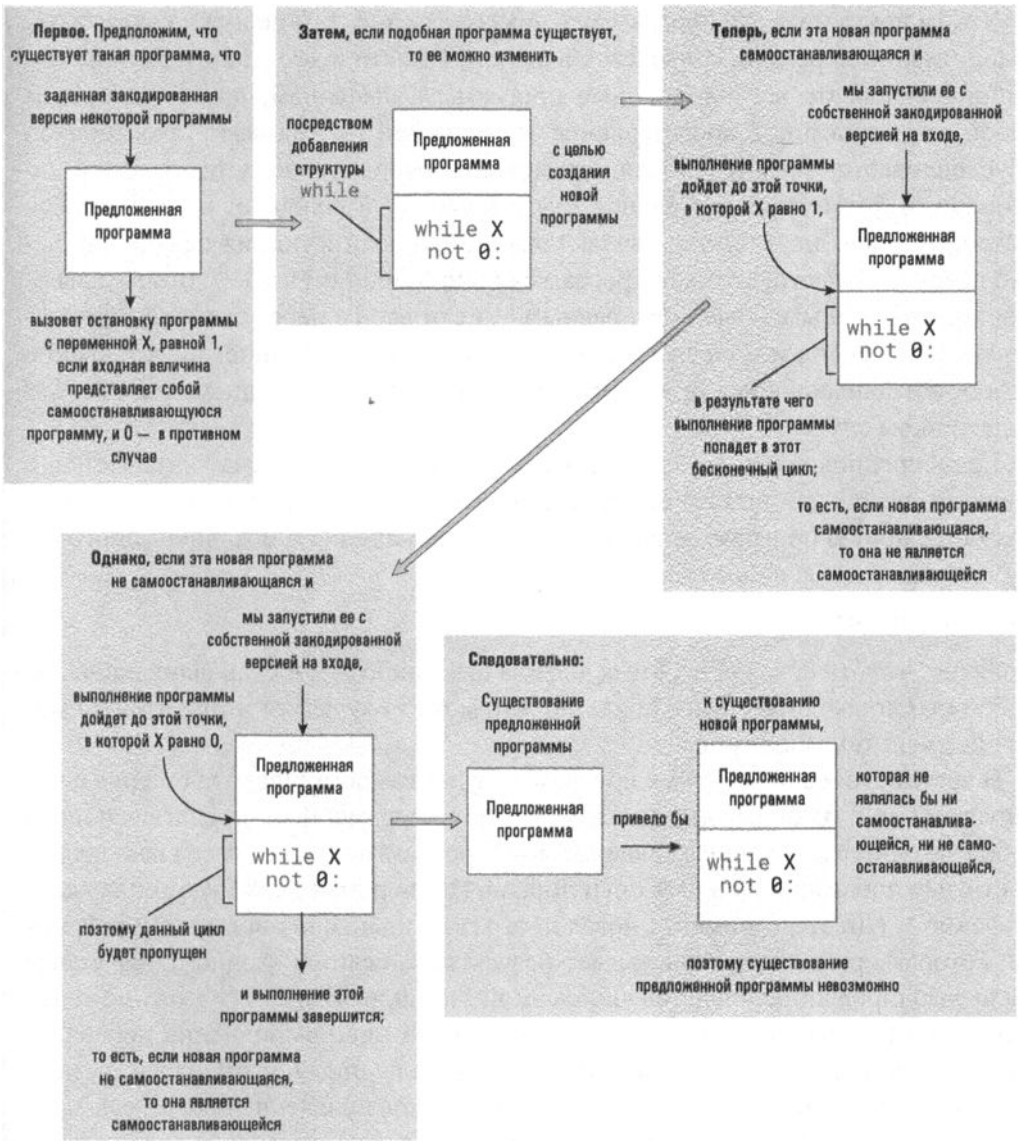


Рис. 12.7. Доказательство неразрешимости проблемы остановки программным путем

Если функция остановки вычислима, то (поскольку простейший язык является универсальным языком программирования) должна существовать программа на простейшем языке, которая позволяет ее вычислить. Другими словами, существует программа на простейшем языке, которая завершается с результирующим значением, равным `1`, если ее вход является закодированной версией самоостанавливающейся программы, и заканчивается с результирующим значением `0` в противном случае.

Чтобы применить эту программу в доказательстве, нам не нужно указывать, какая именно переменная является входной. Вместо этого достаточно просто инициализировать *все* переменные программы значением, представляющим собой закодированное представление тестируемой программы. Такой подход обусловливается тем, что любая переменная, которая не является входной, по своей сути будет такой, начальное значение которой не влияет на получаемое выходное значение. Отсюда мы заключаем, что если функция остановки вычислима, то существует такая программа на простейшем языке, которая завершается с выходным значением, равным `1`, если все ее переменные инициализируются значением, представляющим собой закодированное представление самоостанавливающейся программы, и завершается с выходным значением `0` в противном случае.

Если предположить, что выходная переменная программы называется `X` (если это не так, то достаточно просто переименовать ее переменные), можно получить новую программу, внеся некоторое изменение в исходную программу, например добавив в ее конец оператор

```
while X not 0:
```

Понятно, что эта новая программа должна быть либо самоостанавливающейся, либо нет. Однако ниже будет показано, что может случиться и так, что она может быть ни тем, ни другим.

В частности, если эта новая программа будет самоостанавливающейся и мы запускаем ее с одинаковым значением всех ее переменных, представляющим собой закодированное представление этой программы, то, когда ее выполнение достигнет добавленного нами оператора `while`, переменная `X` будет содержать значение `1`. (До этого момента новая программа идентична исходной программе, которая при выполнении выдает результат `1`, если ее входным значением было закодированное представление самой этой программы.) С этого момента выполнение программы попадает в бесконечный цикл выполнения добавленного нами оператора `while`, поскольку в его теле не предусмотрена какая-либо модификация значения переменной `X`. Однако такое поведение программы противоречит нашему предположению о том, что новая программа также является самоостанавливающейся. Следовательно, мы должны сделать вывод, что новая программа не является самоостанавливающейся.

Однако если новая версия программы не является самоостанавливающейся и мы начнем ее выполнение с одинаковым значением всех ее переменных, представляющим собой закодированное представление этой программы, то, когда ее выполнение достигнет добавленного нами оператора `while`, переменная `X` будет содержать значение `0`. (Это произойдет вследствие того, что команды, предшествующие оператору `while`, составляют исходную программу, которая

выдаст на выходе значение 0 только в том случае, если данная программа не является самоостанавливающейся.) В этом случае добавленный нами цикл while будет просто проигнорирован, и программа завершит свое выполнение. Но такое поведение характерно только для самоостанавливающихся программ, и мы вынуждены заключить, что наша измененная программа является самоостанавливающейся, подобно тому как ранее мы были вынуждены признать ее не самоостанавливающейся.

Подведем итоги. Очевидно, что в данном случае имеет место невозможная ситуация, когда, с одной стороны, программа должна быть или самоостанавливающейся, или нет, а с другой — она не может быть ни той, ни другой. Следовательно, наше исходное предположение привело к противоречию. Другими словами, функция остановки является *невычислимой*.

Итак, поскольку решение проблемы остановки построено на вычислении функции остановки, не являющейся вычислимой по Тьюрингу и, следовательно (согласно тезису Черча–Тьюринга), являющейся невычислимой вообще, то проблема остановки является примером **неразрешимой задачи**. В свою очередь, это означает, что решение этой задачи выходит за пределы возможностей вычислительной техники.

И наконец рассмотрим вопрос, уже обсуждавшийся в главе 11. Это главный основополагающий вопрос о том, достаточно ли мощности вычислительных машин для удовлетворения требований, выдвигаемых задачей создания искусственного интеллекта. Машины способны решать только такие задачи, которые имеют алгоритмическое решение, а выше было показано, что существуют проблемы, не имеющие алгоритмических решений. Следовательно, вопрос сводится к тому, воплощает ли естественный человеческий интеллект нечто большее, нежели просто выполнение исключительно алгоритмических процессов. Если это не так, то границы, которые мы здесь определили, являются границами и для человеческой мысли. Излишне говорить, что это очень спорный, а иногда и вызывающий бурные эмоции вопрос. Если, например, принять, что человеческий разум является не более чем запрограммированной машиной, то можно сделать обоснованный вывод, что люди не обладают свободной волей.



Основные положения для запоминания

- **Некоторые проблемы не могут быть решены с использованием какого бы то ни было алгоритма.**

12.4. Вопросы и упражнения

1. Является ли следующая программа на простейшем языке самоостанавливающейся? Поясните свой ответ.

```
incr X
decr Y
```

2. Является ли следующая программа на простейшем языке самоостанавливающейся? Поясните свой ответ.

```
copy X to Y
incr Y
incr Y
while X not 0:
  decr X
  decr X
  decr Y
  decr Y
decr Y
while Y not 0:
```

3. Что неверно в следующем сценарии?

   В некотором государстве каждый имеет собственный дом. Маляр государства должен покрасить все те и только те дома, которые не выкрашены их хозяевами.

   (*Подсказка.* Кто покрасит дом самого маляра?)

12.5. Сложность задач

В разделе 12.4 мы исследовали проблемы в терминах их разрешимости. В данном разделе мы остановимся на вопросе, всегда ли теоретически разрешимая задача имеет практически достижимое решение. Ниже будет показано, что некоторые проблемы, теоретически имеющие решение, оказываются настолько сложными, что фактически являются неразрешимыми с практической точки зрения.



Основные положения для запоминания

- **Многие проблемы *могут* быть решены в разумные сроки.**

Пространственная сложность

Сложность задачи может определяться с помощью не только оценки времени, необходимого для ее решения, но и оценки объема требуемой машинной памяти. Данный метод носит название *пространственная сложность*. В разделе показано, что временная сложность упорядочения списка из n наименований равна $O(n \lg n)$. Пространственная сложность этой же задачи не больше $O(n + 1) = O(n)$. Действительно, если для упорядочения такого списка применяется сортировка методом вставки, то это потребует пространства, равного объему списка, плюс пространства для временного хранения одной строки. Таким образом, если последовательно сортировать все большие и большие по размеру списки, можно обнаружить, что время, требуемое для выполнения каждого следующего задания, растет намного быстрее, чем объем требуемого для его выполнения пространства. В действительности это вполне закономерное явление. Поскольку на использование пространства требуется некоторое время, пространственная сложность никогда не будет расти быстрее временной.

Обычно используется компромисс между временной и пространственной сложностью. В некоторых приложениях может оказаться полезным предварительное выполнение определенных вычислений и запоминание их результатов в таблице, из которой впоследствии при необходимости они могут быть быстро извлечены. Подобная техника "табличного поиска" сокращает время, необходимое для решения задач, за счет увеличения пространства, необходимого для размещения таблицы. С другой стороны, для уменьшения объема используемой памяти часто применяется сжатие данных, что всегда связано с дополнительными затратами времени на выполнение сжатия и восстановления данных. Поскольку между двумя типами сложности можно достичь определенного компромисса, общая эффективность алгоритма зависит как от его временной, так и от его пространственной сложности.



Основные положения для запоминания

- **Эффективность алгоритма включает в себя как время его выполнения, так и использование им памяти.**

Измерение сложности задачи

Начнем с повторного рассмотрения проблемы эффективности алгоритмов, поднятой в разделе 5.6. В этом разделе для классификации алгоритмов в зависимости от времени, необходимого для их выполнения, мы ввели нотацию с использованием прописной буквы "тета" (Θ). Там же было установлено, что

алгоритм сортировки вставками относится к классу $\Theta(n^2)$, алгоритм последовательного поиска — к классу $\Theta(n)$, а алгоритм двоичного поиска — к классу $\Theta(\log_2 n)$. Здесь мы будем использовать эту систему классификации как средство, которое поможет нам при определении сложности задач. В данном случае мы должны построить систему классификации, которая укажет нам, какая задача является более сложной, и, в конце концов, поможет установить, что решение определенной задачи настолько сложно, что находится за пределами имеющихся практических возможностей.

Причина того, что данное исследование основывается на наших знаниях об эффективности алгоритмов, заключается в намерении измерять сложность задачи в терминах сложности ее решения. Будем считать задачу простой, если она имеет простое решение, и сложной, если для нее простого решения не существует. Отметим, что сам факт существования сложного решения некоторой задачи не означает, что эта задача обязательно сложная. В конце концов, для одной и той же задачи, как правило, существует много решений, одно из которых неизбежно будет сложнее других. Поэтому, чтобы прийти к заключению, что некоторая задача является сложной, необходимо доказать, что для нее *не существует* простых решений.

В область интересов компьютерных наук попадают только те задачи, которые могут быть решены с помощью машин. Решения подобных задач формулируются как алгоритмы. Поэтому сложность задачи определяется свойствами алгоритма, позволяющего найти ее решение. Если говорить точнее, то сложность простейшего алгоритма решения некоторой задачи определяет сложность самой этой задачи.

Но как можно измерить сложность алгоритма? К сожалению, термин *сложность* может трактоваться по-разному. Сложность может, например, определяться количеством принятых решений и разветвлений алгоритма. В таком случае сложным будет считаться алгоритм, представляющий собой набор запутанных и переплетающихся между собой указаний. Подобная трактовка может быть совместима с точкой зрения разработчика программного обеспечения, для которого наибольший интерес представляют вопросы разработки и отображения алгоритмов. Однако это не отражает понятия сложности с точки зрения машины. На самом деле при выборе следующей выполняемой инструкции машина не принимает никаких решений, а просто повторяет свой машинный цикл снова и снова, каждый раз выполняя инструкцию, указанную значением счетчика адреса. Поэтому машина может выполнять самый запутанный набор инструкций с той же легкостью, что и серию последовательно расположенных команд. Следовательно, данная интерпретация оценивает скорее уровень сложности, с которым приходится сталкиваться при разработке алгоритма, а не уровень сложности алгоритма самого по себе.



Основные положения для запоминания

- **Иногда более эффективные алгоритмы являются более сложными.**

Трактовка, более точно отражающая сложность алгоритма, может быть получена в ходе анализа свойств алгоритмов с машинной точки зрения. В этом контексте сложность алгоритма измеряется в терминах необходимого для его выполнения времени, которое, в свою очередь, пропорционально количеству действий, совершаемых машиной. Отметим, что это количество действий вовсе не равно числу инструкций, записанных в тексте программы. Например, цикл, тело которого в распечатке состоит из единственной инструкции вывода на печать, но выполнение которого стократно повторяется, при работе программы равносилен сотне отдельных инструкций вывода на печать. Значит, подобная программа будет считаться более сложной, чем набор из 50 отдельных операторов вывода на печать, хотя в последнем случае текст программы будет существенно длиннее. В конечном итоге данный подход к измерению *сложности* задачи связан со временем, затрачиваемым машиной на ее решение, а не с объемом программы, представляющей это решение.

Следовательно, задача считается *сложной*, если все ее существующие решения требуют больших затрат времени. Данное понятие сложности обычно называют **временной сложностью**. Косвенно вы уже познакомились с понятием временной сложности при изучении эффективности алгоритма в разделе 5.6. В конце концов, изучение эффективности алгоритмов представляет собой анализ их временной сложности, однако результаты обоих процессов абсолютно противоположны, т.е. в действительности определение "более эффективный" эквивалентно определению "менее сложный". Так, в терминах временной сложности последовательный алгоритм поиска (относящийся к классу $\Theta(n)$) является более сложным методом решения задачи поиска значения в списке, чем алгоритм двоичного поиска (который относится к классу $\Theta(\log_2 n)$).

А теперь давайте познакомимся с классификацией задач соответственно их временной сложности. Определим, что (временная) сложность задачи равна $\Theta(f(n))$, где $f(n)$ — некоторое математическое выражение, зависящее от n, если существует алгоритм решения задачи с временной сложностью $\Theta(f(n))$ и не существует алгоритма для решения этой задачи с большей временной эффективностью. Иными словами, (временная) сложность задачи определяется как (временная) сложность лучшего из ее решений. К сожалению, нахождение лучшего решения и доказательство, что найденное решение действительно является лучшим, само по себе является сложной задачей. В подобной ситуации для представления известных сведений о сложности задачи часто используется

O-нотация (произносится как "*нотация «O» большое*"), представляющая собой разновидность Θ-нотации. Точнее говоря, если $f(n)$ является математическим выражением, зависящим от n, и задача может быть решена с помощью алгоритма класса $\Theta(f(n))$, то мы говорим, что это задача сложности "«O» большое от $f(n)$", что записывается как $O(f(n))$. Таким образом, то, что задача принадлежит к классу $O(f(n))$, равносильно утверждению о существовании ее решения (не обязательно лучшего), сложность которого равна $\Theta(f(n))$.

Выполненные нами ранее исследования задач поиска и сортировки показывают, что задача поиска в списке из n строк (при этом наши знания о списке исчерпываются тем, что он был предварительно упорядочен) имеет сложность $O(\log_2 n)$, поскольку алгоритм двоичного поиска позволяет решить данную проблему. Более того, эти исследования показали, что проблема поиска действительно принадлежит к классу $\Theta(\log_2 n)$, так что алгоритм двоичного поиска представляет собой оптимальное решение данной проблемы. В то же время мы знаем, что проблема упорядочения такого списка (когда мы не имеем никакой информации о расположении его элементов) имеет сложность не более $O(n^2)$, поскольку алгоритм сортировки методом вставки позволяет решить данную задачу. Однако, как показывают исследования, проблема сортировки в действительности относится к классу $\Theta(n \log_2 n)$, а отсюда следует, что алгоритм сортировки методом вставки не является лучшим решением этой задачи (в терминах временно*й* сложности).

Примером лучшего решения данной проблемы является алгоритм сортировки слиянием. Данный подход заключается в многократном слиянии маленьких упорядоченных частей списка (при этом каждый раз образуются упорядоченные списки большего размера) до тех пор, пока весь список не будет правильно отсортирован. В каждом отдельном процессе слияния используется алгоритм слияния, с которым мы познакомились при обсуждении обработки последовательных файлов (см. рис. 9.15). Для удобства на рис. 12.8 мы вновь приводим этот алгоритм, но теперь уже в контексте слияния двух списков. Полный (рекурсивный) алгоритм сортировки методом слияния представлен на рис. 12.9 как процедура `MergeSort()` на рис. 12.9. При вызове этой процедуры для упорядочения некоторого списка она сначала проверяет, сколько позиций имеет список. Если меньше двух, то задача процедуры считается выполненной. В противном случае список делится на две части и вызываются копии этой же процедуры `MergeSort()` для сортировки полученных частей, после чего отсортированные части сливаются для получения полностью упорядоченного списка.

Для анализа сложности представленного алгоритма рассмотрим, сколько сравнений необходимо провести между элементами при слиянии списка длиной r элементов со списком длиной s элементов. Процесс слияния происходит путем сравнения элемента из одного списка с элементом из другого и помещения в выходной список меньшего из них. Поскольку на каждом этапе

выполняется одно сравнение, общее число еще не рассмотренных элементов каждый раз уменьшается на единицу. Так как в двух списках имеется всего $r + s$ элементов, то можно сделать вывод, что для слияния этих двух списков потребуется не более $r + s$ сравнений.

```
def MergeLists (InputListA, InputListB, OutputList):
  if (Оба входных списка пусты):
    Стоп, причем список OutputList пуст
  if (Список InputListA пуст):
    Объявить, что он окончен
  else:
    Объявить его первый элемент текущим элементом этого списка
  if (Список InputListB пуст):
    Объявить, что он окончен
  else:
    Объявить его первый элемент текущим элементом этого списка
  while (Ни один из входных списков не окончен):
    Поместить в OutputList элемент с "меньшим" значением ключа
    if (Этот элемент является последним в соответствующем входном списке):
      Объявить, что этот входной список пуст
    else:
      Объявить следующий элемент в этом входном списке его текущим элементом
  Начиная с текущего элемента входного списка, который еще не пуст,
      скопировать все его оставшиеся элементы в OutputList
```

Рис. 12.8. Процедура `MergeLists()`, выполняющая слияние двух списков

```
def MergeSort (List):
  if (Список List имеет более одного элемента):
    Применить процедуру MergeSort для сортировки первой половины списка
    Применить процедуру MergeSort для сортировки второй половины списка
    Применить процедуру MergeLists для слияния первой и второй половин списка
        List для получения отсортированной версии списка List.
```

Рис. 12.9. Рекурсивный алгоритм сортировки слиянием, реализованный в виде процедуры `MergeSort()`

Теперь рассмотрим алгоритм сортировки методом слияния в целом. В нем задача сортировки списка длиной n решается таким образом, что исходная задача распадается на две меньшие, каждая из которых заключается в упорядочении списка длиной приблизительно $n/2$. Эти две задачи, в свою очередь, разбиваются на четыре задачи сортировки списков длиной приблизительно $n/4$. Подобный процесс разделения может быть обобщен в виде древовидной структуры, представленной на рис. 12.10, в которой каждый узел дерева представляет собой отдельную задачу рекурсивного процесса, а отходящие от узла ветви — меньшие задачи, порожденные родительской. Эта структура позволяет

определить общее число сравнений, необходимых для процесса сортировки полного списка. Очевидно, что эта величина представляет собой сумму всех сравнений, выполняемых в каждом узле дерева.

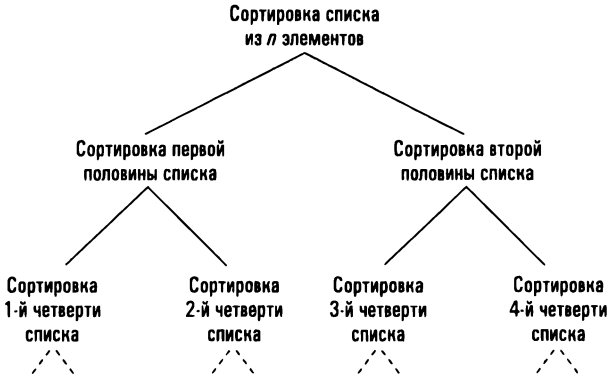


Рис. 12.10. Иерархическое представление множества задач, порожденных алгоритмом сортировки методом слияния

Прежде всего, определим число сравнений, выполняемых на каждом уровне дерева. Отметим, что каждый узел дерева представляет собой задачу упорядочения уникальной части исходного списка. Эта операция выполняется с помощью процедуры слияния уже упорядоченных списков и, следовательно, требует не больше операций сравнения, чем имеется строк в объединяемых списках, как было показано выше. Таким образом, выполнение операций слияния на каждом уровне дерева не требует больше операций сравнения, чем общее количество строк во всех частях списках. Но поскольку каждый уровень состоит из процедур сортировки частей расчлененного исходного списка, общее число сравнений будет не больше полной длины исходного списка. Поэтому на каждом уровне дерева выполняется не более n сравнений. (Кроме того, следует учесть, что самый нижний уровень дерева связан с обработкой списков, длина которых составляет одну или нуль строк, а потому вообще не требует выполнения сравнений.)

Теперь определим число уровней дерева. Предварительно отметим, что процесс разделения задач на меньшие составляющие будет продолжаться до тех пор, пока не будут получены списки длиной меньше двух элементов. Таким образом, количество уровней дерева определяется числом операций деления списка на два подсписка, выполняемых до тех пор, пока список начальной длины n не превратится в величину, не превосходящую 1. Количество таких операций равно $\log_2 n$ или, более точно, $[\log_2 n]$, где запись $[\log_2 n]$ означает округление точного значения $\log_2 n$ до следующего целого числа.

В результате общее число операций сравнения, выполняемых алгоритмом сортировки методом слияния при упорядочении списка длиной n, определяется как результат умножения числа сравнений, производимых на каждом уровне дерева, на общее количество уровней. Таким образом, можно заключить, что это число сравнений будет не больше $n[\log_2 n]$. Поскольку график функции $n[\log_2 n]$ имеет такое же поведение, как и функции $n \log_2 n$, можно принять окончательное решение, что алгоритм сортировки методом слияния относится к классу $O(n \log_2 n)$. Сравнив полученный результат с доказанным математиками утверждением о том, что задача сортировки списка относится к классу $\Theta(n \log_2 n)$, можем сказать, что алгоритм сортировки методом слияния представляет собой оптимальное решение задачи сортировки.

Задачи полиномиального и неполиномиального типов

Предположим, что $f(n)$ и $g(n)$ — это математические выражения. Тогда утверждение, что функция $g(n)$ ограничивает функцию $f(n)$, означает, что при возрастании аргумента n значение функции $f(n)$ непременно окажется больше значения функции $g(n)$ и будет оставаться большим при дальнейшем возрастании аргумента n. Другими словами, выражение "функция $g(n)$ ограничивает функцию $f(n)$" означает, что график функции $f(n)$ для больших значений n находится над графиком функции $g(n)$. Например, функция $\log_2 n$ ограничена функцией n (рис. 12.11, *а*), а функция $n \log_2 n$ — функцией n^2 (рис. 12.11, *б*).

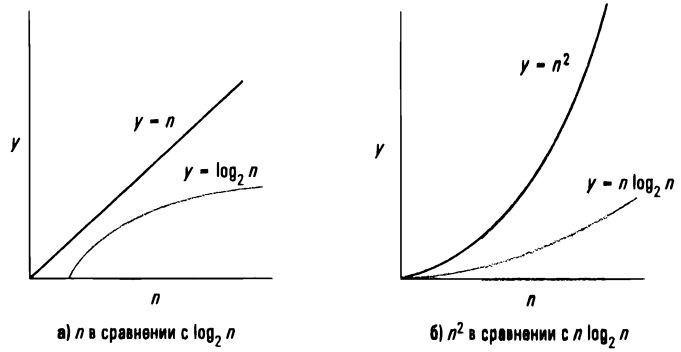


Рис. 12.11. Графики математических функций n, $\log_2 n$, $n \log_2 n$ и n^2

Будем говорить, что задача относится к **полиномиальному типу**, если она принадлежит классу $O(f(n))$, где функция $f(n)$ либо сама является полиномом, либо ограничивается некоторым полиномом. Совокупность всех задач полиномиального типа традиционно обозначается как **P**. Отметим, что выполненные нами выше исследования показывают, что задачи поиска в списке и упорядочения списка относятся к **P**.



Основные положения для запоминания

- **Разумное время означает, что число шагов алгоритма меньше или равно полиномиальной функции (постоянная, линейная, квадратная, кубическая и т.д.) от размера входных данных.**

Утверждение о том, что задача относится к полиномиальному типу, связано со временем, необходимым на ее решение. Часто говорят, что задача, принадлежащая **P**, может быть решена за полиномиальное время или же, что то же самое, имеет полиномиальное временное решение. Определение принадлежности задачи к множеству **P** является достаточно важным моментом в программировании, поскольку это тесно связано с существованием практического решения задачи. И действительно, задачи, не принадлежащие к множеству **P**, характеризуются крайне высоким временем выполнения даже при обработке умеренного объема входных данных. Например, рассмотрим задачу, решение которой состоит из 2^n этапов. Степенная функция 2^n не ограничивается никаким полиномом; если $f(n)$ является полиномом, то при увеличении значения аргумента n мы обнаружим, что значение функции 2^n всегда больше соответствующего значения функции $f(n)$. Это означает, что алгоритм сложности $\Theta(2^n)$ будет менее эффективным и, следовательно, потребует больше времени, чем алгоритм сложности $\Theta(f(n))$. Принято говорить, что алгоритм, сложность которого определяется степенной или экспоненциальной функцией, требует экспоненциального времени выполнения.

В качестве примера рассмотрим задачу определения состава всех возможных подгрупп, которые могут быть сформированы из группы, содержащей n человек. Поскольку существует $2^n - 1$ подобных подгрупп (мы допускаем, что подгруппа может состоять из всей группы, но не рассматриваем подгруппу, не содержащую ни одного человека), любой алгоритм решения данной задачи должен включать не менее $2^n - 1$ этапов. Следовательно, его сложность — не менее $\Theta(2^n - 1)$. Однако выражение $2^n - 1$ относится к экспоненциальному классу и не ограничивается никаким полиномом. Таким образом, при увеличении размера группы, из которой производится выборка, решение данной задачи становится чрезвычайно трудоемким.

В отличие от предыдущего примера, в котором сложность определялась объемом выходной информации, существуют еще более сложные задачи, несмотря на то что от них требуется лишь выдача простого ответа "да" или "нет". Примером такой задачи является определение истинности выражений, касающихся сложения действительных чисел. Например, мы легко можем определить, что на вопрос "Правда ли, что существует действительное число, которое, будучи

прибавленным к самому себе, даст в результате 6?" последует ответ "да", а на вопрос "Правда ли, что имеется ненулевое действительное число, которое, будучи прибавленным к самому себе, даст в результате 0?" последует ответ "нет". Однако с повышением сложности таких вопросов наши возможности дать на них ответ резко снижаются. Если потребуется дать ответы на большое количество подобных вопросов, может возникнуть соблазн обратиться за помощью к компьютеру. К сожалению, было доказано, что поиск ответов на такие вопросы требует экспоненциального времени. Таким образом, при повышении сложности вопросов компьютер не сможет выдавать ответы с требуемой скоростью.

Тот факт, что теоретически разрешимые, но не принадлежащие к множеству **P** задачи имеют столь огромную временную сложность, свидетельствует о том, что с практической точки зрения они являются неразрешимыми. В то же время задачи, имеющие практическое решение, обычно относятся к множеству **P**. Таким образом, определение границ множества **P** можно считать важным направлением исследований в области компьютерных наук.



Основные положения для запоминания

- **Некоторые проблемы не могут быть решены в разумные сроки даже для относительно небольших объемов входных данных.**

NP-задачи

Рассмотрим задачу коммивояжера, заключающуюся в том, что он должен посетить всех покупателей, проживающих в различных городах, не превысив при этом установленной сметы. Таким образом, данная задача сводится к нахождению пути (от его дома, соединяющего все требуемые города, и до его дома), суммарная протяженность которого не превысит определенной величины.

Традиционное решение данной задачи заключается в систематическом рассмотрении возможных путей, сравнении их протяженности с некоторым пределом, пока не будет найден приемлемый вариант или не будут рассмотрены все возможности. Однако данный подход не имеет полиномиального временного решения. По мере увеличения числа городов количество перебираемых возможных путей растет намного быстрее, чем любой полином. Следовательно, данное решение задачи коммивояжера является непрактичным, особенно если необходимо посетить много городов.

