Министерство образования и науки Российской Федерации

ГОУ ВПО "Тамбовский государственный технический университет"

**Ю.Ю. Громов, О.Г. Иванова, Н.А. Земской, А.В. Лагутин,
В.М. Тютюнник, В.Н. Точка, Н.Г. Шахов**

ИНФОРМАТИКА

КУРС ЛЕКЦИЙ

Часть I

Допущено Учебно-методическим объединением вузов
по университетскому политехническому
образованию в качестве учебного пособия для студентов
высших учебных заведений, обучающихся
по специальности 230201 "Информационные системы и технологии"

Тамбов
Издательство ТГТУ
2007

УДК 004.42(075)
ББК 81я73
И741

Рецензенты:

Доктор физико-математических наук, профессор

Е.Ф. Кустов

Доктор физико-математических наук, профессор
В.Ф. Крапивин

И741 Информатика : курс лекций / Ю.Ю. Громов, О.Г. Иванова, Н.А. Земской, А.В. Лагутин, В.М. Тютюнник, В.Н. Точка, Н.Г. Шахов. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2007. – Ч. I. – 364 с. – 100 экз. – ISBN 978-5-8265-0635-6.

Рассмотрены вопросы аппаратного и программного обеспечения современной вычислительной техники, а также алгоритмизации и программирования.

Утверждено Ученым советом университета в качестве учебного пособия для студентов высших учебных заведений, обучающихся по специальностям 230201 – Информационные системы и технологии, 090105 – Комплексное обеспечение информационной безопасности автоматизированных систем, и для студентов среднего профессионального образования, обучающихся по специальности 230105 – Программное обеспечение вычислительной техники и автоматизированных систем.

УДК 004.42(075)
ББК 81я73

ISBN 978-5-8265-0635-6

Ю.Ю. ГРОМОВ, О.Г. ИВАНОВА, Н.А. ЗЕМСКОЙ,
А.В. ЛАГУТИН, В.М. ТЮТЮННИК, В.Н. ТОЧКА, Н.Г. ШАХОВ

ИНФОРМАТИКА

Часть I

Учебное издание

ГРОМОВ Юрий Юрьевич,
ИВАНОВА Ольга Геннадьевна,
ЗЕМСКОЙ Николай Александрович,
ЛАГУТИН Андрей Владимирович,
ТЮТЮННИК Вячеслав Михайлович,
ТОЧКА Владимир Николаевич,
ШАХОВ Николай Гурьевич

ИНФОРМАТИКА

КУРС ЛЕКЦИЙ

Часть I

Редактор Т.М. Глинкина
Компьютерное макетирование Е.В. Кораблевой
Корректор О.М. Ярцева

Подписано в печать 31.10.07
Формат 60 × 84/16. 21,16 усл. печ. л. Тираж 100 экз. Заказ № 687

Издательско-полиграфический центр
Тамбовского государственного технического университета
392000, Тамбов, Советская, 106, к. 14

ПРЕДИСЛОВИЕ

В пособии приводятся общие сведения и характеристики процессов сбора, передачи, обработки и накопления информации; технические и программные средства реализации информационных процессов; модели решения функциональных и вычислительных задач; алгоритмизация и программирование; языки программирования высокого уровня; программное обеспечение и технологии программирования; локальные и глобальные сети ЭВМ; основы защиты информации.

Учебное пособие состоит из двух частей. Первая часть включает в себя вопросы, связанные с аппаратным, программным обеспечением современной вычислительной техники, а также алгоритмизации и программирования. Вторая часть включает такие разделы информатики, как представление данных (структура данных, файловые структуры, структуры баз данных), искусственный