В результате мы приходим к выводу, что для решения данной задачи за разумное время необходимо найти более быстрый алгоритм. Наш интерес подогревается тем, что если удовлетворяющий условиям задачи путь существует

и если нам посчастливится выбрать его с первой же попытки, то описанный выше продолжительный алгоритм на самом деле завершится очень быстро. В частности, приведенный ниже набор операторов может быть выполнен очень быстро и потенциально позволяет найти решение поставленной задачи.

```
Выбрать один из возможных путей и вычислить его протяженность
if (Эта протяженность не больше заданной):
  Объявить об успехе
else:
  Не объявлять ни о чем
```

Однако данная последовательность команд не является алгоритмом в техническом смысле этого слова. Его первая команда не определена в том смысле, что она не указывает, какой именно путь следует выбирать, и при этом даже не дается никакой подсказки, как такое решение может быть принято. Вместо этого данный алгоритм полагается на креативность того механизма, который будет использоваться при выполнении программы, предоставляя ему право принять соответствующее решение самостоятельно. Говорят, что подобные команды не детерминированы, поэтому "алгоритм", содержащий подобные указания, мы будем называть **недетерминированным алгоритмом**.

Отметим, что при увеличении числа городов время, затрачиваемое на недетерминированный алгоритм, увеличивается относительно медленно. Процесс выбора произвольного пути состоит в простом составлении списка всех городов, что может быть выполнено за время, пропорциональное их количеству. Более того, время, требуемое на вычисление общей протяженности проложенного пути, также пропорционально количеству посещаемых городов, а время, необходимое для сравнения его длины с заданным пределом, вообще не зависит от числа городов. Следовательно, время, необходимое на выполнение недетерминированного алгоритма, ограничено какой-то полиномиальной функцией. Таким образом, при использовании недетерминированного алгоритма появляется возможность решения задачи коммивояжера за полиномиальное время.

Конечно, подобное недетерминированное решение нельзя считать полностью удовлетворительным. Фактически оно полагается на удачу. Однако этого вполне достаточно, чтобы предположить существование детерминированного решения задачи коммивояжера, которое выполнялось бы за полиномиальное время. Вопрос об истинности данного предположения остается открытым. В действительности задача коммивояжера является лишь одним из представителей большого класса задач, имеющих недетерминированное решение, выполнение которого требует полиномиального времени, для которых, однако, все еще не найдено детерминированное решение, имеющее полиномиальное время выполнения. Дразнящая эффективность недетерминированных решений указанных задач вызывает у некоторых надежду, что когда-нибудь будут

найдены эффективные детерминированные решения поставленных проблем, но большинство уверены в том, что данные задачи настолько сложны, что их решение невозможно сформулировать в рамках эффективных детерминированных алгоритмов.

Задача, имеющая недетерминированное решение с полиномиальным затрачиваемым временем, называется **недетерминированной полиномиальной задачей** или, для краткости, **NP-задачей**. Для обозначения множества NP-задач обычно употребляется обозначение **NP**. Отметим, что все задачи множества **P** также принадлежат и множеству **NP**, поскольку к любому (детерминированному) алгоритму можно добавить недетерминированную команду, не нарушив при этом его функционирования.

Вопрос о принадлежности всех NP-задач множеству **P** остается открытым, как мы только что убедились на примере задачи коммивояжера. Можно считать, что это наиболее известная проблема из числа еще не решенных в современной теории вычислительных систем. Ее успешное решение может иметь значительные последствия. Например, в следующем разделе рассказывается о существовании шифровальных систем, надежность которых основывается на нереальном количестве времени, затрачиваемом на решение задач, подобных представленной выше (о коммивояжере). Если будет доказано существование эффективных методов решения подобных задач, то данные шифровальные системы будут скомпрометированы.

Попытки разрешения вопроса о тождественности множеств **P** и **NP** привели к открытию класса задач, принадлежащих множеству **NP** и известных как **полные NP-задачи**. Эти задачи имеют следующую особенность: из их полиномиального временного решения могут быть выведены полиномиальные временные решения всех остальных задач, принадлежащих множеству **NP**. Иначе говоря, если удастся найти детерминированный алгоритм для решения одной из полных NP-задач за полиномиальное время, то данный алгоритм затем может быть расширен для решения любой NP-задачи за полиномиальное время. А это значит, что множество **NP** окажется тождественным множеству **P**. Задача коммивояжера относится именно к классу полных NP-задач.

Итак, в этом разделе мы обнаружили, что задачи могут быть классифицированы как разрешимые (имеющие алгоритмическое решение) и неразрешимые (не имеющие такого решения), как показано на рис. 12.12. Более того, множество разрешимых задач включает два подмножества. Одно состоит из задач полиномиального типа, имеющих практическое решение, а второе подмножество включает задачи неполиномиального типа, имеющие практически реализуемые решения только в определенных случаях или только при тщательно подобранных входных данных. Кроме того, существуют некие загадочные NP-задачи, которые пока еще не поддаются точной классификации.

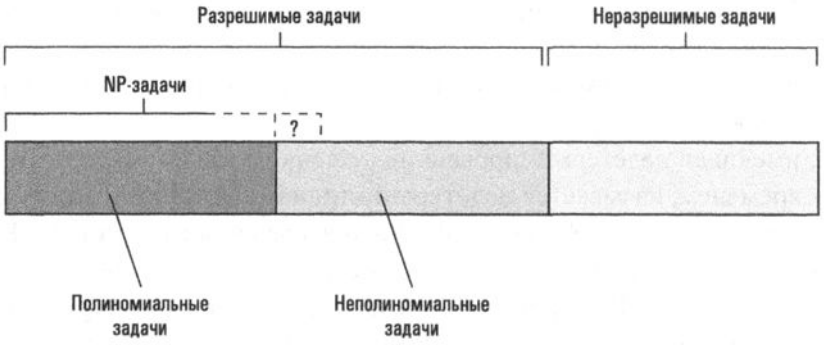


Рис. 12.12. Графическое обобщение результатов классификации задач

К сожалению, многие важные проблемы в области вычислительной техники относятся к классам неподдающихся анализу или NP-задач, и поэтому в большинстве случаев не могут быть решены в разумные сроки. Соответствующие примеры на настоящий момент времени включают компьютерное распознавание естественных языков и изображений, оптимальную маршрутизацию данных в телекоммуникационных сетях и управление интеллектуальными агентами в компьютерных играх. Во многих случаях, если алгоритм пока неизвестен или не позволяет решить задачу за приемлемое время, могут использоваться статистические или эвристические методы — правила "здравого смысла" или обоснованные предположения, позволяющие оценить, какой из нескольких возможных вариантов последующих действий является наиболее перспективным. Несмотря на то что не гарантируется точное или правильное решение проблемы, эвристический поиск часто может привести к приблизительным, по при этом вполне приемлемым результатам в разумные сроки.

Основные положения для запоминания

- **Некоторые проблемы могут быть решены, но не в приемлемые сроки. В этих случаях для поиска решений в разумные сроки могут оказаться полезными эвристические подходы.**
- **Эвристика — это метод, который позволяет найти приблизительное решение, когда классические методы не предоставляют возможности найти точное решение в разумные сроки.**
- **Методы эвристики могут быть полезны для быстрого нахождения приблизительного решения, когда точные методы оказываются слишком медленными.**

Детерминированный и недетерминированный

Во многих случаях существует лишь тонкая грань между детерминированным и недетерминированным "алгоритмом". Однако это различие довольно четкое и существенное. Детерминированный алгоритм не опирается на творческие возможности механизма, выполняющего алгоритм, в то время как недетерминированный "алгоритм" может на них полагаться. Например, сравните инструкцию

```
Идите до следующего перекрестка и поверните направо или налево.
```

и инструкцию

```
Идите до следующего перекрестка и поверните направо или налево в
зависимости от того, что скажет вам человек, стоящий на углу.
```

В любом случае действие, которое должен предпринять следующий указаниям человек, не определяется до момента фактического выполнения инструкции. Однако первая инструкция требует, чтобы человек, следуя указаниям, принял решение на основании собственного суждения, и поэтому является недетерминированной. Во второй инструкции такие требования к человеку, который следует указаниям, не предъявляются, — ему говорят, что надо делать на каждом этапе. Если несколько разных людей последуют первой инструкции, одни могут повернуть направо, а другие — налево. Однако, если несколько человек последуют второй инструкции и получат от стоящего на углу человека одну и ту же информацию, все они повернут в одном направлении. В этом заключается важное различие между детерминированными и недетерминированными "алгоритмами". Если детерминированный алгоритм выполняется многократно с одними и теми же входными данными, каждый раз будут выполняться одни и те же действия. Однако в случае недетерминированного "алгоритма" повторение его в идентичных условиях может приводить к разным действиям и результатам.

12.5. Вопросы и упражнения

1. Предположим, что задача может быть решена с помощью алгоритма класса $\Theta(2^n)$. Что можно сказать о сложности данной задачи?
2. Предположим, что задача может быть решена с помощью как алгоритма класса $\Theta(n^2)$, так и алгоритма класса $\Theta(2^n)$. Всегда ли один алгоритм будет превосходить другой?
3. Выпишите все подгруппы, которые можно сформировать из группы, в которую входят два человека: Алиса и Билл. Выпишите все подгруппы, которые можно сформировать из группы, в которую входят Алиса, Билл и Кэрол. Что можно сказать о подгруппах для группы, состоящей из четырех человек: Алисы, Билла, Кэрол и Девида?

4. Приведите пример задачи полиномиального типа. Приведите пример задачи неполиномиального типа. Приведите пример NP-задачи, о принадлежности которой к полиномиальному типу ничего не известно.
5. Если сложность алгоритма *X* больше, чем алгоритма *Y*, то обязательно ли понять алгоритм *X* будет сложнее, чем алгоритм *Y*? Поясните свой ответ.

12.6. Криптография с открытым ключом

В некоторых случаях тот факт, что проблему трудно решить, можно успешно превратить из пассива в актив. Особый интерес представляет проблема факторизации, т.е. нахождения простых множителей заданного целого числа — математическая задача, эффективное решение которой пока еще не найдено, если оно вообще существует. Например, вооружившись только бумагой и карандашом, вы можете обнаружить, что задача найти все простые множители даже относительно небольших числовых значений, например таких, как 2173, потребует больших затрат времени, а если анализируемое число будет настолько велико, что для его представления потребуется несколько сотен цифр, то задача будет практически неразрешимой, даже если воспользоваться современными технологиями с применением наилучших методов разложения числа на множители, известных в настоящее время.

Неудачи в поисках эффективного способа определения простых множителей больших целых чисел уже долгое время не дают покоя многим математикам, но в области криптографии эта ситуация успешно использовалась для создания популярного метода шифрования и дешифрования сообщений. Этот метод известен как **алгоритм RSA** — это название представляет собой аббревиатуру фамилий его изобретателей: Рона Ривеста (Ron *R*ivest), Ади Шамира (Adi *S*hamir) и Лена Адлемана (Len *A*dleman). Этот алгоритм предполагает шифрование сообщений с использованием одного набора значений, известного как **ключи шифрования**, и дешифрование этих сообщений с использованием другого набора значений, известных как **ключи дешифрования**. Люди, которые знают ключи шифрования, могут шифровать сообщения, но не могут их расшифровать. Единственный человек, который может расшифровывать сообщения, — тот, кто владеет ключами дешифрования. Таким образом, ключи шифрования можно распространять без каких-либо ограничений, не нарушая при этом безопасность системы.

Такие криптографические системы называются системами **шифрования с открытым ключом**, термин, который отражает тот факт, что используемые

для шифрования сообщений ключи могут быть общедоступными без нарушения безопасности системы. И действительно, ключи шифрования часто называют **открытыми ключами**, тогда как ключи дешифрования принято называть **закрытыми ключами** (рис. 12.13).

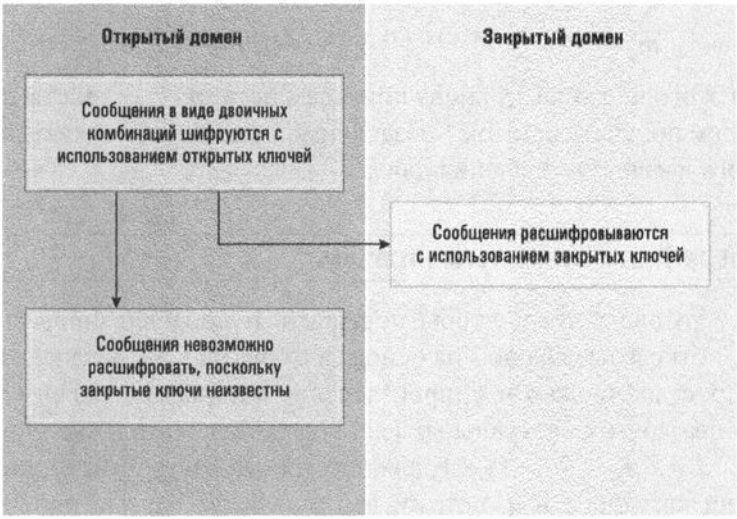


Рис. 12.13. Криптография с открытым ключом

Модульная арифметика

Для описания системы шифрования с открытым ключом RSA удобно использовать нотацию $x \% m$, которая читается как "x по модулю m", чтобы представить остаток, полученный при делении целого числа x на целое число m. Таким образом, $9 \% 7$ равно 2, потому что при делении числа 9 на число 7 получается остаток 2. Аналогичным образом $24 \% 7$ равно 3, потому что $24/7$ дает остаток 3, а $14 \% 7$ равно 0, потому что $14/7$ дает остаток 0. Обратите внимание, что $x \% m$ будет представлено самим x, если x является целым числом в диапазоне от 0 до $m - 1$. Например, $4 \% 9$ равно 4.

Математика говорит, что если p и q — *простые* числа, а m — целое число в диапазоне от 0 до pq (произведение p на q), то для любого целого положительного числа k будет выполняться следующее равенство:

$$1 = m^{k(p-1)(q-1)} \% pq$$

Обоснование этого утверждения мы здесь приводить не будем и ограничимся лишь рассмотрением иллюстрирующего его примера. Предположим, что p и q — это простые числами 3 и 5 соответственно, а m является целым числом 4. Тогда из рассматриваемого утверждения следует, что для любого целого

положительного числа k значение $m^{k(p-1)(q-1)}$ делится на 15 (произведение 3 и 5) с остатком 1. В частности, если $k = 1$, то

$$m^{k(p-1)(q-1)} = 4^{1(3-1)(5-1)} = 4^8 = 65\,536$$

Деление этого числа на 15 дает остаток 1, как ожидалось. Более того, если $k = 2$, то

$$m^{k(p-1)(q-1)} = 4^{2(3-1)(5-1)} = 4^{16} = 4\,294\,967\,296$$

При делении этого числа на 15 вновь получаем остаток *1*. В действительности мы всегда будем получать остаток 1 независимо от целого положительного числа, выбранного в качестве значения для k.

RSA-криптография с открытым ключом

Теперь все готово к тому, чтобы построить и проанализировать систему шифрования с открытым ключом на основе алгоритма RSA. Сначала выбераем два разных простых числа p и q, произведение которых обозначим как n. Затем выберем два других натуральных (т.е. целых положительных) числа e и d, таких, что $e \times d = k(p-1)(q-1) + 1$, для некоторого натурального числа k. Эти значения были названы e и d, потому, что они будут частью процесса шифрования (*encryption*) и дешифрования (*decryption*) соответственно. (Тот факт, что значения e и d *могут* быть выбраны таким образом, чтобы удовлетворять приведенному выше уравнению, является еще одним фактом из математики, который мы здесь подробно рассматривать не будем.)

Таким образом, было выбрано пять числовых значений: p, q, n, e и d. Значения e и n являются ключами *шифрования*. Значения d и n являются ключами *дешифрования*. Значения p и q используются исключительно для построения самой системы шифрования.

> Для получения дополнительных разъяснений давайте рассмотрим конкретный пример. Предположим, что в качестве значений для p и q были выбраны числа 7 и 13. Тогда $n = 7 \times 13 = 91$. Далее, для e и d можно использовать, например, значения 5 и 29, поскольку $5 \times 29 = 145 = 144 + 1 = 2(7-1)(13-1) + 1 = 2(p-1)(q-1) + 1$, как это и требуется. В результате получаем ключи шифрования $n = 91$ и $e = 5$, а ключи дешифрования — $n = 91$ и $d = 29$. Ключи шифрования можно передать всем, кто захочет отправить нам зашифрованные сообщения, а вот ключи дешифрования (а также значения p и q) следует оставить только у себя.

Теперь рассмотрим, как шифруются сообщения. С этой целью предположим, что в данный момент отправляемое сообщение уже представлено как комбинация битов (возможно, с использованием кодов ASCII или Unicode), и значение этой битовой комбинации, интерпретируемой как целое число в

двоичном представлении, будет меньше n. (Если это значение больше, чем n, можно разбить все сообщение на более мелкие сегменты и зашифровать каждый сегмент по отдельности.)

Предположим, что отправляемое сообщение при интерпретации его как целого числа в двоичном представлении будет иметь значение m. Тогда *зашифрованная* версия сообщения будет двоичным представлением значения $c = m^e \% n$. Иначе говоря, зашифрованное сообщение является двоичным представлением остатка, полученного при делении m^e на n.

В частности, если требуется зашифровать сообщение `10111` с использованием ключей шифрования $n = 91$ и $e = 5$, как было выведено в предыдущем примере, сначала примем во внимание, что `10111` — это двоичное представление десятичного числа 23, затем вычислим $23^e = 23^5 = 6\,436\,343$ и, наконец, разделим это значение на $n = 91$, чтобы получить остаток, который в данном случае равен 4. Следовательно, зашифрованная версия сообщения будет `100`, что является двоичным представлением десятичного числа 4.

Чтобы расшифровать сообщение, представляющее собой значение c в двоичном представлении, необходимо вычислить величину $c^d \% n$. Иначе говоря, мы вычисляем значение c^d и делим полученный результат на n для определения остатка от деления. Как можно было ожидать, именно этот остаток и будет значением m исходного сообщения, поскольку

$$\begin{aligned} c^d \% n &= m^{e \times d} \% n \\ &= m^{k(p-1)(q-1)+1} \% n \\ &= m \times m^{k(p-1)(q-1)} \% n \\ &= m \% n \\ &= m \end{aligned}$$

Здесь использованы известные нам сведения о том, что $m^{k(p-1)(q-1)} \% n = m^{k(p-1)(q-1)} \% pq = 1$ и что $m \% n = m$ (потому что $m < n$), как было заявлено ранее.

Продолжаем предыдущий пример: при получении сообщения `100` оно распознается как десятичное значение 4 и вычисляется значение $4^d = 4^{29} = 288\,230\,376\,151\,711\,744$, которое затем делится на $n = 91$, с получением остатка 23, что соответствует исходному сообщению `10111` при выражении полученного остатка в двоичной нотации.

Таким образом, система шифрования с открытым ключом RSA генерируется посредством выбора двух достаточно больших простых целых чисел p и q, на основании которых генерируются значения ключей n, e и d. Значения n и e используются для шифрования сообщений и, следовательно, являются

открытыми ключами. Значения n и d используются для расшифровки сообщений и являются *закрытыми* ключами (рис. 12.14). Вся прелесть этой системы в том, что знание того, как шифровать сообщения, не дает возможности расшифровать сообщения. Таким образом, ключи шифрования n и e могут свободно распространяться среди потенциальных отправителей сообщений. Даже если потенциальные противники получат эти ключи, они все равно не смогут с их помощью расшифровать перехваченные сообщения. Только тот человек, который знает закрытые ключи дешифрования, сможет расшифровать сообщения.

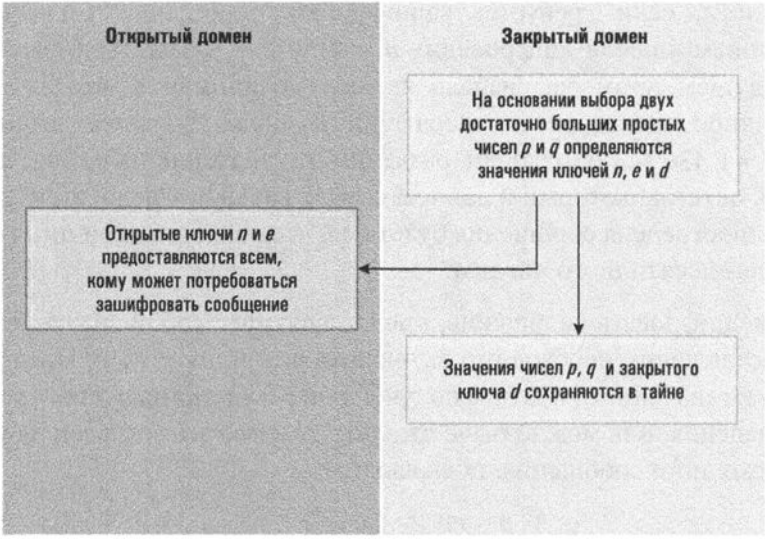


Рис. 12.14. Создание системы шифрования с открытым ключом по алгоритму RSA

Безопасность этой системы основана на предположении, что знание ключей шифрования n и e не позволяет вычислить ключи дешифрования n и d. Тем не менее существуют алгоритмы и для этого! Один из подходов состоит в том, чтобы разложить на простые множители число n, чтобы установить значения параметров p и q, а затем определить значение закрытого ключа d, найдя значение параметра k, такое, что $k(p-1)(q-1)+1$ будет нацело делиться на значение открытого ключа e (полученное частное и будет представлять собой значение закрытого ключа d). Однако выполнение первого этапа этого процесса может потребовать очень много времени, особенно если значения параметров p и q были выбраны достаточно большими. На самом деле, если значения p и q настолько велики, что их двоичные представления включают сотни цифр, то даже лучшим из известных алгоритмов факторинга потребуются годы, прежде чем значения p и q можно будет определить на основании известного значения n. Следовательно, содержимое зашифрованного сообщения будет оставаться

надежно защищенным еще долгое время даже после того, как оно фактически уже утратит свою ценность.

На сегодняшний день пока никто не нашел эффективного способа дешифрования сообщений, зашифрованных с использованием алгоритма RSA, без знания ключей дешифрования, и по этой причине шифрование с открытым ключом на основе алгоритма RSA сейчас широко используется для достижения необходимого уровня конфиденциальности при передаче сообщений через Интернет.

12.6. Вопросы и упражнения

1. Найдите простые множители числа 66 043. (Не тратьте на это слишком много времени. Дело в том, что в действительности выполнение этого задания может потребовать очень много времени.)
2. Используя открытые ключи $n = 91$ и $e = 5$, зашифруйте сообщение 101.
3. Используя закрытые ключи $n = 91$ и $d = 29$, расшифруйте сообщение 10.
4. Найдите подходящее значение для ключей дешифрования n и d в системе шифрования с открытым ключом по алгоритму RSA, созданной на основе простых чисел $p = 7$ и $q = 19$ и ключа шифрования $e = 5$.

ЗАДАНИЯ ПО МАТЕРИАЛУ ГЛАВЫ

(*Задания, отмеченные звездочкой, относятся к разделам для дополнительного чтения.*)

1. Покажите, как на простейшем языке может быть смоделирована следующая структура:

```
while X equals 0:
    .
    .
    .
```

2. Напишите на простейшем языке программу, присваивающую переменной Z значение 1, если переменная X меньше или равна переменной Y, и значение 0 — в любом другом случае.
3. Напишите на простейшем языке программу, которая присваивает переменной Z значение 2^x.

4. В каждом из приведенных ниже случаев напишите на простейшем языке последовательность команд, выполняющую требуемые действия.

   а. Присвоить переменной *Z* значение 0, если значение переменной *X* четное, в противном случае присвоить переменной *Z* значение 1.

   б. Вычислить сумму всех целых чисел от 0 до *X*.

5. Напишите на простейшем языке программу для вычисления частного от деления значения *X* на *Y*. Остатком от деления пренебрегайте, т.е. в результате деления 1 на 2 будет получено значение 0, а при делении 5 на 3 — значение 1.

6. Опишите функцию, вычисляемую с помощью приведенной ниже программы на простейшем языке, предполагая, что входные значения функции представлены переменными *X* и *Y*, а ее выходное значение — переменной *Z*.

```
copy X to Z
copy Y to Aux
while Aux not 0:
  decr Z
  decr Aux
```

7. Опишите функцию, вычисленную с помощью приведенной ниже программы на простейшем языке, полагая, что входные значения этой функции представлены переменными `X` и `Y`, а ее выходное значение помещается в переменную `Z`.

```
clear Z
copy X to Aux1
copy Y to Aux2
while Aux1 not 0:
  while Aux2 not 0:
    decr Z
    decr Aux2
  decr Aux1
```

8. Напишите на простейшем языке программу, которая будет выполнять операцию "исключающее ИЛИ" для значений в переменных `X` и `Y`, помещая ее результат в переменную `Z`. Можете воспользоваться предположением, что начальные значения переменных `X` и `Y` ограничены целочисленными значениями из `0` и `1`.

9. Покажите, что если мы в программах на простейшем языке допустим возможность присваивания операторам метки в виде целочисленных значений, и заменим структуру цикла `while` условным оператором ветвления, представленным в форме

```
if name not 0:
  goto label
```

где `name` — любая переменная, а `label` — целочисленное значение, используемое в качестве метки оператора в другом месте программы, то вновь полученный язык по-прежнему будет универсальным языком программирования.

10. В этой главе было показано, как утверждение

```
copy name1 to name2
```

можно смоделировать в нашем простейшем языке. Покажите, как можно было бы смоделировать это утверждение, если бы структура цикла `while` в простейшем языке была заменена циклом с постусловием, выраженным в виде

```
repeat:...
  until (name equals 0)
```

11. Покажите, что наш простейший язык останется универсальным языком, если оператор `while` будет заменен циклом с постусловием, выраженным в форме

```
repeat:
  ...
  until (name equals 0)
```

12. Спроектируйте машину Тьюринга, которая, начав работу, никогда не остановится и при этом будет использовать единственную ячейку ленты.

13. Разработайте модель машины Тьюринга, которая будет помещать нули во все ячейки, расположенные левее текущей, пока не встретит ячейку, содержащую символ звездочки.

14. Предположим, что на ленте машины Тьюринга последовательности нулей и единиц ограничиваются с обеих сторон звездочками. Спроектируйте машину Тьюринга, которая будет циклически сдвигать эти последовательности на одну ячейку влево в предположении, что машина начинает работу с крайней справа ячейки, содержащей звездочку.

15. Разработайте машину Тьюринга, которая будет инвертировать последовательность нулей и единиц, расположенную между текущей ячейкой ленты (содержащей звездочку) и ближайшей звездочкой слева.

16. Сформулируйте тезис Черча–Тьюринга.

17. Является ли приведенная ниже программа на простейшем языке самоостанавливающейся? Поясните свой ответ.

```
copy X to Y
incr Y
incr Y
while X not 0:
  decr X
  decr X
```

```
  decr Y
  decr Y
decr Y
while Y not 0:
  incr X
  decr Y
while X not 0:
```

18. Является ли приведенная ниже программа на простейшем языке самоостанавливающейся? Поясните свой ответ.

```
while X not 0:
```

19. Является ли приведенная ниже программа на простейшем языке самоостанавливающейся? Поясните свой ответ.

```
while X not 0:
  decr X
```

20. Проанализируйте истинность пары приведенных ниже утверждений.

```
Следующее утверждение истинно.
Предыдущее утверждение ложно.
```

21. Проанализируйте истинность следующего утверждения: "Кок готовит для тех людей на судне, кто сам не готовит себе еду". (А кто готовит еду для кока?)

22. Предположим, вы собираетесь посетить страну, в которой каждый человек является либо абсолютно правдивым, либо полным лжецом. (Правдивый всегда говорит правду, а лжец всегда лжет.) Какой единственный вопрос можно было бы задать человеку, чтобы сразу определить, кто перед вами: правдивый человек или лжец?

23. Охарактеризуйте значение машин Тьюринга в теории компьютерных наук.

24. Охарактеризуйте значение проблемы остановки в теории компьютерных наук.

25. Предположим, что вам необходимо выяснить, приходится ли у кого-то из группы людей день рождения на определенную дату. Один из возможных подходов — опрашивать членов группы по одному. Если вы воспользуетесь этим подходом, то возникновение какого события скажет вам, что такой человек в группе имеется? А какое событие подскажет вам, что такого человека в группе нет? Теперь предположим, что вам необходимо выяснить, обладает ли хотя бы одно из положительных целых чисел определенным свойством, и вы воспользовались тем же самым подходом, заключающимся в систематическом тестировании целых чисел по одному. Если какое-то целое число действительно имеет искомое свойство, то как вы об этом узнаете? Но если искомого свойства нет ни у одного из

целых чисел, то как бы вы могли об этом узнать? Является ли задача тестирования с целью установить, является ли гипотеза истинной, симметричной по отношению к задаче тестирования с целью установить, является ли гипотеза ложной?

26. Является ли поиск определенного значения в списке проблемой полиномиального типа? Аргументируйте ваш ответ.

27. Создайте алгоритм определения, является ли данное положительное число простым. Является ли ваше решение эффективным? Имеет ли предложенное вами решение полиномиальный тип или оно является неполиномиальным?

28. Всегда ли полиномиальное решение задачи лучше экспоненциального? Объясните свой ответ.

29. Означает ли факт существования полиномиального решения задачи возможность ее решения за практически приемлемый промежуток времени? Поясните ваш ответ.

30. Перед программистом поставили задачу разделить группу из четного количества людей на две разные подгруппы так, чтобы между подгруппами была максимально возможная разница в сумме возрастов людей, входящих в каждую из них. Он предложил решение, состоящее в переборе всех возможных пар групп, с вычислением суммарного возраста в каждой подгруппе и последующим выбором пары с максимальной разницей вычисленных значений. В то же время другой программист предложил сначала упорядочить исходную группу людей по возрасту, а потом создать одну подгруппу из более пожилой половины упорядоченной группы, а в другую включить всех оставшихся и более молодых. Каковы сложности каждого из предложенных решений? К какому классу относится поставленная задача сама по себе: к полиномиальному, неполиномиальному или к NP-задачам?

31. Почему метод генерации всех возможных переупорядочиваний списка с последующим выбором желаемого варианта упорядочивания не является удовлетворительным способом сортировки списка?

32. Предположим, что лотерея основана на правильном выборе четырех целочисленных значений, каждое из которых находится в диапазоне от 1 до 50. Кроме того, предположим, что джекпот стал настолько большим, что стало выгодным купить отдельный лотерейный билет для каждой возможной комбинации. Если на покупку одного билета требуется одна секунда, сколько времени займет покупка билетов на все возможные в лотерее комбинации чисел? Как изменится это время в том случае, если для выигрыша в лотерее нужно будет правильно выбрать пять чисел вместо четырех? Какое отношение эта проблема имеет к материалу из данной главы?

33. Является ли приведенный ниже алгоритм детерминированным? Поясните ваш ответ.

```
def mystery (Число):
  if (Число > 5):
    Ответ "Да".
  else:
    Взять число меньше 5 и вывести
      это число в качестве ответа
```

34. Является ли приведенный ниже алгоритм детерминированным? Поясните ваш ответ.

```
Поезжайте прямо
На третьем перекрестке спросите у стоящего на углу человека,
  куда вам повернуть – направо или налево
Поверните так, как вам сказали
Остановитесь, проехав два дома
```

35. Укажите недетерминированные участки данного алгоритма.

```
Выберите три числа в диапазоне от 1 до 100
if (сумма выбранных чисел больше 150):
  Ответ "Да".
else:
  Выберите одно из используемых чисел и
    выведите его в качестве ответа
```

36. Какую временную сложность имеет данный алгоритм — полиномиальную или неполиномиальную? Поясните ваш ответ.

```
def mystery (ПоследовательностьЧисел):
Выберите несколько чисел из
  ПоследовательностьЧисел
if (сумма выбранных чисел превышает 125):
  Ответ "Да"
else:
  не давать никакого ответа
```

37. Какие из приведенных ниже задач относятся к множеству **P**?

а. Задача сложности n^2

б. Задача сложности $3n$

в. Задача сложности $n^2 + 2n$

г. Задача сложности $n!$

38. Опишите различия между доказательством, что задача имеет полиномиальный тип, и утверждением, что она является недетерминированной полиномиальной задачей.

39. Приведите пример задачи, относящейся и к множеству **P**, и к множеству **NP**.

40. Предположим, что имеются два алгоритма решения некоторой задачи. Один имеет временную сложность n^4, а другой — $4n$. Для какого объема входных данных первый алгоритм будет предпочтительнее второго?

41. Предположим, требуется решить задачу коммивояжера для случая, когда существует 15 городов, из которых каждые два соединены единственной дорогой. Сколько существует различных путей, проходящих по всем городам? Сколько времени уйдет на вычисление длины всех путей, если предположить, что вычисление каждого из них занимает одну микросекунду?

42. Сколько сравнений потребуется выполнить при использовании алгоритма сортировки методом слияния (см. рис. 12.9 и 12.8) для упорядочения списка имен "Алиса, Боб, Кэрол, Дэвид"? Сколько сравнений понадобится в случае упорядочения списка "Алиса, Боб, Кэрол, Дэвид, Элейн"?

43. Приведите пример задачи каждого из типов, указанных на рис. 12.12.

44. Разработайте алгоритм для нахождения целочисленного решения уравнения $x^2 + y^2 = n$, где n — некоторое заданное положительное число. Определите временную сложность вашего алгоритма.

45. Еще одна проблема, относящаяся к категории полных NP-задач, — это **задача о рюкзаке**, которая состоит в определении, какие числа из заданного списка при суммировании дают в результате некоторое наперед заданное значение. Например, числа 257, 388 и 782 являются теми элементами списка

642 257 771 388 391 782 304

сумма которых равна 1427. Найдите в этом списке числа, сумма которых равна 1723. Какой алгоритм решения задачи вы использовали? Какова сложность этого алгоритма?

46. Охарактеризуйте сходство между задачей коммивояжера и задачей о рюкзаке (см. задание 45).

47. Приведенный ниже алгоритм упорядочения списка называется сортировкой **по методу пузырька**. Сколько сравнений между элементами списка потребуется для упорядочения списка n элементов?

```
def BubbleSort (Список):
  Counter = 1
  while (Counter < Количество элементов в списке):
  N = Количество элементов в списке
  while (N > 1):
    if (N-й элемент списка меньше предшествующего ему
        значения):
        Поменять местами N-й элемент и предшествующее
          ему значение
    N = N - 1
```

48. Используйте процедуру шифрования с открытым ключом по алгоритму RSA для шифрования сообщения `110` с использованием открытых ключей $n = 91$ и $e = 5$.

49. Используйте процедуру шифрования с открытым ключом по алгоритму RSA для расшифровки сообщения `111` с использованием закрытых ключей $n = 133$ и $d = 5$.

50. Предположим, вы знаете, что открытыми ключами в системе шифрования с открытым ключом на основе алгоритма RSA являются $n = 77$ и $e = 7$. Что будут представлять собой закрытые ключи в этой ситуации? Что позволит вам решить эту проблему в разумные сроки?

51. Найдите простые множители числа 107 531. Как эта задача связана с материалом данной главы?

52. К какому заключению можно прийти, если положительное целое число n не имеет целочисленных простых множителей в диапазоне от 2 до квадратного корня из n? Что это может сказать вам в отношении задачи нахождения простых множителей целых положительных чисел?

ОБЩЕСТВЕННЫЕ И СОЦИАЛЬНЫЕ ВОПРОСЫ

Следующие вопросы приводятся для того, чтобы помочь вам разобраться в некоторых этических, общественных и юридических аспектах использования вычислительной техники, а также в ваших собственных воззрениях и тех принципах, на которых они основаны. Задача не сводится к тому, чтобы просто дать ответы на предложенные вопросы. Вы должны также понять, почему вы ответили именно так, а не иначе, и насколько ваши суждения по различным вопросам согласуются между собой.

1. Предположим, что выполнение наилучшего алгоритма для решения задачи потребует 100 лет. К какому разряду вы отнесете данную задачу — к разрешимым или к неразрешимым?

2. Имеют ли граждане право шифровать свою переписку для предотвращения возможного контроля со стороны государственных служб? Предусматривает ли ваш ответ "надлежащее" обеспечение правопорядка? Кто решает, что именно означает понятие "надлежащее обеспечение правопорядка"?

3. Если рассматривать человеческий разум как алгоритмическое устройство, то как отразится на человечестве признание тезиса Тьюринга? Насколько вы верите в то, что машины Тьюринга имеют вычислительные возможности, сравнимые с возможностями человеческого разума?

4. Выше было показано, что существуют различные вычислительные модели (конечные таблицы, алгебраические формулы, машины Тьюринга и т.д.), имеющие разные вычислительные возможности. Есть ли различия в вычислительных возможностях разных организмов? Есть ли различия в вычислительных возможностях разных людей? Если да, то могут ли люди с более высокими способностями использовать эти способности для обеспечения себе более высокого уровня жизни?
5. Сегодня существуют веб-сайты и приложения для смартфонов, обеспечивающие доступ к картам практически любого города мира. Назначение этих сайтов — помочь найти конкретный адрес, причем они позволяют увеличивать изображения в целях детального просмотра застройки в ближайшем окружении. Многие из этих карт городов также предоставляют доступ к снимкам со спутников со сходными возможностями увеличения, обеспечивающими просмотр детальных изображений отдельных зданий и их окружения. Исходя из этих реальных возможностей, рассмотрим следующую вымышленную ситуацию. Предположим, что спутниковые изображения были дополнены возможностью просмотра видеоизображений в реальном времени, подобных прямой видеотрансляции. Далее предположим, что средства получения видеоизображений со спутников были дополнительно расширены за счет использования технологии обработки инфракрасного излучения. В результате появилась возможность наблюдать за вами в вашем же собственном доме на протяжении 24-х часов в сутки. На какой именно стадии указанной последовательности модернизации системы было нарушено ваше право на невмешательство в личную жизнь? На какой стадии, по вашему мнению, мы в своем сценарии переоценили возможности современных спутниковых систем? Как вы думаете, насколько изложенная ситуация является действительно вымышленной?
6. Предположим, что некоторая компания разработала и запатентовала систему шифрования. Должно ли правительство той страны, в которой расположена данная компания, иметь право на использование данной системы в целях обеспечения национальной безопасности? Должно ли правительство иметь право в целях обеспечения национальной безопасности ограничивать коммерческое использование компанией данной системы? Что изменится, если эта компания будет многонациональной?
7. Предположим, вы покупаете продукт, внутренняя структура которого зашифрована. Есть ли у вас право расшифровать структуру этого продукта? Если да, имеете ли вы право использовать эту информацию в коммерческих целях? А как насчет некоммерческого использования? Что если шифрование было выполнено с использованием секретной системы

шифрования и вы открыли этот секрет? Есть ли у вас право поделиться этим секретом с другими?

8. В свое время философ Джон Дьюи (John Dewey, 1859–1952) ввел термин "ответственная технология". Приведите несколько примеров того, что вы назвали бы "ответственной технологией". На основе своих примеров сформулируйте собственное определение "ответственной технологии". Практикует ли общество "ответственные технологии" в течение последних 100 лет? Должны ли быть предприняты какие-либо действия для обеспечения этого? Если да, то какие именно действия? Если нет, то почему?

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Garey M.R., Johnson D.S. *Computers and Intractability.* — New York: W.H. Freeman, 1979.
2. Hamburger H., Richards D. *Logic and Language Models for Computer Science.* — Englewood Cliffs, NJ: Prentice-Hall, 2002.
3. Hofstadter D.R. *Godel, Escher, Bash: An Eternal Golden Braid.* — St. Paul, MN: Vintage, 1980.
4. Hopcroft J.E., Motwani R., Ullman J.D. *Introduction to Automata Theory, Languages, and Computation,* 3rd ed. Boston, MA: Addison-Wesley, 2007. (Имеется русский перевод 2-го издания этой книги: Хопкрофт Дж., Мотвани Р., Ульман Дж. *Введение в теорию автоматов, языков и вычислений*, 2-е изд. — М.: Издательский дом "Вильямс", 2001.)
5. Lewis H.R., Papadimitriou C.H. *Elements of the Theory of Computation.,* 2nd ed. — Englewood Cliffs, NJ: Prentice-Hall, 1998.
6. Rich E. *Automata. Computability, and Complexity: Theory and Application.* — Upper Saddle River, NJ: Prentice-Hall, 2008.
7. Sipser M. *Introduction to the Theory of Computation,* 3rd ed. — Boston, MA: Cengage Learning, 2012.
8. Smith C., Kinber E. *Theory of Computing: A Gentle Introduction.* — Englewood Cliffs, NJ: Prentice-Hall, 2001.
9. Sudkamp T.A. *Languages and Machines: An Introduction to the Theory of Computer Science*, 3rd ed. — Boston, MA: Addison-Wesley, 2006.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. Мао В. *Современная криптография: теория и практика*. — М.: Издательский дом "Вильямс", 2005.
2. Фергюсон Н., Шнайер Б. *Практическая криптография*. — М. : Издательский дом "Вильямс", 2004.
3. Столлингс В. *Криптография и защита сетей*, 2-е изд. — М.: Издательский дом "Вильямс", 2001.
4. Джексон П. *Введение в экспертные системы*, 3-е изд. — М.: Издательский дом "Вильямс", 2001.
5. Шнайер Б. *Прикладная криптография: протоколы, алгоритмы и исходные коды на языке С, 2-е юбилейное издание*. — СпБ.: ООО "Диалектика", 2017.

Приложение **A**

Код ASCII

Ниже приведен неполный список ASCII-кодов символов. Во втором столбце указаны двоичные значения, причем к исходным семиразрядным кодам слева приписаны нули — для получения восьмибитовых кодов, общепринятых в настоящее время, а в третьем столбце даны шестнадцатеричные значения кодов.

Символ	ASCII-код	Hex	Символ	ASCII-код	Hex	Символ	ASCII-код	Hex
Перевод строки	00001010	0A	>	00111110	3E	^	01011110	5E
Возврат каретки	00001101	0D	?	00111111	3F	_	01011111	5F
пробел	00100000	20	@	01000000	40	`	01100000	60
!	00100001	21	A	01000001	41	a	01100001	61
"	00100010	22	B	01000010	42	b	01100010	62
#	00100011	23	C	01000011	43	c	01100011	63
$	00100100	24	D	01000100	44	d	01100100	64
%	00100101	25	E	01000101	45	e	01100101	65
&	00100110	26	F	01000110	46	f	01100110	66
'	00100111	27	G	01000111	47	g	01100111	67
(	00101000	28	H	01001000	48	h	01101000	68
)	00101001	29	I	01001001	49	i	01101001	69
*	00101010	2A	J	01001010	4A	j	01101010	6A
+	00101011	2B	K	01001011	4B	k	01101011	6B
'	00101100	2C	L	01001100	4C	l	01101100	6C

Окончание таблицы

Символ	ASCII-код	Hex	Символ	ASCII-код	Hex	Символ	ASCII-код	Hex
-	00101101	2D	M	01001101	4D	m	01101101	6D
.	00101110	2E	N	01001110	4E	n	01101110	6E
/	00101111	2F	O	01001111	4F	o	01101111	6F
0	00110000	30	P	01010000	50	p	01110000	70
1	00110001	31	Q	01010001	51	q	01110001	71
2	00110010	32	R	01010010	52	r	01110010	72
3	00110011	33	S	01010011	53	s	01110011	73
4	00110100	34	T	01010100	54	t	01110100	74
5	00110101	35	U	01010101	55	u	01110101	75
6	00110110	36	V	01010110	56	v	01110110	76
7	00110111	37	W	01010111	57	w	01110111	77
8	00111000	38	X	01011000	58	x	01111000	78
9	00111001	39	Y	01011001	59	y	01111001	79
:	00111010	3A	Z	01011010	5A	z	01111010	7A
;	00111011	3B	[	01011011	5B	{	01111011	7B
<	00111100	3C	\	01011100	5C	|	01111100	7C
=	00111101	3D	]	01011101	5D	}	01111101	7D

Приложение **Б**

Электронные схемы обработки чисел в двоичном дополнительном коде

В этом приложении представлены электронные схемы, предназначенные для изменения знака числа на противоположный и для сложения чисел, представленных в двоичном дополнительном коде. Начнем со схемы, показанной на рис. Б.1, которая позволяет преобразовать четырехразрядное число в двоичном дополнительном коде в битовую комбинацию, представляющую то же число, но с противоположным знаком. Например, двоичный код числа 3 будет преобразован в двоичное представление числа –3. Приведенная схема выполняет данную операцию в соответствии с алгоритмом, описанным в главе 1. Следуя этому алгоритму, схема копирует входную битовую комбинацию на выход в направлении справа налево до тех пор, пока не встретит разряд со значением `1`, а затем формирует на выходе дополнения каждого оставшегося входного бита. Поскольку на первый вход крайнего справа логического элемента XOR (исключающее "ИЛИ") постоянно поступает значение `0`, этот элемент просто передает на выход значение на другом его входе. Однако этот выходной сигнал одновременно поступает и на первый вход следующего логического элемента XOR. Если это выходное значение будет равно `1`, то этот логический элемент XOR сформирует на выходе дополнение для его второго входного сигнала. Кроме того, этот же единичный сигнал от первого элемента XOR через логический элемент OR (логическое "ИЛИ") подается на правый вход третьего элемента XOR, чтобы оказать соответствующее влияние на работу этого логического

элемента. Таким образом, первая же единица (справа), которая появится в выходной комбинации, автоматически передается влево, на входы логических элементов старших разрядов, а это приводит к тому, что для всех оставшихся битов числа на выходе будут сформированы их дополнения.

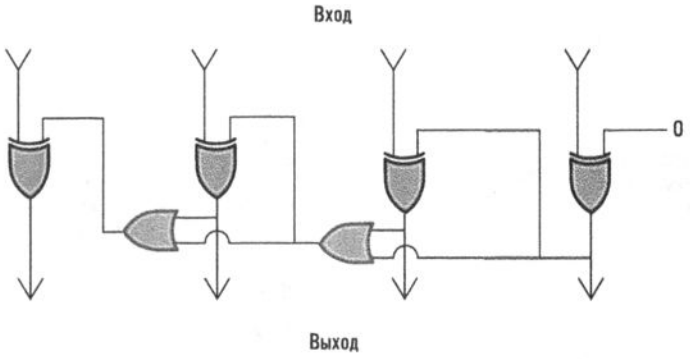


Рис. Б.1. Электронная схема, изменяющая знак числа в двоичном дополнительном коде на противоположный

Далее рассмотрим процесс сложения двух чисел, представленных в двоичном дополнительном коде. Например, рассмотрим решение следующей задачи:

```
  0110
+ 1011
```

Сложение выполняется суммированием отдельных разрядов "по столбцам" в направлении справа налево с использованием одного и того же алгоритма для каждого столбца. Таким образом, если построить электронную схему для сложения значений в одном столбце, то схему для сложения чисел из нескольких столбцов (двоичных разрядов) можно создать, просто копируя в необходимом количестве схему, выполняющую суммирование для одного столбца.

Алгоритм сложения значений в отдельном столбце для задачи сложения чисел из нескольких столбцов состоит в следующем. Необходимо сложить два значения в текущем столбце и добавить эту сумму к значению бита, перенесенному из предыдущего столбца, после чего записать младший значащий бит новой суммы в бит результата и перенести значение избыточного бита в следующий столбец. Электронная схема, реализующая этот алгоритм, представлена на рис. Б.2. На этой схеме верхний логический элемент XOR определяет сумму входных битов, нижний элемент XOR складывает полученную сумму со значением, перенесенным из предыдущего столбца. Два логических элемента AND (логическое "И") вместе с логическим элементом OR передают бит переноса влево. В результате, если в данном столбце оба суммируемых бита равны `1` либо сумма входных битов и бит переноса одновременно равны `1`, в соседний разряд будет перенесено значение `1`.

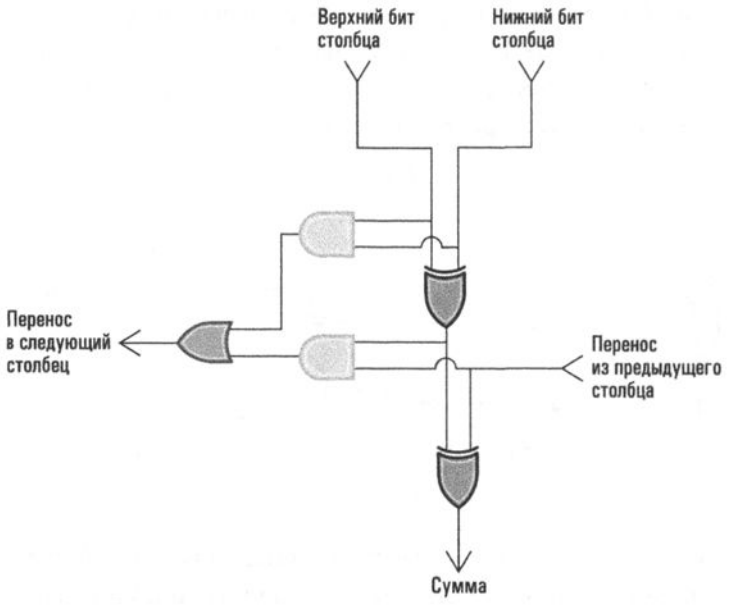


Рис. Б.2. Электронная схема для сложения двоичных значений в отдельном столбце

На рис. Б.3 показано, как несколько копий данной схемы суммирования значений в одном столбце можно использовать для построения электронной схемы, вычисляющей сумму двух чисел, представленных в четырехразрядном двоичном дополнительном коде. На этой схеме каждый прямоугольник представляет собой копию рассмотренной выше электронной схемы суммирования одного разряда. Обратите внимание, что значение бита переноса, поступающее на вход крайнего справа прямоугольника, всегда равно `0`, поскольку для этого разряда никакого переноса из предыдущего столбца не существует. Кроме того, бит переноса из крайнего слева прямоугольника просто игнорируется.

Схема на рис. Б.3 называется *сумматором со сквозным переносом*, поскольку перенос должен проходить сквозь всю схему, от крайнего справа до крайнего слева столбца. Несмотря на простоту реализации такие схемы медленнее выполняют свои функции по сравнению с более совершенными схемами, такими как сумматор с ускоренным переносом, минимизирующий переносы от столбца к столбцу. Поэтому схема, изображенная на рис. Б.3, хотя и подходит для наших целей, в современных вычислительных машинах не используется.

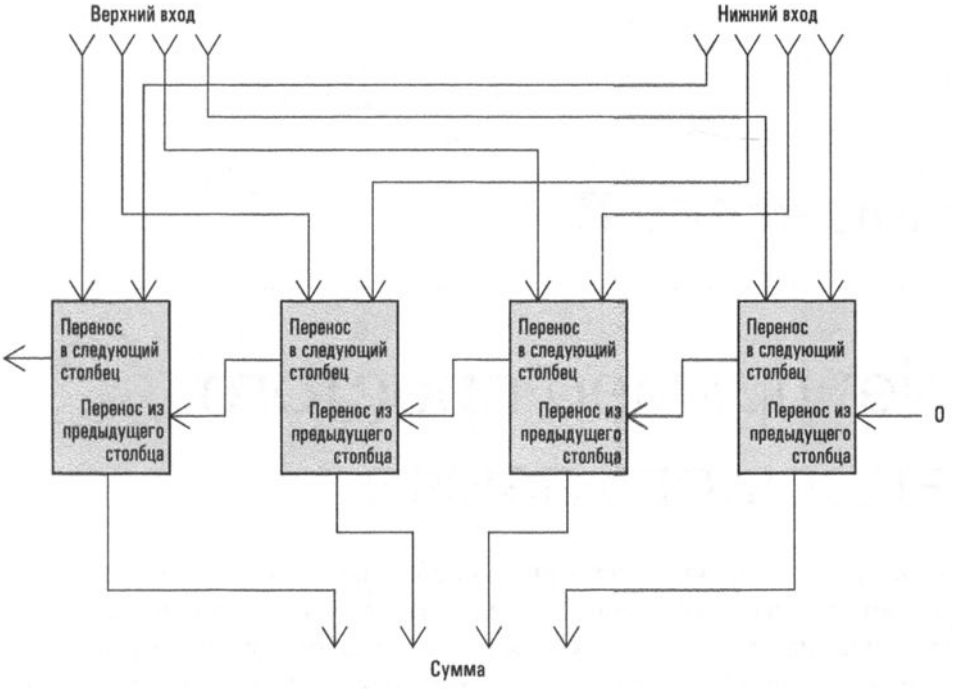


Рис. Б.3. Электронная схема сложения двух четырехразрядных чисел в двоичном дополнительном коде, построенная из четырех копий схемы, представленной на рис. Б.2

Приложение **B**

Vole: пример простого машинного языка

В этом приложении мы рассмотрим простой, но вполне представительный машинный язык, получивший свое название от простейшей гипотетической машины, на которой он мог бы выполняться. Вполне логично будет начать обсуждение именно с рассмотрения архитектуры этой самой гипотетической машины.

Архитектура машины Vole[1]

Рассматриваемая гипотетическая машина Vole имеет 16 регистров общего назначения, пронумерованных от `0x0` до `0xF` (в шестнадцатеричной системе счисления). Длина каждого регистра равна одному байту (восьми битам). Для идентификации регистров в машинных командах каждому регистру присвоен уникальный четырехбитовый код, который представляет собой номер этого регистра. Таким образом, регистр `0x0` идентифицируется как `0000` (шестнадцатеричный 0), а регистр `0x4` — как `0100` (шестнадцатеричное 4).

Память рассматриваемой машины состоит из 256 ячеек, и каждая ячейка имеет уникальный адрес, представляющий собой целое число в диапазоне от 0 до 255. Таким образом, адрес любой ячейки памяти может быть представлен восьмибитовыми двоичными числами от `00000000` до `11111111` (в шестнадцатеричном представлении — от `0x0` до `0xFF`).

[1] Словом "Vole" в английском языке обозначают семейство маленьких шустрых грызунов, широко распространенных по всему миру (мыши-полевки). Как и описываемый в этом приложении процессор, они невелики по размеру, но весьма эффективны.

Предполагается, что числа с плавающей запятой хранятся в памяти этой машины в восьмибитовом формате, подробно обсуждавшемся в разделе 1.7 и наглядно представленном на рис. 1.24.

Машинный язык Vole

Длина каждой команды машины Vole равна двум байтам. Первые четыре бита содержат код операции, а последующие 12 бит образуют поле операндов. В приведенной ниже таблице перечислены и кратко описаны команды, коды которых даны в шестнадцатеричном представлении. Буквы "R", "S" и "T" используются для указания в поле операндов позиции шестнадцатеричных цифр, представляющих идентификаторы регистров, которые могут принимать те или иные значения в зависимости от конкретного случая использования команды. В противоположность этому буквы "X" и "Y" используются для указания в поле операндов позиций тех шестнадцатеричных цифр, которые не являются идентификаторами регистров.

Код операции	Операнд	Описание
`0x1`	RXY	Команда `LOAD` — загрузка в регистр R двоичного кода, который в данный момент находится в ячейке памяти с адресом XY. *Пример*: команда `0x14A3` помещает в регистр `0x4` текущее содержимое ячейки памяти с адресом `0xA3`
`0x2`	RXY	Еще одна команда `LOAD` — помещение в регистр R двоичного кода, записанного в полях XY самой этой команды. *Пример*: команда `0x20A3` помещает в регистр `0x0` шестнадцатеричное значение `0xA3`
`0x3`	RXY	Команда `STORE` — записывает текущее содержимое регистра R в ячейку памяти с адресом XY. *Пример*: команда `0x35B1` помещает содержимое регистра `0x5` в ячейку памяти с адресом `0xB1`
`0x4`	`0`RS	Команда `MOVE` — переписывает текущее содержимое регистра R в регистр S. *Пример*: команда `0x40A4` копирует содержимое регистра `0xA` в регистр `0x4`

Продолжение таблицы

Код операции	Операнд	Описание
`0x5`	RST	Команда `ADD` — текущее содержимое регистров S и T суммируется в предположении, что это два числа в двоичном дополнительном коде. Полученный результат помещается в регистр R. *Пример*: команда `0x5726` суммирует содержимое регистров `0x2` и `0x6` как двоичных чисел в дополнительном коде. Полученная сумма помещается в регистр `0x7`
`0x6`	RST	Еще одна команда `ADD` — текущее содержимое регистров S и T суммируется в предположении, что это два числа в формате с плавающей запятой. Полученный результат помещается в регистр R также в формате с плавающей запятой. *Пример*: команда `0x634E` суммирует содержимое регистров `0x4` и `0xE` как двоичных чисел в формате с плавающей запятой. Полученная сумма помещается в регистр `0x3`
`0x7`	RST	Команда `OR` — выполняет поразрядную операцию OR над текущим содержимым регистров S и T. Полученный результат помещается в регистр R. *Пример*: команда `0x7CB4` помещает в регистр `0xC` результат выполнения поразрядной операции OR над содержимым регистров `0xB` и `0x4`
`0x8`	RST	Команда `AND` — выполняет поразрядную операцию AND над текущим содержимым регистров S и T. Полученный результат помещается в регистр R. *Пример*: команда `0x8045` помещает в регистр `0x0` результат выполнения поразрядной операции AND над содержимым регистров `0x4` и `0x5`
`0x9`	RST	Команда `XOR` — выполняет поразрядную операцию XOR над текущим содержимым регистров S и T. Полученный результат помещается в регистр R. *Пример*: команда `0x95F3` помещает в регистр `0x5` результат выполнения поразрядной операции XOR над содержимым регистров `0xF` и `0x3`

Окончание таблицы

Код операции	Операнд	Описание
`0xA`	R0X	Команда `ROTATE` — выполняет операцию циклического поразрядного сдвига вправо на X позиций текущего содержимого регистра R. При каждом одиночном сдвиге бит из младшего разряда регистра перемещается в его старший разряд. *Пример*: команда `0xA403` выполняет циклический поразрядный сдвиг вправо на 3 бит содержимого регистра `0x4`
`0xB`	RXY	Команда `JUMP` — передает управление команде, хранящейся в ячейке памяти с адресом XY, но только в том случае, если содержимое регистра R в точности совпадает с содержимым регистра 0. В противном случае сохраняется обычный порядок выполнения команд, т.е. следующей для выполнения будет выбрана команда, хранящаяся в ячейке памяти, расположенной сразу после ячейки с данной командой. (Передача управления осуществляется копированием на этапе выполнения команды содержимого, записанного в полях XY самой команды, в счетчик адреса процессора.) *Пример*: команда `0xB43C` осуществляет сравнение содержимого регистра `0x4` с содержимым регистра `0x0`. Если их содержимое одинаково, в счетчик адреса помещается двоичный код `0x3C`, после чего выполнение команды завершается, а следующей будет выполнена команда, выбранная из ячейки памяти с этим адресом. В противном случае непосредственно после сравнения команда завершается и выполнение программы продолжается с сохранением обычного порядка выполнения команд
`0xC`	`000`	Команда `HALT` — завершает выполнение программы. *Пример*: команда `0xC000` прекращает выполнение текущей программы

Приложение Г

Высокоуровневые языки программирования

Это приложение содержит краткий обзор каждого из языков программирования, обсуждавшихся в качестве примеров в главе 6.

Язык Ada

Язык программирования Ada, названный в честь Августы Ады Байрон (Augusta Ada Byron) (1815–1851), в замужестве — баронессы Лавлейс, помощницы Чарльза Бэббиджа (Charles Babbege) и дочери поэта лорда Байрона, был создан по инициативе Министерства обороны США. Военные хотели получить единый язык общего назначения, который можно было бы использовать во всех разработках программного обеспечения, проводимых в этом министерстве. Основной акцент при разработке языка Ada был сделан на средствах программирования компьютерных систем реального времени, которые являются частью более крупных систем, таких как системы управления полетами ракет, системы контроля состояния среды в зданиях и сооружениях, управляющие системы в автомобилях и даже небольшие домашние системы управления. В результате в язык Ada были включены возможности программирования параллельных процессов, а также удобные средства для обработки особых случаев (называемых исключительными ситуациями), которые могут возникать при работе систем. Хотя изначально этот язык разрабатывался как императивный, в стандарте 1995 года в него были добавлены базовые средства объектно-ориентированного программирования, а в стандарте 2007 года эти средства были дополнены и расширены, благодаря чему в настоящее время Ada — полноценный объектно-ориентированный язык программирования.

Структура языка Ada включает целый набор важнейших функциональных особенностей, которые обеспечивают возможность эффективной разработки надежного программного обеспечения, что наглядно демонстрирует тот факт, что на самолете Boeing 777 все программное обеспечение внутреннего контроля и управления написано на языке Ada. Эта особенность также послужила основной причиной, по которой язык Ada был использован в качестве отправной точки при разработке языка SPARK, как уже упоминалось в главе 5.

Язык C

Язык C был разработан в начале 1970-х годов Деннисом Ритчи (Dennis Ritchie), работавшим в то время в компании Bell Laboratories. Хотя первоначально язык C создавался для разработки операционных систем и компиляторов, он быстро получил популярность в среде программистов и приобрел дополнительные преимущества благодаря его стандартизации, выполненной Американским институтом национальных стандартов (ANSI — American National Standards Institute).

Язык C сначала рассматривался просто как некоторый шаг вперед по сравнению с машинным языком. По этой причине его синтаксис более краток и выразителен, чем синтаксис других языков высокого уровня, использующих полные слова английского языка для выражения тех языковых конструкций, которые в языке C представляются с помощью специальных символов. Эта лаконичность является одной из причин чрезвычайной популярности языка C, поскольку позволяет программистам эффективно выражать сложные алгоритмы. (Часто краткое представление алгоритма более доступно пониманию, чем его пространное описание.)

Язык C++

Язык C++ был разработан Бьярне Страуструпом (Bjarne Straustrup) из компании Bell Laboratories как усовершенствованная версия языка C. Цель создания языка C++ — достижение совместимости языка C с объектно-ориентированной парадигмой программирования. В настоящее время язык C++ является не только самостоятельным широко известным объектно-ориентированным языком программирования; он также был использован как отправная точка при разработке двух других ведущих объектно-ориентированных языков: Java и C#.

Язык C#

Язык C# был разработан корпорацией Microsoft в качестве основного инструмента для создания ПО на платформе .NET, которая представляет собой всеобъемлющую комплексную систему, предназначенную для разработки прикладного программного обеспечения на компьютерах, работающих под управлением системного программного обеспечения от Microsoft. Язык C# очень похож на C++ и Java. И действительно, причиной, по которой корпорация Microsoft представила C# как самостоятельный язык программирования, был не тот факт, что он представлял собой действительно нечто новое в смысле собственно языка, а то, что, представив его как иной, отличающийся язык, Microsoft получила возможность добавлять в него или изменять по своему усмотрению его функциональные возможности, не принимая при этом во внимание те стандарты, которые уже прочно ассоциировались с другими языками либо были необходимы по соображениям соблюдения авторских прав других корпораций. Вот почему новизна языка C# заключается, прежде всего, в его особой роли как базового, ведущего языка программирования для разработки программного обеспечения для платформы .NET. При поддержке со стороны Microsoft язык C# и платформу .NET можно уверенно считать важными компонентами в мире разработки программного обеспечения на многие годы вперед.

Язык FORTRAN

FORTRAN (сокращение от "FORmula TRANslator" — "транслятор формул") был одним из первых языков программирования высокого уровня (создан в период с 1954 по 1957 год группой программистов в корпорации IBM) и стал первым языком, получившим широкое признание в компьютерном сообществе. С течением времени его официальное описание претерпело многочисленные изменения, так что теперь современный FORTRAN очень сильно отличается от его первой версии. И действительно, изучая путь развития, который FORTRAN прошел за все эти годы, можно с интересом наблюдать, какое влияние оказывали на него очередные достижения в области разработки языков программирования. Хотя исходный вариант языка был реализован как императивный, новые версии уже включают многие функции объектно-ориентированного и модульного программирования. В научной среде FORTRAN по-прежнему остается одним из самых популярных языков программирования. В частности, многие средства численного анализа и статистические пакеты были созданы и, вероятно, будут и в дальнейшем создаваться на языке FORTRAN.

Язык Java

Java — это объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems в начале 1990-х годов. Его разработчики много позаимствовали из языков C и C++. Сейчас многие восторженно относятся к этому языку, но не по причине его собственно языковых особенностей, а вследствие универсальности его реализации и широкого разнообразия предварительно спроектированных шаблонов, которые доступны разработчикам в среде программирования Java. Универсальность реализации означает, что программа, написанная на языке Java, может успешно функциолнировать на широком диапазоне различных машин, а доступность шаблонов обеспечивает относительную простоту разработки сложных программных комплексов. Например, такие шаблоны, как аплеты и сервлеты, существенно упрощают разработку программного обеспечения для World Wide Web.

Приложение Д

Эквивалентность итеративных и рекурсивных структур

В этом приложении мы будем использовать наш простейший язык, описанный в главе 12, в качестве инструмента для поиска ответа на вопрос об относительной мощности итеративных и рекурсивных структур, поставленный в главе 5. Напомним, что наш простейший язык содержит только три оператора присваивания (`clear`, `incr` и `decr`) и одну управляющую структуру (построенную из оператора `while`). Данный язык имеет ту же вычислительную мощность, что и машина Тьюринга, следовательно, если мы принимаем тезис Черча–Тьюринга, то должны прийти к заключению, что решение любой задачи, теоретически имеющей алгоритмическое решение, можно выразить с помощью нашего простейшего языка.

Первый шаг при сравнении мощности итеративных и рекурсивных структур заключается в замене итеративных структур простейшего языка рекурсивными. Мы удалим из нашего языка оператор `while`, а вместо него наделим наш модифицированный простейший язык поддержкой разделения программы на модули и средствами вызова одного модуля из другого, входящего в состав той же программы. Точнее говоря, мы предположим, что каждая программа на модифицированном языке будет состоять из нескольких синтаксически раздельных программных модулей. Кроме того, предположим, что каждой программе разрешается содержать только один модуль `MAIN`, имеющий следующую синтаксическую структуру:

```
def MAIN():
```

(Здесь точки обозначают наличие других операторов нашего простейшего языка.) Дополнительно предполагается, что все другие модули (семантически подчиненные модулю MAIN) должны иметь следующую структуру:

```
def unit():
  .
  .
  .
```

(Здесь параметр unit представляет имя модуля, которое синтаксически не отличается от имен переменных.) Семантика этой модульной системы заключается в том, что программа всегда начинает свое выполнение с модуля MAIN и прекращает работу, когда в этом модуле достигнут последний выполнимый оператор. Программные модули, отличные от MAIN, могут вызываться как процедуры с помощью следующего условного оператора:

```
if name not 0:
  unit()
```

(Здесь параметр name может быть именем любой переменной, а параметр unit представлять имя любого модуля в программе, кроме модуля MAIN.) Более того, мы позволим любому модулю, кроме MAIN, рекурсивно вызывать самого себя.

Благодаря наличию в модифицированном языке указанных дополнительных возможностей можно смоделировать исключенную из него структуру while, присутствовавшую в исходном простейшем языке. Например, рассмотрим программу на простейшем языке следующего вида:

```
while X not 0:
  S
```

(Здесь S обозначает любую последовательность операторов простейшего языка.) Эта программа может быть заменена следующей модульной структурой:

```
def MAIN():
  if X not 0:
    unitA()
def unitA():
  S
  if X not 0:
    unitA()
```

Следовательно, можно сделать вывод, что модифицированная версия языка имеет все те же возможности, что и исходная версия простейшего языка.

Можно также показать, что любая задача, которая может быть решена с помощью модифицированного языка, может быть решена и с помощью исходного простейшего языка. Для того чтобы показать это, докажем, что любой алгоритм, выраженный на модифицированном языке, может быть записан и на

простейшем языке. Однако для этого необходимо явно определить способ, которым рекурсивная структура новой версии языка может быть смоделирована с помощью структуры `while` простейшего языка.

Для наших целей проще сослаться на тезис Черча–Тьюринга, описанный в главе 12. В частности, из тезиса Черча–Тьюринга и того факта, что простейший язык имеет ту же мощность, что и машина Тьюринга, следует, что ни один язык не может иметь мощность, превышающую мощность исходного простейшего языка. Таким образом, мы можем сделать заключение, что любая проблема, допускающая решение средствами модифицированного языка, может быть также решена средствами исходного простейшего языка.

Теперь мы можем обоснованно утверждать, что мощности модифицированного и исходного простейшего языков равны между собой. Единственное различие состоит в том, что один реализует итеративные управляющие структуры, а другой — рекурсивные. Таким образом, мы приходим к выводу, что в смысле вычислительной мощности эти две управляющие структуры фактически эквивалентны.

Приложение **E**

Ответы к разделам "Вопросы и упражнения"

Глава 1

Раздел 1.1

1. Значение `1` должно быть на одном и только на одном из двух верхних входов, а значение на нижнем входе также должно быть равно 1.
2. Единица на нижнем входе вызывает появление значения `0` на выходе логического элемента `NOT`. Это приводит к тому, что на выходе логического элемента `AND` появляется `0`. В результате на оба входа логического элемента `OR` подается значение `0` (не забывайте, что на верхнем входе триггера сохраняется входное значение `0`), так что на выходе логического элемента `OR` устанавливается `0`. А это означает, что на выходе логического элемента `AND` значение `0` сохранится и после того, как на нижний вход триггера вновь будет подано значение `0`.
3. На выходе верхнего логического элемента `OR` появится значение `1`, а это приведет к тому, что на выходе верхнего логического элемента `NOT` появится `0`. В результате на выходе нижнего логического элемента `OR` появится значение `0`, которое будет преобразовано в `1` на выходе логического элемента `NOT`. Это единичное значение будет представлять собой выходное значение триггера и одновременно сигнал обратной связи, подаваемый на верхний логический элемент `OR`. В результате на его выходе значение `1` будет сохраняться и после того, как на оба входа триггера вновь будет подано значение `0`.

4. **а.** Вся представленная на рисунке схема эквивалентна единственному вентилю OR.

   б. Вся представленная на рисунке схема эквивалентна единственному вентилю XOR.

5. **а.** 0x6AF2 **б.** 0xE85517 **в.** 0x48

6. **а.** 01011111110110010111 **б.** 0110000100001010

   в. 1010101111001101 **г.** 0000000100000000

Раздел 1.2

1. В первом случае после завершения операции ячейка памяти с адресом 6 будет содержать значение 5. Во втором случае в этой ячейке будет находиться значение 8.
2. В результате выполнения первой же операции исходное значение в ячейке с адресом 3 будет утеряно, поскольку поверх него будет записано новое значение. Следовательно, после выполнения второй операции исходное значение из ячейки с адресом 3 не будет перенесено в ячейку с адресом 2 и обе ячейки будут содержать значение, которое исходно хранилось в ячейке с адресом 2.
3. Правильная процедура должна быть следующей.

   Этап 1. Переместить содержимое ячейки с адресом 2 в ячейку с адресом 1.
   Этап 2. Переместить содержимое ячейки с адресом 3 в ячейку с адресом 2.
   Этап 3. Переместить содержимое ячейки с адресом 1 в ячейку с адресом 3.
4. 32 768 бит.

Раздел 1.3

1. Более высокая скорость считывания и передачи данных.
2. Здесь следует вспомнить, что механическое движение намного медленнее электронных процессов, протекающих в компьютере. Это вынуждает минимизировать количество перемещений магнитных головок. Если полностью заполнять поверхность данными перед тем, как перейти к следующей поверхности, придется перемещать магнитную головку каждый раз, когда очередная дорожка будет заполнена. В результате количество перемещений головок будет приблизительно равно общему количеству дорожек на обеих поверхностях. Однако если переходить с одной поверхности на другую путем электронного переключения между магнитными

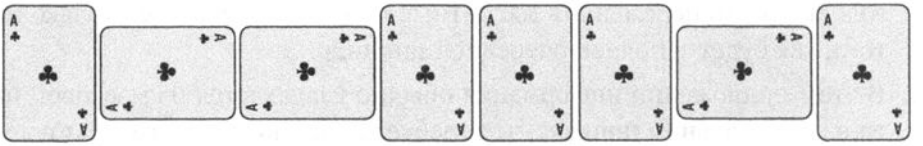
головками, то перемещать магнитные головки потребуется только после того, как будет заполнен очередной цилиндр.

3. В этом приложении информация обычно извлекается из массовой памяти в произвольном порядке, что требует дополнительных затрат времени с учетом наличия единственной спиральной дорожки с данными на компакт- или DVD-диске.
4. Компакт-, DVD- и BD-диски имеют один и тот же физический размер, и информация на всех них размещается в единообразной спиральной дорожке, записываемой на носитель. Достаточно оборудовать устройство лазерами нескольких типов, и это позволит ему считывать данные со всех трех типов оптических дисков.
5. Устройства флеш-памяти не нуждаются в физическом перемещении своих элементов, поэтому они имеют меньшее время ответа и не страдают от физического износа.
6. Магнитные жесткие диски работают быстрее и имеют большую емкость, чем прочие виды магнитных носителей, таких как гибкие диски и магнитная лента. Быстродействие, емкость и возможность многократной перезаписи делают их предпочтительнее оптических устройств. Стоимость единицы хранимой информации для магнитных дисков по-прежнему ниже чем для твердотельных устройств, хотя в последние годы это преимущество постоянно сокращается.

Раздел 1.4

1. Computer Science (*Компьютерные науки*).
2. Две эти битовые комбинации практически одинаковы, за исключением того, что шестой бит, считая от младших разрядов к старшим, всегда равен `0` для прописных букв и `1` — для строчных.
3. **а.** `00100010 01010011 01110100 01101111 01110000 00100001 00100010 00100000 01000011 01101000 01100101 01110010 01111001 01101100 00100000 01110011 01101000 01101111 01110101 01110100 01100101 01110100 00101110`

   б. `01000100 01101111 01100101 01110011 00100000 00110010 00100000 00101011 00100000 00110011 00100000 00111101 00100000 00110101 00111111`

4. 

5. **а.** 5 **б.** 9 **в.** 11 **г.** 6 **д.** 16 **е.** 18

6. **а.** 110 **б.** 1101 **в.** 1011 **г.** 10010 **д.** 11011 **е.** 100

7. В 24 битах можно хранить три символа в кодировке ACSII, т.е. числа от 0 до 999. Однако если использовать эти биты как разряды двоичного числа, то в них можно будет хранить целые числа, вплоть до 16 777 215.

8. **а.** 15.15 **б.** 51.0.128 **в.** 10.160

9. Геометрическое представление предоставляет большие возможности в отношении масштабирования как всего изображения в целом, так и отдельных его элементов. Однако геометрическое представление не позволяет достичь такой фотографической точности сложных объектов, как это возможно в растровых методах. На практике, как это было показано в разделе 1 .9, растровый формат JPEG наиболее популярен в цифровой фотографии.

10. При частоте дискретизации 44 100 Гц для стереозаписи одного часа музыки потребуется объем памяти 635 040 000 байт. А это означает, что она практически полностью заполнит компакт-диск, емкость которого составляет чуть больше 600 Мбайт.

Раздел 1.5

1. **а.** 42 **б.** 33 **в.** 23 **г.** 6 **д.** 31
2. **а.** 100000 **б.** 1000000 **в.** 1100000 **г.** 1111 **д.** 11011
3. **а.** $3\frac{1}{4}$ **б.** $5\frac{7}{8}$ **в.** $2\frac{1}{2}$ **г.** $6\frac{3}{8}$ **д.** $\frac{5}{8}$
4. **а.** 100.1 **б.** 10.11 **в.** 1.001 **г.** 0.0101 **д.** 101.101
5. **а.** 100111 **б.** 1011.110 **в.** 100000 **г.** 1000.00

Раздел 1.6

1. **а.** 3 **б.** 15 **в.** -4 **г.** -6 **д.** 0 **е.** -16
2. **а.** 00000110 **б.** 11111010 **в.** 11101111
   г. 00001101 **д.** 11111111 **е.** 00000000
3. **а.** 11111111 **б.** 10101011 **в.** 00000100
   г. 00000010 **д.** 00000000 **е.** 10000001

4. а. При 4 битах наибольшее число равно 7, а наименьшее — -8.

б. При 6 битах наибольшее число равно 31, а наименьшее — -32.

в. При 8 битах наибольшее число равно 127, а наименьшее — -128.

5. а. 0111 (5 + 2 = 7) **б.** 0100 (3 + 1 = 4)

в. 1111 (5 + (–6) = –1) **г.** 0001 (–2 + 3 = 1)

д. 1000 (–6 + (–2) = –8)

6. а. 0111 **б.** 1011 (переполнение) **в.** 0100 (переполнение)

г. 0001 **д.** 1000 (переполнение)

7.

а.	**б.**	**в.**	**г.**	**д.**
0110	0011	0100	0010	0001
+ 0001	+ 1110	+ 1010	+ 0100	+ 1011
0111	0001	1110	0110	1100

8. Нет. Переполнение возникнет при попытке записать в память число, которое слишком велико для используемой системы. При добавлении положительного числа к отрицательному результат будет равен числу, не превышающему по модулю каждое из этих слагаемых. Таким образом, если исходные числа достаточно малы, чтобы храниться в памяти, то и результат также поместится в памяти.

9. а. 6, поскольку 1110 → 14 – 8

б. -1, поскольку 0111 → 7 – 8

в. 0, поскольку 1000 → 8 – 8

г. -6, поскольку 0010 → 2 – 8

д. -8, поскольку 0000 → 0 – 8

е. 1, поскольку 1001 → 9 – 8

10. а. 1101, поскольку 5 + 8 = 13 → 1101

б. 0011, поскольку –5 + 8 = 3 → 0011

в. 1011, поскольку 3 + 8 = 11 → 1011

г. 1000, поскольку 0 + 8 = 8 → 1000

д. 1111, поскольку 7 + 8 = 15 → 1111

е. 0000, поскольку –8 + 8 = 0 → 0000

11. Нет. Наибольшее число, которое можно представить в двоичной нотации с избытком 8, равно 7, т.е. 1111. Чтобы представить большее число (которое использует 5 битов), нужно выбрать основание, равное как минимум 16. Аналогично число 6 не может быть выражено в двоичной нотации с избытком 4. (Наибольшее число, которое может быть представлено в этом коде, равно 3.)

Раздел 1.7

1. **а.** $5/8$ **б.** $3\frac{1}{4}$ **в.** $9/32$ **г.** $-1\frac{1}{2}$ **д.** $-(11/64)$
2. **а.** `01101011` **б.** `01111010` (ошибка округления)
   в. `01001100` **г.** `11101110` **д.** `11111000` (ошибка округления)
3. `01001001` ($9/16$) больше, чем `00111101` ($13/32$). Ниже приведен простой способ определения, какой из двоичных кодов представляет собой большее число.

   Случай 1. Если знаковые биты чисел разные, из двух чисел больше то, знаковый бит которого равен `0`.

   Случай 2. Если оба знаковых бита чисел равны `0`, нужно просмотреть оставшуюся часть кода слева направо, пока не встретится битовая позиция, в которой числа отличаются друг от друга. Двоичный код в этой позиции которого находится `1`, представляет собой большее число.

   Случай 3. Если знаковые биты обоих чисел равны `1`, необходимо просмотреть оставшуюся часть кода слева направо, пока не встретится битовая позиция, в которой числа отличаются друг от друга. Двоичный код, в этой позиции которого стоит `0`, представляет собой большее число. Простота этого способа сравнения двух чисел является одной из причин, по которым экспоненты чисел с плавающей точкой представляются в двоичной нотации с избытком, а не в дополнительном коде.
4. Большим значением было бы $7\frac{1}{2}$, которое в двоичной системе счисления имеет вид `01111111`. Что касается наименьшего положительного значения, то можно было бы сказать, что существуют "два" правильных ответа. Если придерживаться описанного в тексте процесса кодирования, требующего, чтобы самый старший значащий бит мантиссы был равен `1` (нормализованная форма), то в этом случае ответом является число $1/32$, которое в двоичной системе счисления имеет вид `00001000`. Однако большинство машин не накладывает ограничений на значения, близкие к нулю. Для таких машин правильным ответом будет число $1/256$, которое в двоичной системе счисления будет равно `00000001`.

Раздел 1.8

1. Язык Python рассматривается как *интерпретирующий язык*, поскольку пользователи в интерактивном режиме могут вводить фрагменты программы непосредственно в строку ввода, вместо того чтобы предварительно поместить текст всей программы в файл, запустить программу-компилятор, а затем передать ей этот файл на обработку.

2. **а.** `print('Computer Science Rocks' + '!')`

   б. `print(42)`

   в. `print(3.1416)`

3. **а.** `rockstar = 'programmer'`

   б. `seconds_per_hour = 60 * 60`

   или

   `seconds_per_hour = 3600`

   в. `bodyTemp = 36.7`

4. `metricBodyTemp = (bodyTemp - 32) / 1.8`

Раздел 1.9

1. Кодирование длины серий, частотно-зависимое кодирование, относительное кодирование и словарное кодирование.
2. `121321112343535`
3. Цветные мультфильмы состоят из блоков сплошного цвета с резкими контурами. Кроме того, количество используемых в них цветов обычно ограничено.
4. Нет. Как в стандарте GIF, так и в стандарте JPEG в режиме "базовых строк" используются методы сжатия с потерями, а это означает, что некоторые детали на изображениях будут утрачены.
5. Стандарт JPEG в режиме "базовых строк" извлекает преимущества из того факта, что человеческий глаз не так чувствителен к изменению цвета изображения, как к изменению его яркости. По этой причине в формате JPEG сокращено количество битов для представления информации о цвете без ощутимой потери качества изображения.
6. Временн*ая* маскировка и частотная маскировка.
7. При кодировании информации часто имеет место лишь приблизительное представление точных значений данных. В случае числовых данных при выполнении вычислений имеющиеся неточности от использования приблизительных значений постепенно накапливаются, что может привести к получению ошибочных результатов. Приблизительность значений не столь критична в случае изображений или звука, поскольку закодированные данные обычно лишь хранятся, передаются или репродуцируются. Однако, если изображение или аудиозапись многократно копируется, перезаписывается, а затем перекодируется, имеющиеся неточности приближения могут накапливаться, что в конечном счете приведет к снижению качества данных.

Раздел 1.10

1. Пп. *б*, *в* и *д*.
2. Да. Если в одном байте окажется четное количество ошибок, то проверка четности не позволит их обнаружить.
3. В этом случае ошибки возникают в байтах *а* и *г* из вопроса 1. Ответ на вопрос 2 тот же.
4. **а.**
```
00100010  01000011  01101000
01010011  01101000  01101111
01110100  01100101  01110101
01101111  01110010  01110100
01110000  01111001  01100101
00100001  01101100  01100100
00100010  00100000  00101110
00100000  01110011
```
б.
```
01000100  00110010  00100000
01101111  00100000  00111101
01100101  00101011  00100000
01110011  00100000  00110101
00100000  00110011  00111111
```
5. **а.** BED **б.** CAB **в.** HEAD
6. Одно из решений таково:
```
A  00000
B  11100
C  01111
D  10011
```

Глава 2

Раздел 2.1

1. На некоторых машинах это действие представляет собой процесс, состоящий из двух этапов: сначала содержимое первой ячейки копируется в регистр, а затем записывается в предназначенную для него ячейку памяти. Однако в большинстве машин этот процесс выполняется без использования промежуточных регистров.
2. Значение, которое должно быть записано; адрес ячейки, в которую должны быть записаны значение и команда записи.
3. Регистры общего назначения используются для хранения данных, которые используются прямо сейчас, — командами, выполняемыми в данный

момент. Основная память используется для хранения данных, которые будут использоваться в ближайшем будущем. Массовая память используется для хранения данных, которые, вероятнее всего, не потребуются в ближайшем будущем.

Раздел 2.2

1. Термин "переместить" (*move*) обычно означает удаление значения из одного места и запись его в другое, при этом предыдущее место его хранения остается пустым. Однако в большинстве случаев обработки данных в компьютерах объект, подлежащий перемещению, просто копируется (или клонируется) в новое место.
2. Общий прием, называемый **относительной адресацией**, заключается в указании, *насколько далеко*, а не *куда именно* переходить, например перейти на три команды вперед или на две команды назад. Следует заметить, что команды, использующие относительную адресацию, потребуется изменить, если между исходной командой перехода и командой, на которую следует перейти, позднее будут вставлены дополнительные команды.
3. Ответ на этот вопрос можно обосновать по-разному. Команда записана в форме условного перехода. Однако, поскольку поставленное условие, что 0 должен быть равен 0, выполняется всегда, переход в этом случае будет выполняться так, как если бы никакого условия вовсе не было. Машины с такими командами встречаются часто, поскольку эти команды очень эффективны. Например, если машина разработана для выполнения команд, имеющих структуру вида "Если..., то перейти на ...", такую команду можно использовать как для условного, так и безусловного переходов.
4. `0x156C = 0001010101101100`
   `0x166D = 0001011001101101`
   `0x5056 = 0101000001010110`
   `0x306E = 0011000001101110`
   `0xC000 = 1100000000000000`
5. **а.** Сохранить (`STORE`) содержимое регистра `0x6` в ячейке памяти с адресом `0x8A`.

   б. Перейти (`JUMP`) на команду в ячейке с адресом `0xDE`, если содержимое регистра `0xA` равно содержимому регистра `0x0`.

   в. Выполнить поразрядную операцию И (`AND`) над содержимым регистров `0x3` и `0xC`, поместив результат в регистр `0x0`.

   г. Переместить (`MOVE`) содержимое регистра `0xF` в регистр `0x4`.

6. Команда `0x15AB` требует, чтобы центральный процессор запросил у схемы управления основной памятью содержимое ячейки с адресом `0xAB`. Извлеченное из памяти значение помещается в регистр `0x5`. Команда `0x25AB` не предусматривает такого запроса к памяти. Точнее говоря, в регистр `0x5` просто помещается значение `0xAB`.
7. **а.** `0x2356` **б.** `0xA503` **в.** `0x80A5`

Раздел 2.3

1. Значение `0x34` (шестнадцатеричное).
2. **а.** `0x0F` **б.** `0xC3`
3. **а.** `0x00` **б.** `0x01` **в.** Четыре раза.
4. Машина прекращает работу. Это пример того, что принято называть *самоизменяющимся* кодом, т.е. программа сама изменяет себя. Обратите внимание, что первые две команды помещают значение `0xC0` в ячейку памяти с адресом `0xF8`, а следующие две команды — значение `0x00` в ячейку с адресом `0xF9`. Таким образом, в то время, когда машина выберет команду из ячейки с адресом `0xF8`, в ней уже будет храниться код команды прекращения работы — `0xC000`.

Раздел 2.4

1. **а.** `00001011` **б.** `10000000` **в.** `00101101`
   г. `11101011` **д.** `11101111` **е.** `11111111`
   ж. `11100000` **з.** `01101111` **и.** `11010010`
2. Маска `00111100` с операцией И (AND).
3. Маска `00111100` с операцией Исключающее ИЛИ (XOR).
4. **а.** Окончательный результат равен `0`, если строка содержит четное количество вхождений цифры `1`. В противном случае значение будет равно `1`.
   б. Результат равен значению бита четности при проверке четности.
5. Логическая операция XOR фактически совпадает с операцией сложения, за исключением случая, когда оба операнда равны `1`. В этом случае результат операции XOR равен `0`, в то время как сумма будет равна `10`. Таким образом, операцию XOR можно рассматривать как операцию сложения без переноса разряда переполнения.
6. Для преобразования строчных букв в прописные можно использовать операцию AND с маской `01011111`. Для преобразования прописных букв в строчные — операцию OR с маской `00100000`.

7. **а.** 01001101 **б.** 11100001 **в.** 11101111
8. **а.** 0x57 **б.** 0xB8 **в.** 0x6F **г.** 0x6A
9. 5
10. В двоичном дополнительном коде — 00110110; в виде числа с плавающей точкой — 01011110. Суть здесь в том, что процедуры сложения двух чисел различаются в зависимости от интерпретации заданных двоичных кодов.
11. Одно из решений будет следующим.

    0x12A7 # Загрузка (LOAD) в регистр 0x2 содержимого ячейки памяти с адресом 0xA7.

    0x2380 # Загрузка (LOAD) в регистр 0x3 значения 0x80.

    0x7023 # Операция OR над содержимым регистров 0x2 и 0x2 с помещением результата в регистр 0x0.

    0x30A7 # Запись содержимого (STORE) регистра 0x0 в ячейку памяти с адресом 0xA7.

    0xC000 # Прекращение (HALT) выполнения программы.
12. Одно из решений может быть таким.

    0x15E0 # Загрузка (LOAD) в регистр 0x5 содержимого ячейки памяти с адресом 0xE0.

    0xA502 # (Циклический сдвиг (ROTATE) содержимого регистра 0x5 на два бита вправо.

    0x260F # Загрузка (LOAD) в регистр 0x6 значения 0x0F.

    0x8056 # Выполнение операции AND над содержимым регистров 0x5 и 0x6 с помещением результата в регистр 0x0.

    0x30E1 # Запись (STORE) содержимого регистра 0x0 в ячейку памяти с адресом 0xE1.

    0xC000 # Прекращение (HALT) выполнения программы.

Раздел 2.5

1. **а.** 0x37B5

   б. Миллион раз.

   в. Нет. Обычная страница текста содержит меньше 4000 символов. Таким образом, возможность напечатать 5 страниц в минуту означает, что скорость печати приблизительно равна 20 000 символов в минуту, что значительно меньше, чем один миллион символов в минуту. (Дело в

том, что компьютер может отсылать символы на принтер намного быстрее, чем принтер сможет их напечатать; таким образом, принтер должен иметь возможность попросить компьютер подождать.)

2. Диск будет выполнять 50 оборотов в секунду. Это означает, что за одну секунду под головкой чтения/записи будет проходить 800 секторов. Поскольку каждый сектор вмещает 1024 байта, биты будут проходить под головкой чтения/записи со скоростью приблизительно 6,5 Мбит/с. Таким образом, обмен данными между контроллером и дисководом должен происходить с этой же скоростью, чтобы контроллер мог успевать получать данные, проходящие под головкой диска.
3. Роман в триста страниц, записанный в кодах Unicode , содержит около 2 Мбайт, или 16 000 000 бит информации. Таким образом, для передачи всего романа со скоростью 54 Мбит/с потребуется приблизительно 0,3 секунды.

Раздел 2.6

1. Вызов функции `int()` используется при вводе длин боковых сторон, чтобы преобразовать (обрезать) все значения с плавающей точкой в целочисленные значения, которые затем присваиваются целочисленным переменным. Однако функция `math.sqrt()` возвращает результат в виде числа с плавающей точкой независимо от того, в каком формате были представлены ее входные параметры — как целые числа или числа с плавающей запятой. Чтобы вывести результат в виде целого числа, последнюю строку сценария следует привести к виду

   ```
   print(int(hypotenuse))
   ```

   либо заменить оператор присваивания значения переменной `hypotenuse` следующим:

   ```
   hypotenuse = int(math.sqrt(sideA**2 + sideB**2))
   ```

2. Замените в исходном сценарии оба вызова функции `int()` вызовом функции `float()` — и в результате получится сценарий, работающий только с числами в формате с плавающей точкой.
3. Вот один из вариантов кода на языке Python, который позволит достичь желаемых результатов.

   ```
   print('Ваша скорость бега будет ' + str(mph) + ' миль/ч')
   ...
   print('Общее время пробежки составит ' + str(elapsed_minutes) +
         ' минут, ' + str(elapsed_seconds) + ' секунд')
   ```

4. Вот возможный вариант сценария:

```
number = int(input("Введите целое десятичное число: "))
print("Его двоичное представление: " + str(bin(number)))
```

5. Один из возможных вариантов сценария будет следующим:

```
number = int(input("Введите целое число для шифрования или
дешифрования: "))
number = number ^ 0x55555555
print("Результат будет следующим: " + str(number))
```

6. Неожидаемый вариант введенного значения в таких простых сценариях, как эти, в общем случае приведет к возникновению ошибки либо к неожиданному поведению сценария. В более сложных сценариях вводимые значения могут предварительно проверять на допустимость, прежде чем будет предпринята попытка преобразовать их в целое число, либо осуществляются дополнительные проверки для выявления отрицательных значений, заданных в качестве длины стороны, и т.д.

Раздел 2.7

1. Конвейер может содержать команды `0xB1B0` (выполняемая в данный момент), `0x5002` и, возможно, даже `0xB0AA`. Если значение в регистре `0x1` равно значению в регистре `0x0`, выполняется переход к ячейке с адресом `0xB0`, и работа, затраченная на подготовку команд на конвейере, оказывается бесполезной. Однако при этом не происходит потери времени, поскольку работа, затраченная на эти команды, не требовала дополнительного времени.
2. Если не предпринять меры предосторожности, информация из ячеек с адресами `0xF8` и `0xF9` будет извлечена в качестве команды еще до того, как предыдущая часть программы получит возможность изменить содержимое этих ячеек.
3. **а.** Первым прочитать хранящееся в этой ячейке значение может центральный процессор, который пытается добавить `1` к ее содержимому. Затем это же значение читает другой центральный процессор. (Обратите внимание, что в этом случае оба процессора считали одно и то же значение.) Если первый центральный процессор завершит операцию сложения и запишет результат назад в ячейку до того, как второй процессор завершит операцию вычитания и запишет свой результат, то окончательное значение, хранящееся в ячейке, будет отражать результаты работы только второго процессора.

   б. Центральные процессоры считывают данные из ячейки так же, как это было описано выше, но на этот раз второй центральный процессор

может записать результаты своей работы раньше первого центрального процессора. В результате окончательное значение, записанное в ячейке, будет отражать результаты работы только первого центрального процессора.

Глава 3

Раздел 3.1

1. Традиционным примером является очередь людей, желающих купить билет на некоторое мероприятие. В этом случае может найтись некто, желающий обойти очередь, что может привести к разрушению структуры FIFO.
2. Пп. *б*, *в* и *д*.
3. Встроенные компьютерные системы обычно ориентированы на выполнение конкретных, заранее определенных задач, тогда как ПК являются компьютерами общего назначения. Встроенные системы, как правило, имеют более ограниченные ресурсы в сравнении с ПК того же поколения и часто должны отвечать очень жестким требованиям к их работе при минимальном вмешательстве человека.
4. Разделение времени — это метод, с помощью которого на машине организуется одновременная совместная работа более одного пользователя. Многозадачность — это режим, в котором пользователь одновременно может выполнять на машине сразу несколько различных заданий.

Раздел 3.2

1. *Оболочка*. Осуществляет взаимодействие с внешней средой, окружающей компьютер.

   Менеджер файлов. Координирует использование устройств массовой памяти.

   Драйверы устройств. Обеспечивают взаимодействие системы с периферийными устройствами.

   Менеджер памяти. Координирует использование основной памяти компьютера.

   Планировщик. Координирует выполнение в системе различных процессов.

   Диспетчер. Контролирует распределение времени центрального процессора между различными процессами.

2. Граница между этими понятиями неясна, и различие часто проводится лишь умозрительно. Грубо говоря, утилиты выполняют основные, универсальные задачи, тогда как прикладное программное обеспечение предназначено для решения специфических задач, интересующих отдельных пользователей.
3. Виртуальная память — это воображаемое пространство памяти, которое обеспечивается процессом перемещения (подкачки) данных и программ из памяти на жесткий диск и обратно.
4. При включении машины центральный процессор начинает выполнять программу начальной загрузки, текст которой хранится в ROM. В процессе загрузки центральный процессор копирует программы операционной системы из внешнего запоминающего устройства в некоторую область основной памяти. После завершения копирования он передает управление соответствующей программе операционной системы.

Раздел 3.3

1. Программа — это множество команд. Процесс — это действия, выполняемые в соответствии с этими командами.
2. Центральный процессор завершает текущий машинный цикл, сохраняет состояние текущего процесса и устанавливает в счетчике адреса заранее определенное значение (которое является адресом обработчика прерываний). Таким образом, следующая выполняемая команда — это первая команда в обработчике прерываний.
3. Они могли бы получить более высокий приоритет и, значит, пользоваться определенным предпочтением у диспетчера. Другой вариант — выделять больший промежуток времени для выполнения процессов, имеющих более высокий приоритет.
4. Если каждый процесс будет полностью использовать свой квант времени, машина сможет предоставлять полные кванты почти 20 процессам за одну секунду. Если процессы не будут использовать свои кванты полностью, это значение может быть намного выше, но при этом затраты времени на переключение контекста значительно возрастут и в конечном счете могут оказаться довольно существенными (см. вопрос 5).
5. В целом 5000/5001 часть машинного времени может быть затрачена на собственно выполнение вычислительных процессов. Однако, когда процесс запрашивает выполнение операции ввода-вывода данных, выделенный этому процессу квант времени процессора завершается и процессор приступает к обработке запроса. Таким образом, если каждый процесс

будет выдавать запрос на ввод-вывод уже через одну микросекунду после получения кванта времени, эффективность работы машины может снизиться до 1/2. Иными словами, машина будет затрачивать на выполнение необходимых переключений столько же времени, сколько и на выполнение самих процессов.

Раздел 3.4

1. Эта система гарантирует, что в каждый момент времени ресурс будет использоваться не более чем одним процессом. Однако такой подход требует выполнять распределение ресурсов совершенно иным образом. Если процесс использовал и освободил ресурс, он должен ожидать, пока другие процессы воспользуются им, прежде чем он снова сможет получить к нему доступ. Этот порядок будет сохраняться даже в том случае, когда процесс очень скоро вновь будет нуждаться в данном ресурсе, а другой процесс какое-то время сможет обходиться без него.
2. Если два автомобиля одновременно въезжают в туннель с противоположных концов, каждый из них не будет знать о присутствии другого. Процесс въезда и включения света — это другой пример критической области или, в данном случае, критического процесса. Пользуясь этой терминологией, можно обобщить недостатки, сказав, что автомобили на противоположных концах туннеля могут начать выполнение критического процесса одновременно.
3. **а.** Это гарантирует, что потребности в разделении ресурса не возникнет и он не будет распределяться между многими процессами, т.е. автомобиль или получит в свое распоряжение весь мост, или не получит ничего.

   б. Это означает, что неделимый ресурс может быть получен принудительно.

   в. Это превращает неделимый ресурс в разделяемый, что устраняет всякую конкуренцию.
4. Последовательность стрелок, образующих замкнутый цикл в направленном графе. На этой основе был разработан метод, позволяющий некоторым операционным системам распознать ситуацию взаимной блокировки и предпринять соответствующие действия по ее устранению.

Раздел 3.5

1. Имена и даты считаются плохим выбором в качестве паролей, поскольку это типичный вариант, и следовательно, они представляют собой очень простую задачу в процедуре подбора или угадывания пароля.

Использование полных слов также рассматривается как неудачный выбор, поскольку такой пароль легко может быть раскрыт программой, последовательно проверяющей на совпадение все слова, имеющиеся в словаре. Кроме того, не являются надежными и пароли, состоящие только из цифр, поскольку в них используется очень ограниченный набор символов.

2. Четыре — это количество комбинаций битов, которые могут быть получены при использовании 2 битов. Если необходимо иметь больше уровней привилегий, потребуется уже как минимум 3 бита для представления их номеров, а следовательно, появится возможность реализации вплоть до 8 уровней привилегий. По той же самой причине естественным выбором для количества уровней привилегий меньше четырех будет 2, что является количеством комбинаций, представляемых одним битом.
3. Процесс сможет изменить программы самой операционной системы таким образом, что диспетчер будет передавать все кванты времени только этому процессу.

Глава 4

Раздел 4.1

1. Открытая сеть — это сеть с открытыми спецификациями и протоколами, позволяющими различным поставщикам производить совместимые продукты.
2. И то, и другое устройства предназначены для построения более крупных сетей с топологией шины за счет объединения сетей, меньших по размеру. Однако мост соединяет лишь две сети, позволяя пересылать сообщения из одной в другую, тогда как сетевой коммутатор обеспечивает объединение сразу нескольких сетей, устанавливая между ними соединения, каждое из которых функционирует как мост.
3. Маршрутизатор — это устройство, которое обеспечивает пересылку сообщений между сетями, образующими сеть из сетей.
4. Примерами могут служить фирма рассылки товаров по каталогу и ее клиенты, кассир банка и его клиенты, аптекарь-провизор в рецептурном отделе аптеки и его клиенты.
5. Жизнь современного общества пронизана множеством протоколов, среди которых можно назвать правила дорожного движения, нормы ведения телефонных разговоров, установленные правила этикета.

6. Кластерные вычисления обычно предполагают использование для проведения распределенных вычислений множества специально выделенных компьютеров при условиях их высокой доступности и равномерного распределения нагрузки. Грид-вычисления предполагают менее тесную взаимосвязь компьютеров в сравнении с кластерными вычислениями и могут предусматривать использование машин, которые присоединяются к распределенным вычислениям лишь в те моменты, когда не имеют иной нагрузки.

Раздел 4.2

1. Провайдеры первого и второго уровней обеспечивают формирования ядра всей функциональной структуры Интернета, тогда как назначение провайдеров доступа к Интернету состоит в предоставлении доступа к этому ядру конечным пользователям.
2. Подсистема DNS представляет собой охватывающую всю Всемирную сеть совокупность серверов имен, обеспечивающих преобразование мнемонических адресов в IP-адреса (а также преобразование адресов в обратном направлении).
3. Выражение 3.6.9 представляет битовую комбинацию из трех байтов `000000110000011000001001`. Комбинация битов `0001010100011100` может быть представлена в десятичной нотации с точками как 21.28.
4. На этот вопрос может быть представлено несколько ответов. Один состоит в том, что в обоих случаях используется переход от конкретного к общему. Интернет-адрес в мнемонической форме начинается с имени определенного устройства и последовательно переходит от домена к домену до названия домена высшего уровня. Почтовый адрес начинается с имени получателя и последовательно переходит ко все более и более крупным регионам, таким как улица, город, область, страна. В IP-адресе этот порядок меняется на обратный, поскольку он начинается с комбинации битов, определяющей домен.
5. Серверы имен служат для преобразования мнемонических адресов в IP-адреса. Почтовые серверы отправляют, получают и сохраняют сообщения электронной почты. FTP-серверы предоставляют услуги передачи и хранения файлов.
6. Протоколы могут описывать формат сообщений, которые пересылаются, правильный порядок следования сообщений в процессе обмена и само значение сообщений.
7. Они снимают с исходного сервера нагрузку, создаваемую рассылкой индивидуальных сообщений каждому клиенту. В случае одноранговой

модели эта нагрузка переносится на самих клиентов, тогда как при групповой передаче данных нагрузка перекладывается на маршрутизаторы Интернета.

8. Рассматриваемые критерии могут включать стоимость, портативность, практичность использования компьютера в качестве телефона, необходимость сохранить любые используемые аналоговые телефоны, важность доступа к службам экстренного вызова, надежность и область покрытия различных доступных провайдеров услуг.

Раздел 4.3

1. Адрес URL — это, в сущности, адрес документа в World Wide Web. Браузер — это программа, которая помогает пользователю получить доступ к гипертекстовому документу.
2. Язык разметки представляет собой систему вставки в документ некоторой поясняющей его особенности информации.
3. HTML — это конкретный язык разметки текстовых документов, тогда как XML представляет собой набор стандартных правил создания языков разметки.
4. **а.** Тег `<html>` отмечает начало HTML-документа.

   б. Тег `<head>` указывает на начало общего заголовка документа.

   в. Тегом `</p>` отмечается конец абзаца текста.

   г. Тегом `</a>` отмечается конец элемента-ссылки, связывающего данный документ с другим документом.
5. Сторона клиента и сторона сервера — это термины, используемые для указания, где именно выполняются действия: на компьютере пользователя или на компьютере сервера.

Раздел 4.4

1. Канальный уровень получает сообщение и передает его на сетевой уровень. На сетевом уровне определяется направление, в котором это сообщение следует переслать, после чего оно возвращается на канальный уровень для отправки. Более высокие сетевые уровни не используются для целей маршрутизации, хотя современные маршрутизаторы могут использовать транспортный и прикладной уровни для предоставления некоторых дополнительных услуг, например таких, как избирательная фильтрация или многоуровневое качество обслуживания.

2. В отличие от протокола TCP, UDP — это протокол без обратной связи, который не подтверждает получение сообщения адресатом.
3. Для определения, какому именно компоненту прикладного уровня следует передать поступившее сообщение, транспортный уровень использует номера портов транспортного протокола.
4. Фактически ничего. На любом компьютере в Интернете программист может так модифицировать программное обеспечение, что этот компьютер получит возможность сохранять все проходящие через него записи. Вот почему конфиденциальные данные следует шифровать.

Раздел 4.5

1. Сокет — это абстрактное представление процессов на прикладном уровне стека сетевых протоколов, используемых для подключения к другим процессам в сети с помощью средств транспортного уровня, расположенного ниже.
2. Для создания нового сокетного соединения требуется указать четыре элемента: сетевой адрес и номер порта источника, сетевой адрес и номер порта получателя.
3. Основное различие в процедурах создания клиентского и серверного соединений состоит в том, что сервер должен заранее начать прослушивать порт с номером, предварительно согласованным с клиентом, чтобы иметь возможность обнаружить запрос от клиента на установку с ним соединения и начать взаимодействие. Что касается клиента, то для организации взаимодействия ему достаточно просто предпринять попытку установить соединение с известным ему сервером.
4. В принципе клиент и сервер могут обмениваться символьными данными с использованием любого подходящего им обоим способа их представления. В частности, в языке Python функция `sendall()` библиотеки `socket` для передачи символьных данных использует однобайтовые коды ASCII.

Раздел 4.6

1. Фишинг представляет собой способ получения конфиденциальной информации посредством предложения пользователям ввести их пароли, номера кредитных карт и так далее и рассылки соответствующих сообщений электронной почты, замаскированных под сообщения от неких законных организаций, таких как банк или информационный отдел некой административной службы. Компьютеры не защищают от фишинга,

пользователи должны полагаться на свой здравый смысл, сообщая конфиденциальные данные другим лицам без надлежащей проверки.

2. Локальный шлюз представляет собой маршрутизатор, который просто пересылает пакеты (фрагменты сообщений) по мере их поступления. Следовательно, брандмауэр на шлюзе может фильтровать трафик не по его содержанию, а лишь по адресам пакетов.
3. Использование паролей защищает данные (а значит, и информацию). Использование шифрования защищает только информацию.
4. В случае использования системы шифрования с открытым ключом знания лишь того, как сообщение было зашифровано, недостаточно, чтобы его расшифровать.
5. Проблемы интернациональны по самой своей природе, а следовательно, не подчиняются юрисдикции единственного правительства. Более того, правовые нормы просто предоставляют пострадавшей стороне основания для юридических процессов, но никак не защищают их от нападения.

Глава 5

Раздел 5.1

1. Процесс — это выполнение алгоритма. Программа — это запись алгоритма.
2. Во вступительной главе этой книги приводились примеры алгоритмов для исполнения музыкальных произведений, управления стиральными машинами, конструирования моделей, выполнения фокусов, а также алгоритм Евклида. Многие из "алгоритмов", с которыми мы сталкиваемся в обыденной жизни, не соответствуют их формальному определению. В тексте был приведен пример алгоритма деления в столбик. Другой пример — алгоритм, который день за днем выполняют часы, перемещая свои стрелки и отсчитывая время.
3. Неформальное определение не отвечает требованию, по которому шаги алгоритма должны быть последовательными и однозначными. Оно сводится к общим требованиям, чтобы шаги были выполнимыми и в итоге приводили к определенному результату.
4. Здесь есть два важных момента. Первый заключается в том, что эти команды определяют бесконечный процесс. Второй — в том, что в действительности процесс рано или поздно достигнет ситуации, когда в кармане не останется ни одной монеты. Фактически алгоритм даже может

оказаться в этой ситуации уже в самом начале своей работы. С данной точки зрения задача является неоднозначной, поскольку представленный алгоритм не дает нам указаний, как поступить в этой ситуации.

Раздел 5.2

1. Один из примеров можно получить методом композиции. На химическом уровне примитивами считаются молекулы, хотя в действительности эти частицы состоят из атомов, которые, в свою очередь, образуются из электронов, протонов и нейтронов. Сегодня мы знаем, что даже эти "примитивы" имеют составные части.
2. Если функция составлена правильно, ее можно использовать в качестве строительного блока для более крупных программных структур без пересмотра ее внутреннего устройства.
3.
```
X = большее из двух заданных чисел
Y = меньшее из двух заданных чисел
while (Y ! = 0):
  Remainder = остаток от деления X на Y
  X = Y
  Y = Remainder
GCD = X
```
4. Все цвета можно получить посредством комбинирования красного, синего и зеленого цветов, поэтому экраны мониторов, телевизоров или смартфонов разрабатываются так, чтобы генерировать именно эти три основных цвета.

Раздел 5.3

1. **а.**
```
if (n == 1 or n == 2):
    ответ - это список, содержащий одно число n
  else:
    разделить n на 3, получив частное q и остаток r
    if (r == 0):
      ответ - это список, содержащий q троек
    if (r == 1):
      ответ - это список, содержащий (q - 1) троек
        и 2 двойки
    if (r == 2):
      ответ - это список, содержащий q троек и
        одну двойку
```

   б. Результатом был бы список, содержащий 667 троек.

   в. Возможно, вы экспериментировали с малыми входными величинами перед тем, как нашли подходящий вариант решения.

2. **а.** Да. *Подсказка.* Поместите первую фишку в центр доски так, чтобы она не попала в квадрант, содержащий вырезанный квадрат, а накрыла по одному квадрату во всех остальных квадрантах. В результате каждый квадрант будет представлять собой уменьшенный вариант исходной задачи.

   б. Доска с одним вырезанным квадратом содержит $2^{2n} - 1$ квадратов, и каждая фишка покрывает точно три квадрата.

   в. Задания *а* и *б* этого вопроса представляют собой замечательный пример того, как знание решения одной проблемы позволяет решить другую (см. четвертую фазу Полиа).

3. Исходное сообщение имеет вид "*This is the correct answer*".

4. Попытка просто собрать из кусочков полную картину будет представлять собой восходящий подход к решению задачи. Однако если предварительно взглянуть на крышку коробки с пазлом с целью увидеть, что представляет собой собираемая картина, то это добавит в выбранный вами подход элементы нисходящей методологии.

Раздел 5.4

1. Изменить условие в операторе `while` так, чтобы он читался следующим образом: "Искомое значение не равно текущему входному значению, и есть еще входные значения, подлежащие проверке".

2.
```
Z = 0
X = 1
repeat:
  Z = Z + X
  X = X + 1
  until (X == 6)
```

3. Доказано, что такой синтаксис может вызывать проблемы при чтении программ на языке C. Когда ключевые слова `do` и `while` одного цикла разделены несколькими строками программного текста, читающие программу часто испытывают затруднения с правильной интерпретацией оператора `while`. В частности, ключевое слово `while` в конце оператора `do` часто воспринимается ими как начало оператора `while`. Следовательно, практический опыт указывает, что лучшим подходом является использование разных ключевых слов для представления циклических структур с предпроверкой и постпроверкой условия.

4.

Cheryl	Alice	Alice
Gene	Cheryl	Brenda
Alice	Gene	Cheryl
Brenda	Brenda	Gene

5. Настаивать на том, чтобы опорный элемент помещался над равным ему элементом в списке, бессмысленно. Например, сделайте предложенные изменения, а затем примените новую программу к списку, все элементы которого одинаковы.

6. ```
   def Сортировка (Список):
     N = 1
     while (N меньше длины списка):
       J = N + 1
       while (J не больше длины списка):
         if (элемент в позиции J меньше, чем элемент
               в позиции N):
           поменять местами эти два элемента
       J = J + 1
     N = N + 1
   ```

7. Приведенное ниже решение является неэффективным. Можете ли вы сделать его более эффективным?

   ```
   def Сортировка (Список):
     N = длина списка
     while (N больше, чем 1):
       J = длина списка
       while (J больше, чем 1):
         if (элемент в позиции J меньше, чем элемент
               в позиции J - 1):
           поменять местами эти два элемента
       J = J - 1
     N = N - 1
   ```

Раздел 5.5

1. Первым проверяемым именем будет `Henry`, следующим будет `Larry` и последним будет искомое имя `Joe`.
2. 8, 17
3. 1, 2, 3, 3, 2, 1
4. Условием окончания является выражение "`N` больше или равно 3" (или "`N` не больше 3"). Это то условие, при котором новые дополнительные активации создаваться уже не будут.

Раздел 5.6

1. Если машина может отсортировать список из 100 имен за секунду, то она способна выполнить 1/4 (10 000 – 100) сравнений в секунду. Это означает, что каждое сравнение выполняется приблизительно за 0,0004 секунды. Следовательно, сортировка списка из 1000 имен (которая в среднем
   ```

потребует выполнения 1/4 (1000 000 – 1000) сравнений) займет около 100 секунд, или 12/3 минуты.

2. Алгоритм бинарного поиска принадлежит к классу $\Theta(\log_2 n)$, алгоритм последовательного поиска — к классу $\Theta(n)$, а алгоритм сортировки вставками — к классу $\Theta(n^2)$.
3. Класс $\Theta(\log_2 n)$ содержит наиболее эффективные алгоритмы, за которыми следуют алгоритмы классов $\Theta(n)$, $\Theta(n^2)$ и $\Theta(n^3)$.
4. Нет. Ответ неправильный, хотя может показаться верным. На самом деле у двух из трех карт обе стороны одинаковы. Следовательно, вероятность выбора такой карты равна 2/3.
5. Нет. Если делимое меньше делителя, как, например, в дроби 3/7, ответ будет равен 1, хотя он должен быть равен 0.
6. Нет. Если значение переменной `X` равно `0`, а значение переменной `Y` не равно `0`, то полученный ответ будет неверным.
7. Каждый раз, когда выполняется проверка условия прекращения суммирования, утверждение "*Sum* = 1 + 2 +... + *K* и *K* меньше или равно *N*" является истинным. Объединив его с условием прекращения суммирования "*K* больше или равно *N*", мы получим желаемый вывод "*Sum* = 1 + 2 +... + *N*". Поскольку переменная *K* инициализирована нулем и увеличивается на каждом шаге цикла, в итоге ее значение обязательно должно достичь значения *N*.
8. К сожалению, нет. Проблемы, выходящие за рамки управления разработкой аппаратного и программного обеспечения, такие как механические сбои и электрические помехи, могут оказать влияние на ход вычислений.

Глава 6

Раздел 6.1

1. Программа на языке третьего поколения является машинно-независимой в том смысле, что ее команды не содержат машинных атрибутов, таких как номера регистров и машинные адреса ячеек памяти. С другой стороны, ее можно считать машинно-зависимой, поскольку в ней по-прежнему возможны ситуации арифметического переполнения и появления ошибки округления.
2. Основное отличие заключается в том, что ассемблер переводит каждую команду исходной программы в единственную машинную команду, тогда как компилятор часто порождает сразу несколько команд на машинном языке, чтобы получить эквивалент единственной команды исходной программы.

3. Декларативная парадигма программирования основывается на разработке описания задачи, которую требуется решить. Функциональная парадигма требует от программиста описывать решение задачи в виде решений более мелких задач. Объектно-ориентированное программирование делает акцент на описании компонентов предметной области задачи.
4. Языки третьего поколения позволяют выразить программу в большей степени в терминах предметной области задачи и в гораздо меньшей степени зависимы от сугубо машинных аспектов, как это было в языках предыдущих поколений.

Раздел 6.2

1. Использование констант с описательными именами помогает повысить читабельность программы.
2. Оператор объявления вводит используемую в программе терминологию, а выполняемый оператор описывает отдельный этап реализуемого в программе алгоритма.
3. Это типы `Integer` (целые числа), `float` (действительные числа), `character` (символьный тип) и `Boolean` (логический тип).
4. В императивных и объектно-ориентированных языках программирования обычно присутствуют управляющие структуры типа оператора условия `if-else` и цикла `while`.
5. В однородном массиве (`array`) все компоненты имеют один и тот же тип, тогда как в неоднородном массиве (`aggregate` или `structure`) их тип может быть разным.

Раздел 6.3

1. Область видимости переменной — это область программы, в которой эта переменная является доступной.
2. Функция, возвращающая значение, — это такая функция, которая передает в вызывающую программу вычисленное значение, связанное с ее именем.
3. Потому что они именно этим и являются. Операторы ввода-вывода в действительности представляют собой лишь обращение к подпрограммам, входящим в состав операционной системы компьютера.
4. Формальный параметр является идентификатором, используемым только в пределах функции. Он представляет собой всего лишь "нишу" для

значения, т.е. для фактического параметра, который передается функции при ее вызове.

5. Функция, которая получает значение параметра по ссылке, потенциально способна внести в это значение изменения, которые будут видимы вызывающей программе. Напротив, при передаче параметра по значению функция получает лишь его копию, и любые изменения, которые она внесет в эту копию, останутся невидимыми за ее пределами.

Раздел 6.4

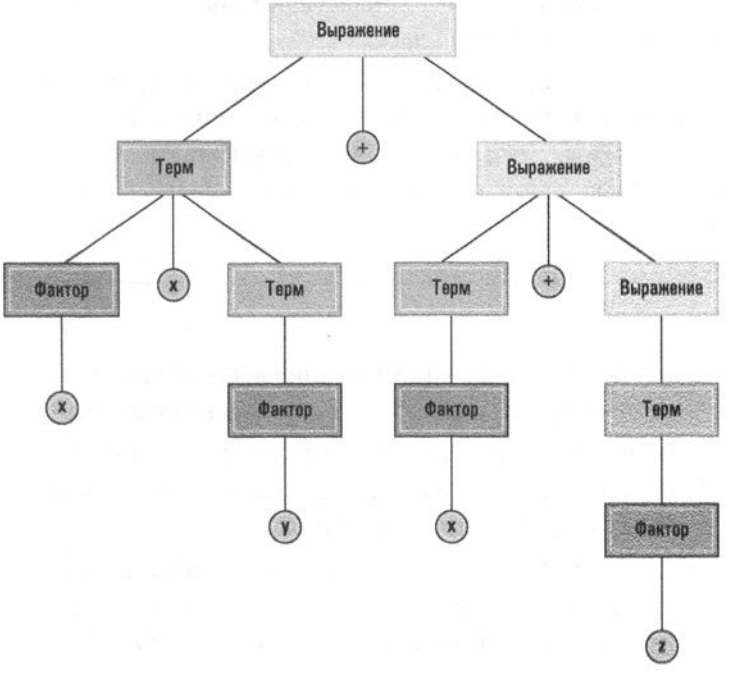
1. *Лексический анализ* — это процесс идентификации лексем.

   Синтаксический анализ — это процесс распознавания грамматической структуры программы.

   Генерация кода — это создание команд объектного кода программы.

2. *Таблица символов* — это место хранения информации, которую синтаксический анализатор извлекает из объявлений переменных в программе.

3. На синтаксических диаграммах термы, которые отображаются в овалах, являются терминальными. Термы, которые требуют дальнейшего уточнения, отображаются на синтаксических диаграммах в прямоугольниках и называются *нетерминальными*.

4. 

5. Строки, которые будут соответствовать структуре **Ча-ча-ча**, должны состоять из одной или более следующих подстрок:

```
forward backward cha cha cha
backward forward cha cha cha
swing right cha cha cha
swing left cha cha cha
```

Раздел 6.5

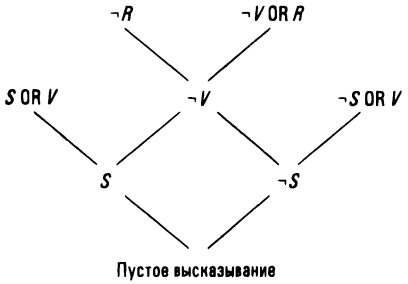
1. Класс — это описание структуры, свойств и функций объекта.
2. В программе, вероятно, должен присутствовать класс `MeteorClass`, на основании которого будут создаваться объекты, представляющие метеориты. В пределах класса `LaserClass` должна, вероятно, присутствовать переменная экземпляра с именем `AimDirection`, определяющая, в каком направлении нацелен лазер. Значение этой переменной могло бы использоваться такими методами, как `fire()` — *выстрел*, `turnRight()` — *повернуть право* и `turnLeft()` — *повернуть влево*.
3. Класс `Employee` может содержать информацию об имени сотрудника, его адресе, стаже и так далее, класс `FullTimeEmployee` — о пенсионных выплатах, а класс `PartTimeEmployee` — о количестве рабочих часов, которые должны быть отработаны за неделю, о почасовой ставке и т.п.
4. Конструктор — это специальный метод класса, который вызывается при создании экземпляра объекта этого класса.
5. Некоторые элементы в классе определяются как закрытые (`private`), чтобы предотвратить возможность прямого доступа к ним со стороны других элементов программы. Если элемент является закрытым, то последствия любой его модификации должны быть ограничены внутренними пределами класса.

Раздел 6.6

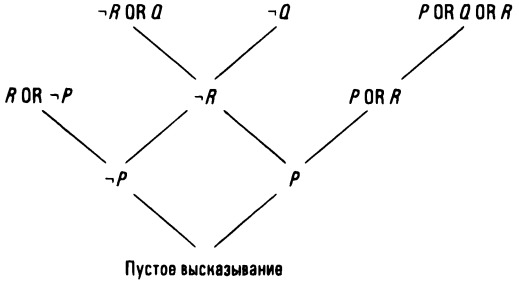
1. Список может включать способы инициирования выполнения параллельных процессов и методы реализации обмена данными между ними.
2. В первом ответственность возлагается на процессы, а во втором — на сами данные. Вторые имеют преимущество, заключающееся в концентрации задачи в единственной точке программы.
3. К таким областям следует отнести составление прогнозов погоды, управление потоками авиаперевозок, моделирование сложных систем (от ядерных реакторов до пешеходных потоков), организацию работы компьютерных сетей и сопровождение баз данных.

Раздел 6.7

1. Высказывания R, T и V. Например, мы можем показать, что высказывание R является следствием, посредством добавления его отрицания к совокупности высказываний и применения резолюции, которая может привести к пустому высказыванию, как показано ниже.



2. Нет. Совокупность высказываний является противоречивой, поскольку резолюция может привести к пустому высказыванию, как показано ниже.



3.
```
mother(X, Y) :- parent(X, Y), female(X).
father(X, Y) :- parent(X, Y), male(X).
```

4. Система Prolog придет к заключению, что carol является своей собственной сестрой. Чтобы решить эту проблему, необходимо, чтобы в правиле учитывался тот факт, что X не может быть равным Y. На языке Prolog это требование записывается как X \= Y. Следовательно, корректная версия правила будет выглядеть как

```
sibling(X, Y) :-X \= Y, parent(Z, X), parent(Z, Y).
```

и утверждать, что X является родным братом или сестрой Y, если X и Y не одно и то же лицо и при этом имеют общего родителя. В приведенной ниже еще одной версии правила дополнительно утверждается, что X является родным братом или сестрой Y только в том случае, если они имеют *двух* общих родителей.

```
sibling(X, Y) :- X \= Y, Z \= W
parent(Z, X), parent(Z, Y),
parent(W, X), parent(W, Y).
```

Глава 7

Раздел 7.1

1. В контексте разработки программы длинная последовательность операторов присваивания не сложнее нескольких вложенных операторов `if`.
2. Как возможный вариант в качестве метрики можно предложить количество ошибок, найденных после некоторого периода эксплуатации программного обеспечения. В данном случае одна из возможных проблем заключается в том, что эту величину невозможно измерить заранее.
3. Суть здесь в том, чтобы подумать, как можно измерить свойства программного обеспечения. Один возможный подход для оценки количества ошибок в определенной части программного обеспечения заключается в преднамеренном внесении в него некоторых ошибок непосредственно при разработке. Затем, после того как программное обеспечение предположительно отлажено, проверяют, много ли внесенных ошибок осталось неисправленными. Например, если из 7 преднамеренно внесенных ошибок 5 было выявлено, можно сделать вывод, что было исправлено только 5/7 от общего количества ошибок в этой части программы.
4. Возможные ответы включают обнаружение метрик, разработку готовых компонентов, разработку CASE-инструментов и переход к стандартам. Другое направление, которое будет рассматриваться ниже, в разделе 7.5, — это разработка систем моделирования и обозначений, таких как UML.

Раздел 7.2

1. Небольшие дополнительные усилия, приложенные на этапе разработки, могут оказаться чрезвычайно полезными при модификации программы.
2. В ходе анализа требований определяются задачи, которые должна решать предполагаемая система. На этапе разработки уточняется, как именно система будет выполнять свои задачи. При реализации осуществляется реальное создание системы. Этап тестирования необходим для того, чтобы удостовериться, что система действительно делает то, для чего она была предназначена.
3. Спецификации требований к программному обеспечению — это письменное соглашение между клиентом и фирмой по разработке программного обеспечения, в котором указываются требования и спецификации программного обеспечения, подлежащего разработке.

Раздел 7.3

1. Традиционный подход по принципу модели водопада требует, чтобы этапы анализа требований, проектирования, реализации и тестирования выполнялись строго в указанном порядке. Более новые модели допускают больше гибкости, открывая этим возможности для использования метода проб и ошибок.
2. Можно упомянуть пошаговую модель, итеративную модель и модель экстремального программирования (XP).
3. Традиционный эволюционный метод создания прототипов реализуется в пределах организации, разрабатывающей программное обеспечение, тогда как разработка программ с открытым исходным кодом не ограничивается некоторой единственной организацией. В случае разработки программ с открытым исходным кодом человек, руководящий разработкой, не обязательно должен определять, какие именно усовершенствования предполагаются и будут реализовываться, тогда как в случае традиционного эволюционного метода создания прототипов человек, управляющий разработкой программного обеспечения, назначает отдельных исполнителей для решения конкретных задач по улучшению программы.
4. Это вопрос, на который вы сами должны найти ответ. Если бы вы были администратором в компании, занятой созданием программного обеспечения, могли бы вы принять методологию разработки программ с открытым кодом при разработке программного обеспечения, которое ваша компания собирается продавать на рынке?

Раздел 7.4

1. Главы романа следуют одна из другой в единой сюжетной линии, в то время как статьи энциклопедии в значительной степени независимы одна от другой. Следовательно, между главами в романе существует больше связей, чем между статьями в энциклопедии. Однако статьи в энциклопедии, вероятно, имеют более высокий уровень связности, чем главы в романе.
2. Примером связанности по данным может быть общий счет матча. Другими примерами "связанности", которые могут иметь место, являются усталость, импульс, знания, полученные о стратегии противника и, возможно, уверенность в себе. Во многих видах спорта связность отдельных единиц увеличивается за счет отказа от конечного состояния команды при завершении одного этапа игры и начала следующего этапа

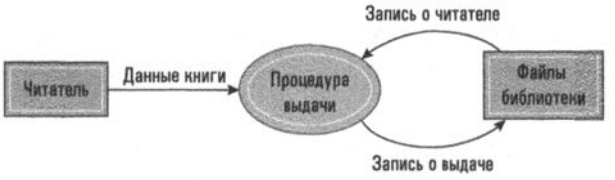
со стандартной исходной позиции. Например, в футболе или баскетболе каждый период начинается со стандартной схемы расположения игроков на поле, — без учета того, где они на нем находились в конце предыдущего периода. В других случаях счет в каждой единице игры ведется независимо от других, как в теннисе, где каждый сет может быть выигран или проигран независимо от того, с каким результатом закончились предыдущие сеты.

3. Это сложная задача. С одной стороны, можно было бы начать, поместив все в один модуль. В результате была бы достигнута низкая степень связности при полном отсутствии связей между модулями. Если начать деление этого единственного модуля на более мелкие модули, то в результате уровень связанности модулей будет повышаться. Отсюда мы можем заключить, что увеличение связности приводит к повышению связанности модулей задачи. С другой стороны, предположим, что рассматриваемая проблема естественно делится на три очень связных модуля, которые мы назовем `A`, `B` и `C`. Если в нашем первоначальном проекте не было такого естественного разделения (например, в одном модуле, возможно, были размещены половина решения задачи `A` вместе с половиной решения задачи `B` и т.д.), то можно ожидать, что исходно связность была низкой, а связанность высокой. В этом случае перепроектирование системы путем выделения задач `A`, `B` и `C` в отдельные модули, скорее всего, уменьшит межмодульную связанность и одновременно увеличит внутримодульную связность.
4. Уровень связанности определяется количеством связей, существующих между модулями программы. Связность представляет степень взаимосвязанности внутренних частей модуля. Сокрытие информации — это ограничение совместного доступа к информации.
5. Вам, вероятно, потребуется добавить стрелку, указывающую, что функция `ControlGame()` должна сообщить функции `UpdateScore()` о том, кто выиграл очередную подачу, а также еще одну стрелку в обратном направлении, указывающую, что функция `UpdateScore()` должна передать сведения о текущем состоянии всей игры (например, "сет окончен" или "матч окончен"), когда она возвращает управление функции `ControlGame()`.
6. На рис. 7.5 удалите все горизонтальные стрелки, за исключением первой и последней. Это будет означать, что объект `Judge` должен оценить подачу мяча, выполненную объектом `PlayerA`, и немедленно вызвать метод `updateScore()` объекта `Score`. (Это решение, безусловно, игнорирует возможность второй подачи. Как вы измените проект программы, чтобы реализовать возможность второй подачи после неудачной первой?)

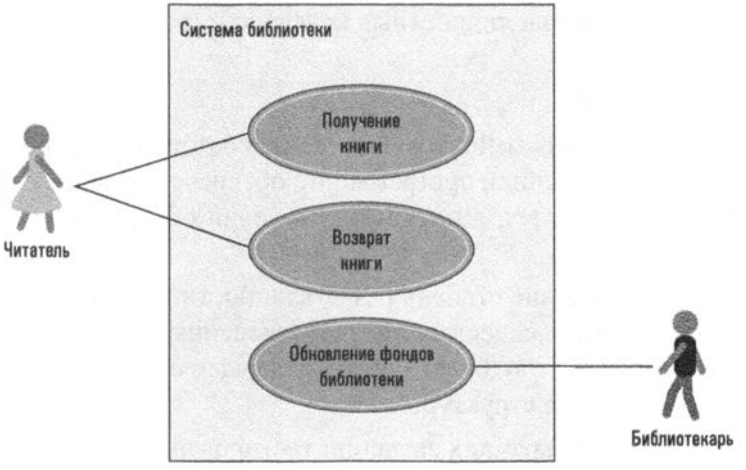
7. Традиционный программист пишет программы в терминах операторов, подобных тем, с которыми вы познакомились в главе 6. Ассемблер компонентов создает программу, связывая между собой уже готовые блоки, называемые компонентами.
8. На этот вопрос существует много ответов. Одна комбинация состоит в том, чтобы приложение Календарь автоматически устанавливало будильник в приложении Часы с целью уведомить пользователя о предстоящей встрече. Кроме того, приложение Календарь может использовать компоненты приложения Карты для предоставления указаний об адресе, по которому будет проводиться предстоящая встреча.

Раздел 7.5

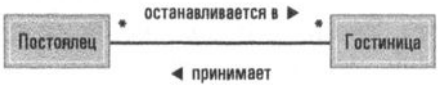
1. Убедитесь, что ваша диаграмма представляет именно движение данных в этой ситуации (а не перемещение книг). На диаграмме ниже показано, что данные о книге (от читателя) и запись о читателе (из файлов библиотеки) совместно используются для создания записи о выдаче книги, которая помещается в файлы библиотеки.



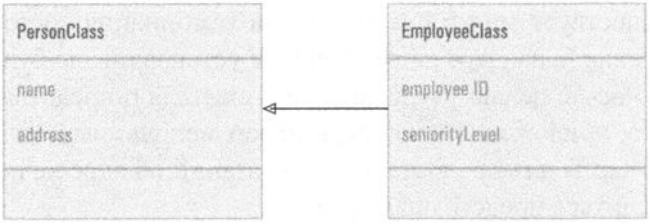
2.



3.



4. Понятие *человек* представляет класс `PersonClass` со свойствами `name` (имя) и `address` (адрес), а понятие *работник* — класс `EmployeeClass` со свойствами `employee ID` (идентификатор работника) и `seniorityLevel` (уровень старшинства).



5. Просто нарисуйте прямоугольник, охватывающий всю диаграмму, и добавьте в его верхний левый угол метку **sd**, как на рис. 7.13.
6. Шаблоны проектирования предоставляют стандартизованные, тщательно разработанные подходы к реализации повторяющихся структур в проектах систем программного обеспечения.

Раздел 7.6

1. Группа обеспечения качества (SQA) осуществляет контроль и обеспечение соблюдения требований систем контроля качества, принятых в организации.
2. Люди имеют склонность не вести записи о том, что было ими сделано по ходу выполнения проекта (принятые решения, выполненные действия и т.д.). Им также свойственны личностные конфликты, ревность, столкновение интересов.
3. Ведение записей и аудит.
4. Задача тестирования программного обеспечения состоит в обнаружении ошибок. Поэтому разработчики программного обеспечения рассматривают как неудачный такой тест, который не позволил обнаружить новую ошибку.
5. Можно принять во внимание степень разветвленности в модуле. Например, процедурный модуль, содержащий многочисленные циклы и операторы `if-then-else`, вероятно, будет более подвержен ошибкам, чем модуль с простой логической структурой.
6. Согласно методу анализа крайних значений тестирование программного обеспечения следует выполнить как с входным списком из 100 элементов, так и с пустым входным списком. Кроме того, полезно будет проверить, как поведет себя программа, если входной список уже будет иметь требуемую упорядоченность.

Раздел 7.7

1. Документация принимает форму пользовательской документации, системной документации и технической документации. Она может иметь вид сопроводительной документации; внутри программы может быть представлена в форме комментариев и ясно написанного кода; принимать вид интерактивных сообщений, которые программа может выводить на экран дисплея; иметь вид словарей данных или проектной документации, такой как структурные схемы, диаграммы классов, схемы потоков данных и диаграммы последовательностей.
2. И на стадии разработки, и на стадии модификации. Дело в том, что вносимые изменения должны быть документированы так же тщательно, как и исходная версия программы. (Программное обеспечение документируется и на стадии использования. Например, пользователь системы может обнаружить проблемы, которые затем, вместо исправления, будут просто описаны в последующих изданиях руководства пользователя. Более того, широко распространены книги, посвященные использованию популярных систем программного обеспечения, которые были написаны уже после того, как соответствующее программное обеспечение уже некоторое время использовалось.)
3. Разные люди будут иметь различные мнения об этом. Одни будут утверждать, что цель всего проекта — это программа, поэтому именно она является более важной. Другие будут утверждать, что программа ничего не стоит, если она не документирована, поскольку если невозможно понять, что она делает, то и использовать (или модифицировать) ее будет невозможно. Более того, при наличии хорошей документации задача создания программы может быть "легко" решена заново.

Раздел 7.8

1. **а.** Как насчет возможности регулировки наклона дисплея или формы мыши? В отношении смартфонов как насчет использования сенсорных экранов вместо мыши или наклона телефона для обеспечения удобства ввода?

   б. Как насчет расположения окна на дисплее, включая дизайн панелей инструментов, полос прокрутки и раскрывающихся меню? Что касается смартфонов, как насчет наведения камеры на предметы, представляющие интерес в понимании человека?
2. **а.** Было бы крайне непрактично и неудобно использовать на смартфоне мышь (или даже стилус). Кроме того, гораздо меньший размер экрана

смартфона требует, чтобы по причине пространственных ограничений любых несущественных элементов на экране было как можно меньше. По этой причине полосы прокрутки либо вовсе опускаются, либо отображаются в виде тонких линий.

б. Скользящее прикосновение к экрану дисплея — это вполне естественный жест для нашего сознания. Аналогичным жестом мы можем перемещать на письменном столе бумаги или другие предметы. Можно добавить, что это движение для нас более естественно, чем использование полос прокрутки на настольном компьютере. Когда при перетаскивании ползунок на полосе прокрутки движется в указанном направлении, само прокручиваемое изображение перемещается в противоположном направлении. Для человека, который никогда не пользовался компьютером, такое поведение может показаться нелогичным.

3. Возможный ответ — "Учет различных психологических и физиологических характеристик человека". Другой хороший ответ — "Дизайн интерфейса фокусируется на внешних, а не внутренних, характеристиках программной системы".
4. Три характеристики человека, которые обсуждаются в тексте, — это формирование привычек, узость внимания и ограниченные возможности работать с несколькими фактами одновременно. Можете ли вы предложить что-либо иное? Что можно сказать относительно склонности человека делать предположения?

Раздел 7.9

1. Уведомление об авторских правах подтверждает право собственности на произведение и определяет круг лиц, имеющих право использовать это произведение. Все выполненные работы, включая спецификации требований, проектную документацию, исходный код и конечный продукт, обычно требуют значительных инвестиций в производство. Физическое лицо или корпорация должна предпринять определенные шаги, чтобы гарантировать, что ее права собственности защищены и что вся ее интеллектуальная собственность не используется нежелательными сторонами.
2. Авторские права и патентное законодательство приносят обществу выгоду, поскольку стимулируют создателей новой продукции делать ее общедоступной. Без такой защиты компании будут испытывать сомнения в целесообразности крупных инвестиций в новые продукты.
3. Отказ от ответственности не защищает компанию от юридического преследования за небрежность.

Глава 8

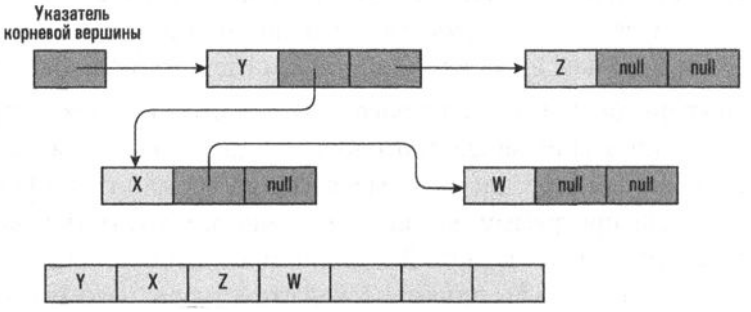
Раздел 8.1

1. **Список**. Список членов спортивной команды.
   Стек. Стопка подносов в кафетерии.
   Очередь. Очередь посетителей в кафетерии.
   Дерево. Организационная схема построения различных управляющих структур.
2. Стеки и очереди можно рассматривать как особые типы списков. В случае *списка* общего типа записи могут быть вставлены и удалены в любом его месте. В случае *стека* запись можно вставить только в его вершину и удалить тоже только из его вершины. В случае *очереди* запись может быть вставлена только в хвост, а удалена только из головы очереди.
3. Буквами в стеке в направлении от вершины к основанию будут ***E***, ***D***, ***B*** и ***A***. Если извлечь букву из стека, то это будет буква ***E***.
4. Буквами в очереди в направлении от головы к хвосту будут ***B***, ***C***, ***D*** и ***E***. Если удалить из очереди букву, то это будет буква ***B***.
5. Листовыми (конечными) узлами этого дерева являются узлы ***D*** и ***C***. Корнем дерева должен быть узел ***B***, поскольку у всех остальных имеются родительские вершины.

Раздел 8.2

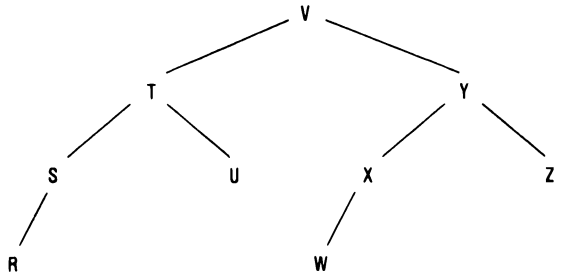
1. Данные в основной памяти компьютера фактически хранятся в индивидуально адресуемых ячейках памяти. Структуры данных, такие как массивы, списки или деревья, моделируются в ней таким образом, чтобы сделать эти данные в большей степени доступными для их пользователей.
2. Если бы вы должны были написать программу для игры в шашки, структура данных, представляющая шахматную доску, вероятно, была бы статической, поскольку размер доски не меняется во время игры. Однако, если бы вы писали программу для игры в домино, структура данных, представляющая раскладку из костей домино, построенную на столе во время игры, вероятно, была бы динамической структурой, потому что эта раскладка по ходу игры и от игры к игре будет меняться в размерах и не может быть предопределена заранее.
3. Телефонный справочник — это набор указателей (телефонных номеров) на людей. Улики, оставленные на месте преступления, являются (возможно, зашифрованными) указателями на преступника.

Раздел 8.3

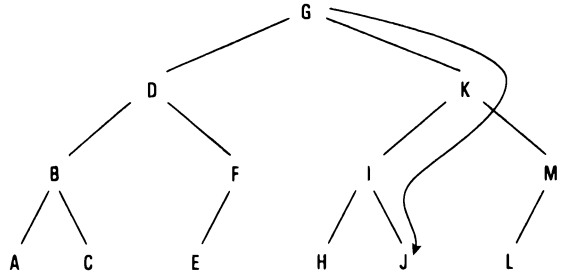
1. 5 3 7 4 2 8 1 9 6
2. Если r — это номер строки матрицы, то формула будет следующей: $r(j-1)+(i-1)$.
3. $(c-i)+j$
4. Указатель начала списка имеет значение `null`.
5.
```
def PrintList(List):
  Last = последнее имя, которое должно быть напечатано
  Finished = false
  CurrentPointer = List.Head
  while ((CurrentPointer не равен null) and (Finished не
          равно false)):
    print(CurrentPointer.Value)
    if (только что напечатанное имя == Last):
      Finished = true
    CurrentPointer = CurrentPointer.Next
```
6. Указатель стека указывает на ячейку, которая находится непосредственно ниже основания стека.
7. Представьте стек в виде одномерного массива, а указатель стека — в виде переменной целого типа. Затем этот указатель стека используйте в качестве индикатора положения вершины стека в массиве вместо указания точного адреса памяти.
8. И при пустой, и при полностью заполненной очереди значения в указателях на ее вершину и хвост совпадают. Поэтому, чтобы различить эти два состояния очереди, потребуется дополнительная информация.
9. 

Раздел 8.4

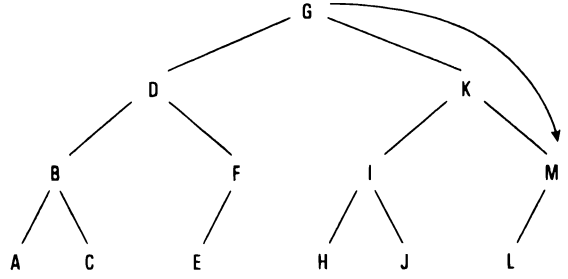
1.



2. При поиске значения J:



При поиске значения P:



3.

```
def PrintTree(Tree):
  if (Tree is not None):
    PrintTree(Tree.Left)
    print(Tree.Value)
    PrintTree(Tree.Right)
```

```
def PrintTree(Tree):
  if (Tree is not None):
    PrintTree(Tree.Left)
    print(Tree.Value)
    PrintTree(Tree.Right)
```

Здесь при печати K

4. В каждом узле каждый дочерний указатель может использоваться для представления уникальной буквы в алфавите. Слово может быть представлено в виде пути вниз по дереву согласно последовательности указателей, представляющих написание слова. Если узел представляет конец правильно написанного слова, он может быть помечен особым образом.

Раздел 8.5

1. Тип данных — это шаблон, а реализация этого типа — реальный объект, созданный на основании этого шаблона. В качестве примера можно привести следующую аналогию: *собака* — это тип животного, тогда как *Ласси* и *Рекс* являются реализациями этого типа.
2. Определяемый пользователем тип данных представляет собой лишь описание *организации* данных, тогда как абстрактный тип данных дополнительно включает в себя операции, предназначенные для манипулирования данными этого типа.
3. Предварительно следует отметить, что возможен выбор между реализацией списка в виде непрерывного списка либо в виде связанного списка. Этот выбор повлияет на структуру функций, предназначенных для вставки новых элементов, удаления старых и поиска заданных элементов. Однако ваш выбор не должен быть виден пользователю экземпляра абстрактного типа данных.
4. Абстрактный тип данных будет содержать как минимум описание структуры данных для хранения баланса счета, а также функции, предназначенные для внесения депозита и снятия средств посредством выдачи чека.

Раздел 8.6

1. Как абстрактные типы данных, так и классы являются шаблонами для создания экземпляров (реализаций) данного типа. Однако классы являются более общими понятиями в том смысле, что они поддерживают механизм наследования и могут описывать даже такой набор компонентов, который состоит только из функции.
2. Класс — это шаблон, на основании которого создаются объекты.
3. Класс может включать циклическую очередь вместе с функциями, обеспечивающими для нее добавление элементов, удаление элементов, проверку полного заполнения очереди и проверку, является ли очередь пустой.

Раздел 8.7

1. **а.** `0xA5`

   б. `0xA5`

   в. `0xCA`
2. `0xD50F`, `0x2EFF`, `0x5FFE`

3. 0x2EA0, 0x2FB0, 0x2101, 0x20B5, 0xD50E, 0xE50F, 0x5EE1, 0x5FF1, 0xBF14, 0xB008, 0xC000
4. При обходе связанного списка, в котором каждая запись состоит из двух ячеек памяти (ячейка данных, за которой следует указатель на следующую запись), для извлечения данных можно использовать команду вида 0xDR0S, а команду 0xDR1S — для получения указателя следующей записи. Если бы использовалась форма 0xDRTS, то точное определение ячейки памяти, на которую ссылаются, можно было бы откорректировать, изменяя значение в регистре T.

Глава 9

Раздел 9.1

1. Отдел снабжения может быть заинтересован в записях о запасах на складе, чтобы размещать заказы на сырье, в то время как бухгалтерия нуждается в информации для составления балансового отчета.
2. Модель базы данных обеспечивает такое видение базы данных в масштабах всей организации, которое более совместимо с прикладными приложениями, чем ее фактическая организация. Следовательно, определение модели базы данных является первым шагом к обеспечению возможности использования базы данных в качестве абстрактного инструмента.
3. Прикладное программное обеспечение переводит запросы пользователя, сформулированные в терминах приложения, в эквивалентную серию запросов, выраженную в терминах модели базы данных, поддерживаемой СУБД. В свою очередь, система управления базой данных преобразует эти запросы в требуемую последовательность действий, выполняемых с реальной базой данных.

Раздел 9.2

1. **а.** Джерри Смит **б.** Шерил Кларк **в.** S26Z
2. Одно из решений будет следующим:

```
TEMP ← SELECT from JOB
     where Dept = "PERSONNEL"
LIST ← PROJECT JobTitle from TEMP
```

В некоторых системах это приводит к появлению списков, в которых одно и то же название работы повторяется несколько раз. Это значит, что в нашем списке может несколько раз встречаться должность “секретарь”.

Однако обычно операцию PROJECT разрабатывают так, чтобы удалить повторяющиеся кортежи из результирующего отношения.

3. Одно из решений будет следующим:

```
TEMP1 ← JOIN JOB and ASSIGNMENT
      where JOB.JobId = ASSIGNMENT.JobId
TEMP2 ← SELECT from TEMP1
      where TermDate = "*"
TEMP3 ← JOIN EMPLOYEE and TEMP2
      where EMPLOYEE.EmplId = TEMP2.EmplId
RESULT ← PROJECT Name, Dept from TEMP3
```

4.
```
select JobTitle
from JOB
where Dept = "PERSONNEL"
select EMPLOYEE.Name, JOB.Dept
from JOB, ASSIGNMENT, and EMPLOYEE
where (Job.Job = ASSIGNMENT.JobId) and
(ASSIGNMENT.EmplId = EMPLOYEE.EmplID)
and (ASSIGNMENT.TermDate = "*")
```

5. Сама по себе модель не обеспечивает независимость данных. Это прерогатива системы управления базой данных. Независимость данных достигается путем наделения системы управления базой данных способностью предоставлять прикладному программному обеспечению неизменную структуру организации отношений, даже если реальная организация отношений будет изменяться.

6. Посредством наличия общих атрибутов. Например, отношение EMPLOYEE в этом разделе связано с отношением ASSIGNMENT через атрибут Emplld, а отношение ASSIGNMENT — с отношением JOB с помощью атрибута Jobld. Атрибуты, используемые для соединения отношений между собой, иногда называют атрибутами соединения.

Раздел 9.3

1. Это могут быть методы присваивания и извлечения значения как атрибута StartDate, так и атрибута TermDate. Еще один метод может применяться для подготовки отчета о полном стаже работника.

2. Постоянный объект — это объект, который может сохраняться в базе данных бесконечно долго.

3. Один из способов — создать объект для каждого типа продукции на складе. Каждый из этих объектов мог бы хранить сведения об остатке соответствующей продукции на складе, ее стоимости и содержать ссылки на неоплаченные заказы на нее.

4. Как было показано в начале раздела, объектно-ориентированные базы данных были созданы для упрощения работы с неоднородными типами данных — по сравнению с реляционными базами данных. Кроме того, тот факт, что объекты могут содержать методы, которые будут играть важную роль при подготовке ответов на запросы, дает объектно-ориентированным базам данных дополнительное преимущество над реляционными базами данных, в которых базовые отношения содержат только данные.

Раздел 9.4

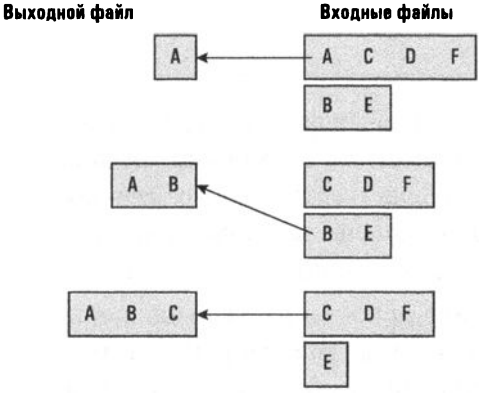
1. После того как транзакция достигнет точки фиксации результатов, система управления базой данных принимает на себя обязанность следить за тем, чтобы транзакция была завершена полностью. Транзакция, которая не достигла своей точки фиксации, лишена такой гарантии. Если возникают проблемы, может потребоваться выполнить ее еще раз.
2. Один из способов — на мгновение приостановить выполнение пересекающихся транзакций, что позволит всем конкурирующим транзакциям полностью завершить свою работу. Таким образом, будет зафиксирована точка, в которой потенциальный каскадный откат будет остановлен.
3. Если бы транзакции выполнялись поочередно, конечный остаток на счету был бы равен 100 долларам. Если же первая транзакция будет выполнена после того, как вторая считает исходное значение остатка, но до того, как она сохранит свои результаты, окончательный остаток на счету будет равен 200 долларов. Если же вторая транзакция будет выполнена после того, как первая считает исходное значение остатка, но до того, как она сохранит результаты своего выполнения, окончательный остаток на счету составит 300 долларов.
4. **а.** Если никакая другая транзакция не имеет монопольного доступа, совместный доступ к данным разрешается.

   б. Если другая транзакция уже имеет некоторую форму доступа, система управления базой данных обычно заставляет новую транзакцию подождать или выполняет откат других транзакций и предоставляет доступ новой транзакции.
5. Ситуация взаимной блокировки может возникнуть в том случае, если каждая из двух транзакций получает монопольный доступ к различным элементам данных, а затем требует получения доступа к тем элементам, которые уже захвачены другой транзакцией из этой пары.

6. Указанную выше ситуацию взаимной блокировки можно устранить посредством отката одной из транзакций (с использованием файла журнала) и предоставления другой транзакции доступа к тому элементу данных, который ранее удерживался первой транзакцией.

Раздел 9.5

1. Необходимо пройти следующие стадии.



2. Идея такова: сначала нужно разделить файл так, чтобы хранить его содержимое во множестве отдельных файлов, содержащих по одной записи. Затем следует сгруппировать содержащие по одной записи файлы попарно и применить к каждой паре алгоритм сортировки слиянием. В результате будет получено в два раза меньше файлов, содержащих по две записи, причем каждый из них будет уже отсортирован. Теперь можно снова сгруппировать их попарно и вновь применить к этим парам алгоритм сортировки слиянием. В итоге получим вдвое меньше файлов вдвое большего размера, каждый из которых также уже упорядочен. Продолжая действовать подобным образом, в итоге получим один файл, который будет содержать все исходные записи, отсортированные в требуемом порядке. (Если на какой-либо стадии этого процесса встречается нечетное количество файлов, следует просто отложить один из файлов в сторону и объединить его с файлом большего размера на следующей стадии.)
3. Если файл хранится на ленте или компакт-диске, его физическая организация, скорее всего, является последовательной. Однако если файл хранится на магнитном диске, он, скорее всего, разбросан по различным секторам на диске, и последовательный характер файла является концептуальным свойством, которое поддерживается системой указателей или некоторой формой списка, содержащего записи о секторах, в которых хранятся фрагменты файла.

4. Сначала необходимо найти требуемый ключ в файле индекса. Это позволит узнать местоположение требуемой записи. Когда известно, где находится нужная запись, остается лишь считать ее оттуда.
5. Неудачно выбранный алгоритм хеширования приводит к более высокой кластеризации данных, чем обычно, а значит, и к увеличению случаев переполнения сегментов. Поскольку записи переполнения каждого раздела запоминающего устройства организуются в виде связанного списка, поиск записей в области переполнения, по сути, является поиском в последовательном файле.
6. Номера назначенных сегментов будут следующими.

   а. `0` **б.** `0` **в.** `3` **г.** `0` **д.** `3`

   е. `3` **ж.** `3` **з.** `3` **и.** `3` **к.** `0`

   Значит, все записи будут помещены в сегменты с номерами `0` и `3`, а сегменты `1`, `2`, `4` и `5` останутся пустыми. Проблема состоит в том, что число участков памяти, которые нужно использовать (6), и значения в ключевом поле имеют общий делитель `3`. (Можете проверить работу этого же алгоритма хеширования для этих же значений ключевых полей, но с использованием семи участков памяти, чтобы убедиться, насколько ситуация улучшится.)
7. По сути, задача сводится к тому, чтобы использовать алгоритм хеширования при распределении людей из некоторой группы в одну из 365 категорий. Очевидно, что алгоритм хеширования сводится к вычислению дня рождения. Забавно, что достаточно только 23-х человек, чтобы вероятность совпадения дней рождения хотя бы у двух из этой группы превысила 0,5. В терминах хешированных файлов это означает, что при хешировании записей по 365 доступным участкам памяти запоминающего устройства кластеризация с большой вероятностью может возникнуть уже после ввода первых 23 записей.

Раздел 9.6

1. Поиск шаблонов в динамических данных связан с определенными трудностями.
2. *Описание классов* — определение характеристик подписчиков определенного журнала.

   Различение классов — идентификация особенностей, которыми различаются подписчики двух журналов.

   Кластерный анализ — определение журналов, которые привлекают схожих подписчиков.

Ассоциативный анализ — определение связей между подписчиками различных журналов и различных покупательских привычек.

Анализ выбросов — определение тех подписчиков журнала, которые не соответствуют профилю его обычных подписчиков.

Анализ последовательных шаблонов — определение тенденций в подписке на журналы.

3. Технология куба данных позволяет анализировать данные о продажах в разрезе продаж по месяцам, продаж по географическим регионам, продаж по классам продуктов и т.д.
4. Традиционные запросы к базе данных извлекают факты, хранящиеся в базе данных. Проведение интеллектуального анализа данных предполагает поиск закономерности среди этих фактов.

Раздел 9.7

1. Сравните ваш ответ на этот вопрос с ответом на следующий вопрос. Оба они поднимают одну и ту же проблему, но в разном контексте.
2. Смотрите ответ на предыдущий вопрос.
3. Вы могли бы получать сообщения или приглашения, которые иначе не смогли бы получать, но вы могли бы также стать объектом приставаний или жертвой преступления.
4. Дело здесь в том, что свободная пресса может предупреждать публику об имевших место потенциальных злоупотреблениях и таким образом будоражить общественное мнение. В большинстве случаев, описанных в тексте, именно свободная пресса вынудила предпринять действия по исправлению ошибок, предупредив общественность о существующей проблеме.

Глава 10

Раздел 10.1

1. Обработка изображений связана с анализом двухмерных изображений; назначение 2D-графики — преобразование двухмерных фигур в изображения, тогда как задача 3D-графики — преобразование в изображения трехмерных сцен.
2. Традиционная фотография позволяет создавать изображения реальных сцен, тогда назначение 3D-графики — создание изображений виртуальных сцен.
3. Первый этап — это "построение" самой виртуальной сцены. На втором этапе на ее основе осуществляется создание изображений.

Раздел 10.2

1. Три этапа создания изображения в 3D-графике — это *моделирование* (построение сцены), *рендеринг* (создание изображения) и *отображение* (отображение созданного изображения).
2. Окно изображения — это *часть* проекционной плоскости, используемая для построения изображения.
3. Буфер кадра — это область памяти, в которой содержится закодированная версия изображения.

Раздел 10.3

1. Это ромб (скошенный квадрат).
2. Процедурная модель — это программный модуль, управляющий построением объекта.
3. Список объектов может включать покрытую травой землю, вымощенную камнем дорожку, беседку, деревья, кустарники, облака, солнце и актеров. Суть здесь в том, чтобы подчеркнуть масштаб графа сцены — он может содержать множество деталей.
4. Представление *всех* объектов в виде полигональной сетки обеспечивает единый подход к процедуре рендеринга. (В большинстве случаев рендеринг рассматривается как задача рендеринга плоских граней, а не рендеринга объектов в целом.)
5. Наложение текстуры — это инструмент для связывания двухмерного изображения с поверхностью объекта.

Раздел 10.4

1. *Зеркальный свет* отражается от поверхности примерно в одном направлении. *Диффузный свет* отражается от поверхности одновременно в разных направлениях. *Рассеянный свет* — это свет, который не имеет точно определяемого источника.
2. Отсечение — это процесс исключения из дальнейшего рассмотрения тех объектов (или частей объектов), которые не лежат в визуальном коридоре.
3. Предположим, что подсветка должна появиться в середине грани, и она вызывается определенной ориентацией поверхности объекта в этой точке грани. Поскольку при затенении по методу Гуро ориентация поверхности объекта учитывается только вдоль границ грани, данный эффект

подсветки, скорее всего, будет пропущен. Однако, поскольку при затенении по методу Фонга выполняется определение ориентации поверхности объекта для точек в пределах всей грани, наличие эффекта подсветки, скорее всего, будет обнаружено.

4. Конвейер рендеринга обеспечивает стандартизированный подход к процедуре рендеринга, что в конечном итоге приводит к повышению эффективности систем реализации рендеринга. В частности, конвейер рендеринга может быть реализован во встроенном программном обеспечении компьютера, а это означает, что процесс рендеринга может выполняться быстрее, чем если бы задача была реализована с помощью традиционного программного обеспечения.
5. Цель этого вопроса — заставить вас задуматься о различиях между локальными и глобальными моделями освещения, а не дать конкретный, заранее определенный ответ. Потенциальные решения, которые вы могли бы предложить, включают в себя размещение соответствующим образом измененных копий объектов за поверхностью зеркала с учетом того, что поверхность зеркала прозрачна, или попытки обработки изображений в зеркале в виде отбрасываемых теней.

Раздел 10.5

1. Нас интересуют только те лучи, которые в конечном итоге достигают окна изображения в отдельных его пикселях. Если начинать отслеживание лучей от источника света, заранее не известно, по какими именно лучами необходимо следовать.
2. Метод распределенной трассировки лучей разработан с целью избежать избыточного сияния объектов сцены, присущего им при использовании традиционного метода трассировки, — за счет отслеживания нескольких лучей, исходящих из каждой точки отражения.
3. Метод излучательности требует выполнения очень большого объема расчетов и не позволяет точно отображать зеркальные эффекты.
4. Как метод трассировки лучей, так и метод излучательности реализуют глобальную модель освещения, и оба эти метода являются весьма вычислительно интенсивными. Однако метод трассировки лучей имеет тенденцию к получению чрезмерно блестящих поверхностей, тогда как в методе излучательности создаваемые поверхности чаще всего имеют тусклый оттенок.

Раздел 10.6

1. Точного ответа на этот вопрос нет. Если считать, что изображение задерживается в нашем восприятии точно на 200 миллисекунд, и, соответственно, отображать на экране по пять кадров в секунду, то к моменту проецирования следующего кадра предыдущий кадр уже просто исчезнет. Это, вероятно, приведет к пульсациям изображения, и его будет неудобно смотреть в течение длительного времени, но эффект анимации все же будет иметь место. (В действительности относительно медленная скорость смены кадров может приводить к эффекту грубой анимации.) Именно по этой причине минимальная скорость в пять кадров в секунду оказывается намного ниже кинематографического стандарта движущегося изображения, равного двадцати четырем кадрам в секунду.
2. *Раскадровка* — это последовательность изображений сцен в ключевых точках сюжета.
3. *Фазовка* — это процесс создания кадров, которые заполняют промежутки между ключевыми кадрами.
4. *Динамика* — это раздел механики, описывающий движение объектов как следствие действующих на них сил. *Кинематика* — это раздел механики, описывающий движение объекта без учета тех сил, которые вызывают это движение.

Глава 11

Раздел 11.1

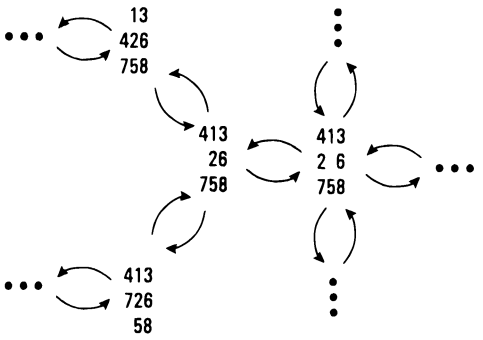
1. Упомянутые в тексте главы типы включают рефлективные действия; действия, основанные на знаниях о реальном мире; действия, направленные на достижение поставленной цели; обучение и восприятие.
2. Мы не преследуем здесь цель получить однозначный ответ на данный вопрос, — он приведен только лишь для того, чтобы показать, насколько тонкими являются аргументы, касающиеся понятия "интеллект".
3. Хотя большинство, без сомнения, ответят "нет", мы, вероятно, дали бы положительный ответ на этот вопрос, если бы он касался тех же действий и в той же обстановке, но совершаемых человеком. Обратите внимание, что подобная реакция могла бы иметь место даже в том случае, если бы мы не смогли объяснить, в чем же заключаются отличия в каждом случае.

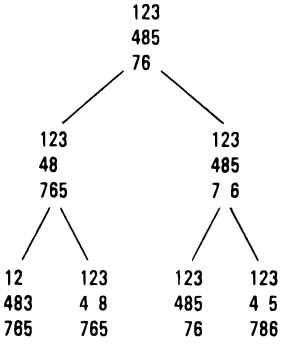
4. Не существует правильного или неправильного ответа на этот вопрос. Правильнее всего будет утверждать, что машина демонстрирует поведение, напоминающее разумное.
5. Не существует правильного или неправильного ответа на этот вопрос. Следует отметить, что чат-боты, т.е. программы, предназначенные для эмуляции чата, испытывают трудности в ведении содержательного разговора даже в течение относительно короткого периода времени. В результате чат-боты могут быть легко идентифицированы как машины.

Раздел 11.2

1. В случае дистанционного управления система нуждается только в получении изображения, тогда как робот, чтобы воспользоваться изображением для маневрирования, должен "понимать" его смысл.
2. Возможная интерпретация одной части рисунка никак не согласуется с интерпретацией другой его части. Для того чтобы отразить эту точку зрения в программе, можно было бы выделить интерпретации, допустимые для различных пересечений линий, а затем написать программу, которая пытается найти множество совместимых интерпретаций (по одной на каждое пересечение). В действительности, если лучше обдумать предложенное решение, то, вероятно, оно так или иначе совпадет с теми действиями, которые вы сами предпринимаете при попытке распознать изображение. Замечаете ли вы, что переводите взгляд с одного конца рисунка на другой, одновременно пытаясь совместить различные возможные интерпретации? (Если эта тема вас заинтересовала, обратитесь к работам Д.Э. Хоффмана (D.A. Huffman), М.Б. Кловса (M.B. Clowes) и Д. Уолтца (D. Waltz).)
3. На рисунке изображена группа из четырех кубиков, но только три из них являются видимыми. Дело в том, что для понимания этой ситуации требуются значительные "умственные" усилия.
4. Интересно, не правда ли? Подобные тонкие различия в значении представляют значительную проблему в области понимания естественного языка.
5. Для точного перевода предварительно необходимо получить информацию, о чем идет речь в данном контексте — о породах лошадей или о стиле работы некоторых людей.
6. В результате грамматического анализа будут получены идентичные структуры, однако при семантическом анализе будет установлено, что в первом случае говорится о том, где была построена изгородь, а во втором — когда она была построена.
7. Они брат и сестра, причем сестра на год старше брата.

Раздел 11.3

1. Порождающие системы обеспечивают унифицированный подход к решению различных задач. Это означает, что хотя исходно задачи могут иметь самую разную формулировку, все они могут быть переформулированы в терминах порождающих систем с приведением их к задаче нахождения пути в графе состояний.
2. 

3. Глубина дерева равна четырем. Верхняя часть дерева выглядит следующим образом.


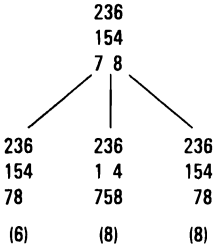
4. Для решения этой задачи потребуется слишком много бумаги и времени.
5. Наша эвристическая система для решения головоломки "Восьмерка" построена на анализе текущей ситуации по методу, который использует альпинист, взбирающийся на гору. Такая близорукость послужила причиной того, что в примере из данного раздела наш алгоритм сначала попытался пойти по неверному пути, подобно тому как альпинист может подвергнуть себя опасности, прокладывая путь по ориентирам только в непосредственной близости от себя. (Благодаря такой аналогии эвристические системы, основанные на локальной или текущей информации, часто называются *системами восхождения*.)
6. Система перемещает фишки 5, 6 и 8 либо по часовой стрелке, либо против часовой стрелки, пока требуемое состояние не будет достигнуто.

7. Проблема здесь заключается в том, что наша эвристическая система игнорирует важность анализа свойств свободного места, расположенного по соседству с фишкой, которая находится не на своем месте. Если свободное место окружено фишками, занимающими правильные позиции, то некоторые из них потребуется предварительно переместить, чтобы иметь возможность переместить те фишки, которые находятся не на своих местах. Таким образом, было бы неправильно рассматривать все окружающие свободное место фишки как занимающие требуемые места. Чтобы устранить этот недостаток, мы должны сначала заметить, что фишку, которая находится на правильном месте, но отделяет свободную позицию от неправильно расположенных фишек, следует переместить с правильной позиции в другое место, а затем возвратить обратно. Таким образом, каждая правильно расположенная фишка на пути между свободным местом и ближайшей неправильно расположенной фишкой находится как минимум в двух шагах от своего окончательного положения. Следовательно, используемый алгоритм вычисления оценочной стоимости можно модифицировать следующим образом.

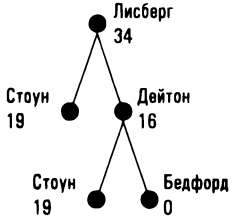
   Сначала надо вычислить оценочную стоимость так, как это делалось раньше. Однако если свободное место совершенно изолировано от неправильно расположенных фишек, следует найти ближайший путь между свободной позицией и любой из неправильно расположенных фишек, умножить число фишек на этом пути на 2 и добавить результат к ее вычисленной ранее оценочной стоимости.

   В этой системе листья дерева, показанного на рис. 10.10, будут иметь оценочную стоимость 6, 6 и 4 (слева направо), поэтому правильная ветка будет выбрана с самого начала работы алгоритма.

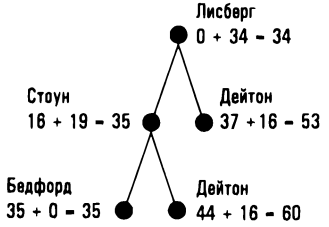
   Тем не менее новая система тоже недостаточно надежна. Например, рассмотрим приведенную ниже конфигурацию. Решение заключается в том, чтобы сдвинуть фишку 5 вниз, повернуть два верхних ряда по часовой стрелке, пока фишки в них не займут правильное положение, вернуть фишку 5 назад и переместить фишку 8 в правильную конечную позицию. Но согласно нашей новой эвристической системе следует начать с фишки 8, поскольку состояние, полученное в результате этого начального хода, имеет стоимость 6, тогда как все остальные состояния — стоимость 8.



8. Решение, найденное по алгоритму поиска по первому наилучшему совпадению, — это путь из Лисберга в Дейтон, а затем в Бедфорд. Однако этот путь не является самым коротким.



9. Найденное решение — путь из Лисберга в Стоун, а затем в Бедфорд. Этот путь самый короткий.



Раздел 11.4

1. Знание реального мира — это информация об окружающей среде, которую человек использует для понимания и рассуждений. Разработка методов представления, хранения и извлечения подобной информации является основной целью исследований в области искусственного интеллекта.
2. В этом случае используется предположение о замкнутости мира.
3. Проблема фреймов — это проблема корректного обновления информации в хранилище знаний машины по мере возникновения тех или иных событий. Задача усложняется тем, что многие события имеют разнообразные косвенные последствия.
4. Подражание, обучение с учителем и обучение с подкреплением. Метод обучения с подкреплением не предполагает непосредственного вмешательства человека.
5. Традиционные методы предполагают создание компьютерной системы в единственном экземпляре. Эволюционные методы предусматривают создание нескольких поколений пробных систем, среди которых со временем может быть обнаружена "хорошая" система.

Раздел 11.5

1. Все варианты приводят к появлению на выходе значения `0`, за исключением варианта `1, 0` — для него выходное значение будет равно `1`.
2. Присвоить каждому входному значению весовой коэффициент, равный `1`, и установить пороговое значение, равное `1,5`.
3. Основная проблема, указанная в тексте, заключается в том, что процесс обучения может зацикливаться, повторяя одну и ту же последовательность корректирующих действий снова и снова.

Раздел 11.6

1. Реактивный подход состоит в том, что вместо полного детального плана предстоящих действий используется набор простых правил, на основании которых и принимаются необходимые решения по мере возникновения тех или иных обстоятельств.
2. Основной смысл этого задания в том, чтобы вы задумались, насколько широка область робототехники. Она охватывает все аспекты искусственного интеллекта, а также многочисленные темы в других областях науки и техники. Главное назначение робототехники состоит в том, чтобы разработать действительно автономные машины, которые смогут самостоятельно двигаться и разумно реагировать на все, что происходит в окружающей их среде.
3. Внутренний контроль и физическая структура.

Раздел 11.7

1. Не существует правильного или неправильного ответа.
2. Не существует правильного или неправильного ответа.
3. Не существует правильного или неправильного ответа.

Глава 12

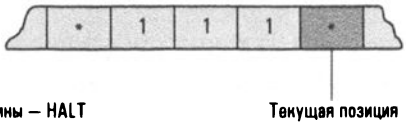
Раздел 12.1

1. Что вы скажете о логических операциях `AND`, `OR` и `XOR`? Фактически в главе 1 при представлении этих функций были использованы именно соответствующие таблицы.

2. Вычисление размера выплат на погашение займа, определение площади круга или расчет пробега автомобиля.
3. Математики называют такие функции *трансцендентными*. Примером таких функций являются логарифмы и тригонометрические функции, которые нельзя вычислить с помощью исключительно алгебраических операций. Например, точные значения тригонометрических функций в действительности можно было бы вычислить только путем рисования соответствующего треугольника, точного измерения длин его сторон и только потом — посредством применения алгебраической операции деления.
4. Одним из примеров является проблема деления угла на три равные части. На практике это означает, что древнегреческие математики не могли построить угол, который был бы точно на одну треть больше заданного угла. Суть в том, что вычислительная система греков в виде линейки и циркуля представляет собой типичный пример системы с ограничениями.

Раздел 12.2

1. В результате получится следующая диаграмма.



2.

Текущее состояние	Содержимое текущей ячейки	Записываемое значение	Направление перемещения	Следующее состояние
СТАРТ	*	*	Влево	СОСТОЯНИЕ 1
СОСТОЯНИЕ 1	0	0	Влево	СОСТОЯНИЕ 2
СОСТОЯНИЕ 1	1	0	Влево	СОСТОЯНИЕ 2
СОСТОЯНИЕ 1	*	0	Влево	СОСТОЯНИЕ 2
СОСТОЯНИЕ 2	0	*	Вправо	СОСТОЯНИЕ 3
СОСТОЯНИЕ 2	1	*	Вправо	СОСТОЯНИЕ 3
СОСТОЯНИЕ 2	*	*	Вправо	СОСТОЯНИЕ 3
СОСТОЯНИЕ 3	0	0	Вправо	ОСТАНОВ
СОСТОЯНИЕ 3	1	0	Вправо	ОСТАНОВ

3.

Текущее состояние	Содержимое текущей ячейки	Записываемое значение	Направление перемещения	Следующее состояние
СТАРТ	*	*	Влево	ВЫЧЕСТЬ
ВЫЧЕСТЬ	0	1	Влево	ЗАНЯТЬ
ВЫЧЕСТЬ	1	0	Влево	НЕ ЗАНИМАТЬ
ЗАНЯТЬ	0	1	Влево	ЗАНЯТЬ
ЗАНЯТЬ	1	0	Влево	НЕ ЗАНИМАТЬ
ЗАНЯТЬ	*	*	Вправо	НУЛЬ
НЕ ЗАНИМАТЬ	0	0	Влево	НЕ ЗАНИМАТЬ
НЕ ЗАНИМАТЬ	1	1	Влево	НЕ ЗАНИМАТЬ
НЕ ЗАНИМАТЬ	*	*	Вправо	ВОЗВРАТ
НУЛЬ	0	0	Вправо	НУЛЬ
НУЛЬ	1	0	Вправо	НУЛЬ
НУЛЬ	*	*	На месте	ОСТАНОВ
ВОЗВРАТ	0	0	Вправо	ВОЗВРАТ
ВОЗВРАТ	1	1	Вправо	ВОЗВРАТ
ВОЗВРАТ	*	*	На месте	ОСТАНОВ

4. Дело в том, что концепция машины Тьюринга была предложена для выражения понятия "вычислить". Это означает, что любую ситуацию, в которой имеет место вычисление, можно описать с помощью компонентов и операций машины Тьюринга. Например, человек, подсчитывающий подоходный налог, выполняет некоторое вычисление. В этом случае машина Тьюринга заменяет собой человека, а ее лента — бумагу, на которой записываются цифры.

5. Машина, описанная нижеследующей таблицей, останавливается, если начинает работу с четного числа, и никогда не останавливается, если начинает работу с нечетного числа.

Текущее состояние	Содержимое текущей ячейки	Записываемое значение	Направление перемещения	Следующее состояние
СТАРТ	*	*	Влево	СОСТОЯНИЕ 1
СОСТОЯНИЕ 1	0	0	Вправо	ОСТАНОВ
СОСТОЯНИЕ 1	1	1	На месте	СОСТОЯНИЕ 1
СОСТОЯНИЕ 1	*	*	На месте	СОСТОЯНИЕ 1

Раздел 12.3

1. ```
   clear AUX
   incr AUX
   while X not 0:
     clear X
     clear AUX
   while AUX not 0:
     incr X
     clear AUX
   ```

2. ```
   while X not 0:
     decr X
   ```

3. ```
   copy X to AUX
   while AUX not 0:
     S1
     clear AUX
   copy X to AUX
   invert AUX (См. ответ на вопрос 1)
   while AUX not 0:
     S2
     clear AUX
   while X not 0:
     clear AUX
     clear X
   ```

4. Если предположить, что X обозначает ячейку памяти с адресом 0X40 и каждый сегмент программы начинается с нулевого адреса (0X00), то можно получить следующую таблицу преобразований.

   ```
   clear X
   incr X
   decr X
   while X not 0:
     .
     .
     .
   ```

5. Как и в реальных машинах, с отрицательными числами можно работать с использованием соответствующей системы кодирования. Например, правый бит в каждой строке можно использовать в качестве знака, а остальные — для представления абсолютной величины числа.

6. Функция умножения на 2.
   ```

Раздел 12.4

1. Да. Фактически эта программа останавливается независимо от начальных значений ее переменных, и поэтому она должна останавливаться, если ее переменные инициализируются значениями, являющимися закодированным представлением программы.
2. Программа останавливается, только если начальное значение `X` заканчивается на `1`. Поскольку ASCII-кодом символа точки с запятой является комбинация `00111011`, закодированная версия программы должна заканчиваться на `1`. Следовательно, программа является самоостанавливающейся.
3. Дело в том, что логика здесь та же, что и в нашем доказательстве утверждения, что проблема остановки не имеет алгоритмического решения. Если маляр покрасит свой дом, то он не имеет права его красить, и наоборот, если он не покрасит свой дом, то он обязан его покрасить.

Раздел 12.5

1. Мы можем лишь заключить, что проблема имеет сложность $\Theta(2^n)$. Если бы мы смогли доказать, что "наилучший алгоритм" решения задачи принадлежит классу $\Theta(2^n)$, то можно было бы утверждать, что эта задача принадлежит классу $\Theta(2^n)$.
2. Как правило, алгоритмы из класса $\Theta(n^2)$ ограничивают алгоритмы из класса $\Theta(2^n)$, но для малых входных значений экспоненциальные алгоритмы часто ограничивают полиномиальные. В действительности экспоненциальный алгоритм иногда бывает предпочтительнее — когда все возможные входные данные являются малыми числами.
3. Дело в том, что число подгрупп растет экспоненциально, и с этой точки зрения задача перечисления всех возможных вариантов быстро становится все более трудоемкой.
4. В класс полиномиальных задач входит задача сортировки списка, которая может быть решена с помощью полиномиального алгоритма, например алгоритма сортировки по методу вставки. К классу неполиномиальных задач относится задача перечисления всех подгрупп, которые могут быть сформированы из данной группы. Любая полиномиальная задача является NP-задачей. Примером NP-задачи является задача о странствующем коммивояжере, однако пока не доказано, что она относится к классу полиномиальных.

5. Нет. Наше использование термина "сложность" относится ко времени, необходимому для выполнения алгоритма, а не к тому, насколько сложно понять этот алгоритм.

Раздел 12.6

1 Единственный возможный вариант 211 × 313 = 66 043. Оба множителя — простые числа.

2. Сообщение `101` является двоичным представлением десятичного числа 5. Тогда $5^e = 5^5 = 15\,625$. Далее вычисляем $15\,625 \pmod{91} = 64$, что в двоичном коде будет выглядеть как `1000000`. Таким образом, зашифрованной версией сообщения является `1000000`.

3. Сообщение `10` является двоичным представлением десятичного числа 2. Тогда $2^d = 2^{29} = 536\,870\,912$. Далее вычисляем $536\,870\,912 \pmod{91} = 32$, что в двоичном коде будет выглядеть как `100000` в двоичной записи. Таким образом, расшифрованной версией сообщения является `100000`.

4. Прежде всего, определяем значение $n = p \times q = 7 \times 19 = 133$. Чтобы найти значение d, необходимо выбрать целое положительное число k, такое, что $k(p - 1)(q - 1) + 1 = k(6 \times 18) + 1 = 108k + 1$ делится без остатка на $e = 5$. Значения $k = 1$ и $k = 2$ нам не подходят, но если $k = 3$, то $108k + 1 = 325$, что делится на 5. Следовательно, частное 65 и является требуемым значением d.

Предметный указатель

A
ANSI, 74; 76
ASCII, 74; 112

B
BIOS, 237

C
CAD, 81
CASE-инструмент, 528
CISC-архитектура, 153
COBOL, 437
CRC-карты, 562

D
Direct3D, 753
DMA, 180
DNS-сервер, 289
DoS-атака, 329
DSL, 182

E
EFI, 237

F
FIFO, 596
Firmware, 237
FORTRAN, 437
FTP, 291

G
GIF, 123
Google, 34
GUI, 230

H
HDMI, 178
HTML, 301
HTML-документ, 302
HTTP, 291; 300
HTTPS, 334

I
IMAP, 294
IP-адрес, 286
IP-телефония, 295
ISO, 75; 77

J
JPEG, 123

L
LIFO, 594
Linux, 229

M
MIDI, 82
MIMD, 200
MP3, 126
MPEG, 125

N
NP-задача, 881; 883
  полная, 883

O
OpenGL, 753

P
POP3, 294
Python, ; 109
  ввод-вывод, 192; 193
  операторы, 113
  пример программы, 195; 115
  управляющие конструкции, 188
  функции, 189

R
RISC-архитектура, 152

S
SISD, 200
SMTP, 292
SSD-диск, 72
SSH, 291
STL, 604

T
TIFF, 125

U
UML, 555
Unicode, 112
URL-адрес, 300
USB, 178
UTF-8, 76

V
VoIP, 295

W
Wi-Fi, 272
World Wide Web, 299

X
XML, 306

A
Абстрактный инструмент, 40
Абстракция, 39
Авторские права, 577
Агент, 776
  восприятие, 777
  декларативные знания, 777
  процедурные знания, 777
Агрегатный тип, 593
Ада Байрон, 32
Адресация
  косвенная, 637
  непосредственная, 637
  прямая, 637
Аксон, 817
Алгоритм, 24; 38; 356
  RSA, 886; 888; 891
  блок-схема, 385
  восходящие/нисходящий методы, 377
  двоичного поиска, 394; 876
  детерминированный, 885
  запись на псевдокоде, 364
  итерационные структуры, 380
  недетерминированный, 882; 885
  последовательного поиска, 380
  поэтапное уточнение, 377
  правильность, 411
  представление, 359
  примитивы, 361
  программа, 24
  процесс решения задач, 372
  рекурсивные структуры, 394
  слияния файлов, 689
  сложность, 874
  создание, 372
  сортировка
    методом вставки, 388
    слиянием, 876; 877
  управление
    рекурсией, 402
    циклами, 382
  фазы Полиа, 373
  формальное определение, 357
  художника, 747
  эффективность, 405; 410; 873
Алиасинг, 745
Анализ выбросов, 699
Анизотропные поверхности, 741
Анимация, 759
  динамика, 762
  кадры, 759
  кинематика, 763
  основные положения, 759
  процесс анимации, 764
  раскадровка, 761
  фазовка, 761
Антивирусная программа, 333
Аплет, 310
Арифметические операции, 175
Архитектура
  Интернета, 282
  компьютера, 148; 199
    CISC-архитектура, 153
    RISC-архитектура, 152
    кеш-память, 151
    контроллеры, 178
    многопроцессорные машины, 200
    регистр команд, 162
    регистры, 148
    счетчик адреса, 162
    шина, 149; 178
Ассемблер, 435
  компонентов, 550
Ассоциативный анализ, 699
Атрибут, 663
Аудитория книги, 10
  преподавателю, 14
  студенту, 17
Аутентификация, 256; 339

Б
Базы данных, 654
  влияние на общество, 701
  защита от сбоев, 682
  значение, 655
  модель, 662
  независимость данных, 661
  обеспечение целостности, 681
  объектно-ориентированные, 677
  распределенные, 660; 661
  реляционные, 663
  система управления, 659
  схема, 658

Байт, 62
биты, 62
Безопасность
администратор, 251
антивирусные программы, 333
аутентификация, 256
брандмауэр, 331
криптография, 334
правовое регулирование, 339
привилегированные команды, 255
программы аудита, 252
прокси-сервер, 332
сертификация, 338
типы атак, 327
учетная запись, 251
цифровая подпись, 339
Библиотека стандартных шаблонов, 604
Биоинформатика, 699
Бит, 52; 59
комбинации двоичных разрядов, 74
четности, 128
Битовая карта, 79
Блок-схема, 385
Ботнет, 330
Брандмауэр, 331
Браузер, 300; 308
Булевы операции, 53
Буфер
глубины, 747
кадра, 725
Бэббидж, 31

В

Ввод-вывод, 192
Веб-сервер, 300; 309
Веб-страница, 302
Вентиль, 55
Верификация
предусловия, 413
программ, 411
утверждения, 414
Взаимная блокировка, 246
Визуальный коридор, 743
Виртуальная память, 234
Вирус, 327
Влияние на общество, 43
Восприятие, 777
Вредоносные программы, 327
ботнет, 330
вирус, 327
троян, 328
червь, 328
шпионское ПО, 328

Г

Генерация кодов, 487
Генетические алгоритмы, 815
Гиперссылка, 299
Гипертекст, 299
Голосовая связь, 295
Граф
состояний, 793
сцены, 735; 752
Графический
адаптер, 751
процессор, 752
Грид-вычисления, 280
Группа обеспечения качества, 565

Д

Данные, 41
базовые структуры, 592
биты, 52
интеллектуальный анализ, 697
представление
звука, 82
изображений, 79
текста, 74
числовых значений, 77
сжатие, 119
скорость передачи, ; 127
спулинг, 248
Двоичная система счисления, 77; 85; 86
Двоичное сложение, 87
в дополнительном коде, 94
Двоичный дополнительный код, 92
Декларативное программирование, 504
Дерево, 596
поиска, 797
представление в памяти, 615
пример реализации, 622
Десятичная система счисления, 85
Диаграмма
вариантов использования, 555
взаимодействия, 560
классов, 556
последовательности, 560
потоков данных, 553
Динамика, 762
Динамическая типизация, 111
Диспетчер, 234; 239; 241; 243

Дистанция Хэмминга, 130
Диффузный свет, 740
Добыча данных, 697
Документация
  пользовательская, 569
  системная, 570
  техническая, 570
Домен, 287
Доменное имя, 288
Доступ к знаниям, 810
Драйвер устройства, 233

Ж

Жесткий диск, 67
  запись информации, 67
  параметры, 68
Жизненный цикл ПО, 531

З

Задание, 220
Задача, 500
  коммивояжера, 881
Запись, 593
  представление в памяти, 607
Затенение, 748
  по методу Гуро, 750
  по методу Фонга, 750
Зеркальный свет, 740
Знаковый разряд, 92
Знания реального мира, 808

И

Идентификатор, 435
Извлечение информации, 790
Индекс, 691
Инкапсуляция, 498
Интегральная схема, 59
Интегрированная
  программная система, 489
  среда разработки, 529
Интеллектуальная собственность, 577
Интерактивная обработка, 222
Интернет, 33; 43; 269; 282; 314
  IP-адрес, 286
  URL-адрес, 300
  World Wide Web, 299
  адресация, 286
  архитектура, 282
  безопасность, 327
  веб-сервер, 300
  голосовая связь, 295
  доменное имя, 288
  домены, 287
  доступ, 286
  инженерный совет, 285
  маршрутизация, 316
  пакеты, 315
  передача сообщений, 313
  поисковые машины, 308
  приложения, 291
  провайдеры, 282
  протоколы, 311
    TCP/IP, 318
  сервер имен, 289
  сетевое ПО, 313
  спам, 330
  узлы, 284
  электронная почта, 292
Интернет-адресация, 286
Интерпретатор, 437
Интерфейс
  пользователя, 230
    настольных компьютеров, 574
    смартфона, 572
  человек–машина, 571
Интранет, 283
Информационный поиск, 789
Искусственный интеллект, 776
  агент, 776
  алгоритм A*, 806
  генетические алгоритмы, 815
  знания реального мира, 808
  инженерный подход, 779
  машинное обучение, 812
  нейронные сети, 816
  обработка языка, 787
  порождающие системы, 793
  последствия, 828
  робототехника, 823
  способность
    к восприятию, 783
    к рассуждению, 792
  теоретические изыскания, 780
  эвристические методы, 799
  экспертная система, 796
Итерационные структуры, 380
Итерация, 367

К

Карта памяти, 73
Каталог, 232
Кеш-память, 151

Кибербезопасность, 327
Кинематика, 763
Класс, 633
Кластеризация, 695
Кластерные вычисления, 280
Кластерный анализ, 698
Клиент, 277
Ключевое слово, 111; 482
Ключи
  дешифрования, 888
  закрытые, 336; 890
  открытые, 336; 890
  шифрования, 888
Когнетика, 572
Код
  ASCII, 75; 902
  CRC, 130
  ISO, 75
  Unicode, 76
  UTF-8, 76
  двоичный дополнительный, 79
  Хоффмана, 120
Кодирование
  длины серий, 119
  метод LZW, 121
  представление текста, 74
Коллизия, 272
Команды
  арифметические и логические, 155; 171
  ввода-вывода, 155; 192
  код операции, 158
  машинный цикл выполнения, 162
  передачи данных, 154
  переменной длины, 155
  поле операндов, 158
  управления, 156
Комментарии, 113; 449; 464
Компакт-диск, 70
Компилятор, 437
Компиляция "на лету", 480
Компонент, 550
Компьютер, 32; 90
  ENIAK, 29
  архитектура, 148
  влияние на общество, 43
  встроенные системы, 226
  выполнение программы, 162
  данные, 41
  загрузка, 236
  конвейерная обработка, 199
  концепция хранимой программы, 150
  многопроцессорный, 200
  операционная система, 220
  основная память, 62
  переносной, 33
  персональный, 32
  производительность, 199
  происхождение, 27
  разностная машина, 31
  смартфон, 33
  тактовая частота, 164
Компьютерная графика, 720
  2D-графика, 721
  3D-графика, 722; 724
  моделирование, 724
  основные положения, 720
  рендеринг, 724
Компьютерные сети, 268
  грид-вычисления, 280
  Интернет, 269; 282
  классификация, 268
  кластерные вычисления, 280
  клиент, 277
  маршрутизатор, 275
  мост, 274
  облачные вычисления, 281
  объединение, 274
  повторитель, 274
  протоколы, 271
  распределенные системы, 280
  сервер, 277
  сетевой коммутатор, 275
  топология, 269
  шлюз, 276
Компьютерный артефакт, 52
Конвейерная обработка, 199
Конвейер рендеринга, 743; 751
Конечная система, 284
Конкатенация, 114
Константа, 457
Конструктор, 495
Контроллер, 178
Контрольный
  байт, 130
  бит, 128
Конфиденциальность, 341; 703
Концепция
  сокрытия информации, 549
  хранимой программы, 151
Кортеж, 663

Криптография, 334
  ключи шифрования, 886
  с открытым ключом, 886; 888
Критическая область, 245; 502
Кроссплатформенное ПО, 438
Куб данных, 700

Л

Лексема, 481
Лексический анализ, 480
Лингвистика, 780
Литерал, 456
Логические операции, 171
  Python, 186
Логический вывод, 504
Логическое программирование, 443; 507

М

Магнитные системы памяти
  дискеты, 69
  диски, 67
  лента, 69
Мантисса, 102; 106
Маршрутизатор, 275
Маршрутизация
  номер порта, 317
Массив, 454; 592
  адресный полином, 605
  представление в памяти, 602
  развертка, 604
Массовая память, 66
  магнитные диски, 67
  флеш-накопители, 71
Материнская плата, 148
Машина Тьюринга, 852
  вычисления, 852
Машинное обучение, 812
Машинный язык, 152; 435
  команды
    арифметические и логические, 155; 171
    ввода-вывода, 155
    передачи данных, 154
    переменной длины, 155
    управления, 156
  типы адресации, 637
Метаданные, 657
Метарассуждения, 810
Метка конца файла, 688
Метод
  захвата движения, 764
  излучательности, 757
  коррекции ошибок, 132
  трассировки лучей, 755
Методология RUP, 539
Метрики, 527
Микропроцессор, 148
Моделирование
  глобальное освещение, 754
  объектов, 726
  отображение текстуры, 733
  оцифровка, 730
  плоские грани, 727
  процедурные модели, 730
  системы частиц, 731
  сцены в целом, 735
Модель
  P2P, 278
  базы данных, 662
  водопада, 538
  итерационная, 539
  клиент/сервер, 277
    пример, 322
  одноранговая, 278
  пошаговая, 538
Модем, 182
Модули, 542
  компоненты, 550
  связанность, 545
  связность элементов, 547
  сокрытие информации, 549
  структурная схема, 542
Модульная арифметика, 887
Модульность, 541
Морфинг, 760
Мост, 274
Мультизадачный режим, 224
Мультипрограммный режим, 224

Н

Наследование, 497
Начальная загрузка, 235
Нейрон
  аксон, 817
  пороговое значение, 817
  синапс, 817
Нейронные сети, 816; 819
  обучение, 820
Нормализованная форма, 104
Нотация
  двоичная, 77; 86
    представление дробей, 89
    с избытком, 97
    с плавающей точкой, 101

десятичная, 85
шестнадцатеричная, 59

О

Область видимости, 469
Облачные вычисления, 281
Обработка
в реальном времени, 223
естественного языка, 780
изображений, 721
языка, 787
контекстный анализ, 789
семантический анализ, 788
Обработчик прерываний, 240
Обучение
без учителя, 813
с подкреплением, 812
с учителем, 812
Объект, 446; 494; 633
конструктор, 495
переменная экземпляра, 492
постоянный, 678
экземпляр класса, 447
Объектно-ориентированная парадигма, 633
Объектно-ориентированное программирование, 445; 491
инкапсуляция, 498
классы, 447; 491
конструкторы, 495
методы, 446; 492
наследование, 497
объекты, 446
полиморфизм, 498
Окно изображения, 725
Оператор, 113
case, 462
for-цикл, 463
if-else, 461
switch, 462
while, 461
арифметический, 114
булев, 114
выполняемый, 449
комментарии, 449; 464
конкатенация, 114
объявления, 449
функции, 469
присваивания, 111; 458
репликация, 114
структуры ветвления, 461
управляющий, 459
Операции
арифметические, 175
логические, 171
Python, 186
сдвига, 174
Python, 188
Операционная система, 220
Linux, 229
Microsoft Windows, 244
администратор, 251
архитектура, 227
балансировка загрузки, 225
безопасность, 250
программы аудита, 252
уровни привилегий, 255
задание, 220
интерактивная обработка, 222
компоненты, 230
масштабирование, 225
многоядерная, 249
мультизадачный режим, 224
мультипрограммный режим, 224
начальная загрузка, 234
пакетная обработка, 221
прерывания, 240
процесс, 238
разделение времени, 239
управление памятью, 233
файловая система, 232
ядро, 232; 234
Оптимизация кода, 488
Оптические системы, 70
формат записи, 70
Основная память, 62
адресация, 63
доступ, 64
Отладка, 116; 434
Отношение, 663
Отображение текстуры, 733
Оцифровка, 730
Очередь, 596
заданий, 221
представление в памяти, 613
циклическая, 615
Ошибки
времени выполнения, 117
округления, 105
при передаче информации, 128

семантические, 117
синтаксические, 116

П

Пакет, 315
счетчик переходов, 320
Пакетная обработка, 221
Память
емкость, 65
магнитные системы, 67
массовая, 66
оптические системы, 70
основная, 62
постоянная, 235
прямой доступ, 180
триггер, 55
ячейки, 62
Папка, 232
Парадигма программирования, 440
декларативная, 443
императивная, 443; 543
объектно-ориентированная, 445; 543
функциональная, 444
Пароли, 253
Перегрузка операций, 459
Передача данных, 127
Переменные, 111
глобальные, 469
локальные, 469
область видимости, 469
Переполнение, 96
Перфокарта, 28
Пиксель, 79
Планировщик, 234; 243
управление процессами, 239
Плоский файл, 654
Поведенческий интеллект, 801
Поверхности Безье, 729
Повторитель, 274
Подсхема, 658
Поиск в глубину, 799
Поисковые машины, 308
Полигональная сетка, 728
Полиморфизм, 498
Порождающая система, 792
Порождение, 793
Последовательный файл, 686
Поток, 500
Потоковое мультимедиа, 296
Почтовый сервер, 292
Правила вывода, 795
Предикат, 507
Представление
дробей, 89; 101
звука, 82
знаний, 809
изображений, 79
текста, 74
числовых значений, 77
целых чисел, 91
Прерывание, 240; 242
Приведение типов, 487
Приложения CAD, 81
Примитивы, 361
Принцип
модульности, 541
Парето, 567
Проблема
остановки, 865
неразрешимость, 868
скрытого терминала, 272
фреймов, 811
Провайдер
доступа к Интернету, 283
Интернета, 282
Проверка на четность, 129
Программа, 24
антивирусная, 333
ассемблер, 435
браузер, 300
верификация, 411
вредоносная, 327
вывод результата, 110
выполнение, 162; 165
машинный цикл, 162
жизненный цикл, 531
идентификаторы, 435
интерпретатор, 437
исходная, 479
комментарии, 113
компилятор, 437
машинная независимость, 439
начальной загрузки, 235
объектный код, 479
отладка, 116
параллельная обработка, 500
переменные, 111
самоостанавливающая, 867

тестирование, 118
транслятор, 437
трансляция, 479
этапы разработки, 532
Программирование, 42; 109
выражения, 456
декларативное, 504
константы, 457
литералы, 456
мониторы, 502
основные концепции, 448
парадигмы, 440
типы данных, 112
трансляция программы, 479
универсальный язык, 857
управляющие операторы, 459
Программист, 536
Программное обеспечение, 227
технология разработки, 526
Проектирование ПО
инструменты, 553
Проекционная плоскость, 724
Прокси-сервер, 332
Пропускная способность, 184
Пространство состояний, 793
Протокол
CSMA/CA, 272
CSMA/CD, 271
FTP, 291
HTTP, 291; 300
HTTPS, 334
IMAP, 294
IP, 318
MIME, 294
NNTP, 291
POP3, 294
SMTP, 292
SSH, 291
SSL, 334
TCP, 318; 319
TLS, 335
UDP, 318; 320
VoIP, 295
Wi-Fi, 272
блокировки, 684
Интернета, 311
принудительной отмены-ожидания, 685
сетевой, 271
фиксации/отката изменений, 681
Процедура, 191; 476; 477
Процедурная модель, 730
Процесс, 238; 501
взаимная блокировка, 246
методы взаимодействия, 277
семафоры, 243
Процессор
многоядерный, 185; 200
тактовая частота, 164
Псевдокод, 363
итерация, 367
нотация Python, 364
обобщенные имена, 370
функции, 368
циклы, 382
Путь доступа, 232

P

Разделение времени, 223; 239; 241
Разработка ПО
альфа-тестирование, 568
анализ требований, 533
бета-тестирование, 568
гибкие методы, 540
документирование, 569
лицензирование, 577
методологии, 538
модульность, 541
на основе компонентов, 550
обеспечение качества, 564
проектирование, 535
реализация, 536
модулей, 542
с открытым исходным кодом, 540
тестирование, 537; 566
шаблоны проектирования, 562
экстремальное программирование, 540
Раскадровка, 761
Распознавание изображений, 784
анализ изображения, 787
обработка изображения, 785
Распределенные системы, 280
Рассеянный свет, 741
Рассуждения, 792
Растеризация, 745
Реализация модулей, 542
Регистр
команд, 162
общего назначения, 148; 149
Резольвента, 505

Рекурсия, 394; 916
  управление, 402
  условие окончания, 402
Рельефное текстурирование, 750
Реляционная модель, 663
  проектирование, 664
Реляционные операции, 668
Рендеринг, 724; 739
  z-буфер, 747
  алиасинг, 745
  затенение, 748
  отражение света, 739
  отсечение, 743
  построчное сканирование, 745
  потоковый, 743
  преломление света, 742
  тени, 753
  удаление невидимых поверхностей, 746
Репликация, 114
Робот, 823
  автономный, 824
  команды, 825
  реактивный, 825
Робототехника, 823
  эволюционная, 826

C

Самосоотносимость, 866
Сборка мусора, 601
Связанность, 545
Связность, 547
Семантическая ошибка, 117
Семантическая сеть, 790
Семафор, 243
Сервер, 277
  имен, 289
Сервлет, 310
Сетевой
  коммутатор, 269; 275
  концентратор, 270
Сети доставки контента, 298
Сеть из сетей, 275
Сжатие
  аудио и видео, 125
  данных, 119
    методы, 119
    с потерями, 119
  изображений, 122
Синапс, 817
Синтаксическая
  диаграмма, 482
  ошибка, 116
Система частиц, 731
Системный аналитик, 536
Скорость передачи данных, 183
Словарное кодирование, 120
Словарь данных, 554
Сложность задач, 872
  временная, 873; 875
  измерение, 873
  нотация "О" большое, 876
  пространственная, 873
  разумное время решения, 880
Смартфон, 33; 225; 551
  искусственный интеллект, 790
  программыное обеспечение, 503
Сопровождение ПО, 531
Спам, 330
  фильтрация, 332
Спецификация требований к ПО, 534
Список, 593
  непрерывный, 609
  нулевой указатель, 609
  представление в памяти, 608
  связанный, 609
Спулинг, 248
Спуфинг, 331
Стандарт
  DSL, 182
  Ethernet, 271
  HDMI, 178
  USB, 178
Стек, 594
  механизм возврата, 594
  представление в памяти, 612
Структура, 593
Структурное программирование, 460
Структуры данных
  абстракция, 599
  базовые, 592
  динамические, 600
  манипулирование, 619
  реализация, 602
  статические, 600
  указатели, 600
СУБД, 659
  механизм блокировок, 683
  откат, 682
  точка фиксации, 682
Схема, 658

Сцена, 726
  объекы, 726
Сценарий, 110; 451
  пример, 115
Счетчик адреса, 162; 600

Т

Тактовая частота, 164
Тезис Черча–Тьюринга, 855; 863
Текстовый
  процессор, 77
  редактор, 77
Темы компьютерных наук, 37
Тестирование, 118; 566
  альфа-тестирование, 568
  анализ граничных условий, 567
  бета-тестирование, 568
  основных путей, 567
Тест Тьюринга, 781
Тета-классы, 410
Тип
  данных, 112; 451
    Boolean, 453
    character, 452
    integer, 451
    real, 451
    абстрактный, 630
    встроенный, 453
    запись, 455
    определяемый пользователем, 628
    структура, 454
  задачи
    NP-задачи, 881
    полиномиальный, 879
    экспоненциальный, 880
Точка доступа, 270
Транслятор, 437; 479
Трансляция, 479
  генерация кодов, 487
  ключевые слова, 482
  лексический анализ, 480
  оптимизация кода, 488
  приведение типов, 487
  синтаксический анализ, 481; 483
  таблица символов, 486
Триггер, 55
Троян, 328

У

Указатель, 600; 612
  в машинном языке, 635
Управление
  памятью, 233
  циклами, 382
Устройство
  слово состояния, 181
Утечка памяти, 601
Утилиты, 228
Учетная запись, 251

Ф

Фазовка, 761
Файл
  индексированный, 690
  метка конца файла, 688
  последовательный, 686
  текстовый, 76; 686
  хешированный, 692
Файловая система, 232
Фишинг, 329
Флеш-накопители, 71
Флеш-память, 71
  SSD-диски, 72
  карты памяти, 73
  накопители, 72
Формат
  GIF, 123
  JPEG, 123
  Lab, 80
  MP3, 126
  MPEG, 125
  RGB, 80
  TIFF, 125
  векторный, 81
  оцифровка звука, 82
  растровый, 79
  с плавающей точкой, 79; 101
Фрактал, 732
Функция, 189; 468; 476; 848
  вызов, 189
  вычисление, 848
  невычислимая, 850; 865; 871
  отношение ограничения, 879
  параметры, 470; 473
  событийно-управляемая, 478

Х

Хеширование, 692
  кластеризация, 695

Хеш-таблица, 693
Хеш-файл, 693
Хеш-функция, 693
Хранилище данных, 698

Ц

Центр
  проекции, 724
  сертификации, 338
Центральный процессор
  структура, 148
Цикл, 382
  for, 463
  с постпроверкой, 386
  с предпроверкой, 386
  условие окончания, 383
Цифровая подпись, 339

Ч

Червь, 328
Чип, 59
Численный анализ, 106
Числовые данные, 77
  тип представления, 78

Ш

Шаблон проектирования, 562
Шестнадцатеричная нотация, 59
Шина, 149; 178
Шифрование
  с открытым ключом, 336
  с симметричным ключом, 335
Шлюз, 276
Шпионское ПО, 328

Э

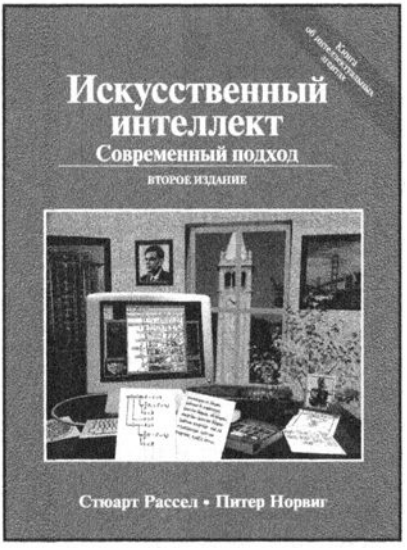
Эволюционное программирование, 816
Эвристика, 800
Эвристические методы, 799
Экспертные системы, 796
Экстремальное программирование, 540
Электронная почта, 292
  почтовый сервер, 292
Электронная схема, 904
Эргономика, 572
Эффективность
  алгоритма, 405
  тета-классы, 410; 873

Я

Ядро ОС, 232
  обработчик прерываний, 240
Язык
  Ada, 448; 500; 912
  Alice, 467
  C, 448; 472; 913
  C#, 448; 480; 497; 914
  C++, 448; 497; 913
  COBOL, 437
  FORTRAN, 437; 448; 914
  HTML, 301
  Java, 448; 480; 497; 500; 915
  Perl, 451
  Prolog, 507
  Python, 109; 247; 450; 472; 477
    операторы, 113
    списки, 595
  SPARK, 416; 913
  SQL, 673
  Swift, 465
  Visual Basic, 451; 465
  Vole, 157; 908
    команды, 909
  XML, 306
  ассемблера, 435
  визуального программирования, 467
  грамматика языка, 482
  интерпретирующий, 109
  ключевые слова, 111
  комментарии, 113
  машинно-независимый, 436
  машинный, 152
  моделирования UML, 555
  простейший, 857; 916
    операторы, 858
    программирование, 861
    универсальность, 863
    управляющая структура, 859
  разметки, 307
  сценариев, 451
  типы данных, 112; 451
  универсальный, 857
Языки
  естественные, 438
  формальные, 438
Ячейка памяти, 62
  адрес, 63

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ СОВРЕМЕННЫЙ ПОДХОД
ВТОРОЕ ИЗДАНИЕ

С. Рассел,
П. Норвиг



www.dialektika.com

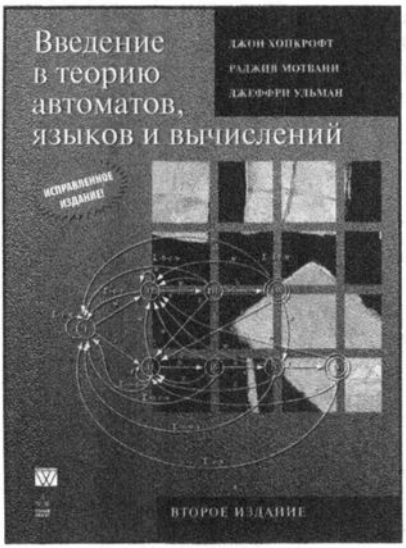
В книге представлены все современные достижения и изложены идеи, которые были сформулированы в исследованиях, проводившихся в течение последних пятидесяти лет, а также собраны на протяжении двух тысячелетий в областях знаний, ставших стимулом к развитию искусственного интеллекта как науки проектирования рациональных агентов. Теоретическое описание иллюстрируется многочисленными алгоритмами, реализации которых в виде готовых программ на нескольких языках программирования находятся на сопровождающем веб-сайте. Книга предназначена для использования в базовом университетском курсе или в последовательности курсов по специальности. Применима в качестве основного справочника для аспирантов, специализирующихся в области искусственного интеллекта, а также будет небезынтересна профессионалам, желающим выйти за пределы избранной ими специальности. Благодаря кристальной ясности и наглядности изложения вполне может быть отнесена к лучшим образцам научно-популярной литературы.

ISBN 978-5-907114-65-4

в продаже

ВВЕДЕНИЕ В ТЕОРИЮ АВТОМАТОВ, ЯЗЫКОВ И ВЫЧИСЛЕНИЙ
2-е издание

Джон Хопкрофт,
Раджив Мотвани,
Джеффри Ульман



www.williamspublishing.com

Книга известных американских ученых посвящена теории автоматов и соответствующих формальных языков и грамматик — как регулярных, так и контекстно-свободных. Во второй части рассматриваются концепции теории вычислений и различные машины Тьюринга, при помощи которых формализуются понятия разрешимых и неразрешимых проблем, а также определяются функции временной и емкостной оценки сложности алгоритмов. Изложение ведется строго, но доступно, и сопровождается многочисленными примерами, а также задачами для самостоятельного решения. Книга будет полезна читателям различных категорий — студентам, аспирантам, научным сотрудникам, преподавателям высших учебных заведений, а также всем, кто интересуется математическими основами современной вычислительной техники.

ISBN 978-5-8459-1969-4 **в продаже**

ГЛУБОКОЕ ОБУЧЕНИЕ
ГОТОВЫЕ РЕШЕНИЯ

Давид Осинга



www.williamspublishing.com

Благодаря готовым примерам, приведенным в книге, вы научитесь решать задачи, связанные с классификацией и генерированием текста, изображений и музыки. В каждой главе описывается несколько решений, объединяемых в единый проект, например приложение, реализующее тренировку музыкальной рекомендательной системы. Также имеется глава с описанием методик, которые в случае необходимости помогут выполнить отладку нейронной сети. Основные темы книги:

- использование векторных представлений слов для вычисления схожести текстов;
- построение рекомендательной системы фильмов на основе ссылок в Википедии;
- визуализация внутренних состояний нейронной сети;
- создание модели, рекомендующей эмодзи для фрагментов текста;
- повторное использование предварительно обученных сетей для создания службы обратного поиска изображений;
- генерирование пиктограмм с помощью генеративно-состязательных сетей (GAN), автокодировщиков и рекуррентных сетей (RNN);
- распознавание музыкальных жанров и индексирование коллекций песен.

ISBN: 978-5-907144-50-7

в продаже

ЛУЧШЕЕ РУКОВОДСТВО ДЛЯ ОВЛАДЕНИЯ ОСНОВАМИ КОМПЬЮТЕРНЫХ НАУК!

Эта книга написана как для студентов, выбравших компьютерные науки своей профессией, так и для учащихся, специализирующихся в любых других дисциплинах. Широкий охват материала вместе с четким изложением делает ее доступной для читателей с любым базовым уровнем. Назначение этой книги — всестороннее представление о предмете компьютерных наук, охватывающее все его аспекты, от сугубо практических до полностью абстрактных. Такой подход к изучению базовых понятий открывает студентам любых, необязательно компьютерных дисциплин всю широту предмета и позволяет получить общее представление о тех возможностях, которые доступны им в современном технократическом обществе. Изложение материала ведется от простого к сложному, от конкретных аспектов к абстрактным, и каждая рассматриваемая тема непосредственно подводит к следующей. Тем не менее отдельные главы и разделы книги достаточно независимы и вполне могут рассматриваться как самостоятельные единицы.

Большое преимущество этой книги — наличие около 1000 заданий и упражнений, предназначенных для закрепления понимания основных излагаемых концепций, а также обсуждение этических и юридических аспектов рассматриваемых технологий, которые необходимо знать, чтобы использовать их безопасно и ответственно. Очень интересны также подборки общественных и социальных вопросов, приведенных в конце каждой главы, назначение которых — заставить читателя задуматься о связях между излагаемым материалом и тем обществом, в котором они живут. Важной особенностью данного, тринадцатого издания является переход к использованию языка Python для записи примеров кода и псевдокода.

ОБ АВТОРАХ

Дж. Гленн Брукшир, заслуженный профессор в отставке университета Маркетт, в котором он многие годы преподавал курсы *Формальные языки, Введение в компьютерные науки* и *Теория вычислений*. Гленн Брукшир является автором всех предыдущих изданий книги *Компьютерные науки. Базовый курс*.

Деннис Брилов, доцент отделения математики, статистики и компьютерных наук университета Маркетт с 2005 г., недавно получил высшую педагогическую награду университета — премию Teaching Excellence Award. Привлечен в качестве соавтра при подготовке 12- и 13-го изданий этой книги.

Категория: компьютерные науки/основы информатики
Уровень: начальный-средний

ISBN 978-5-907144-63-7

Диалектика
www.dialektika.com

Pearson
www.pearson.com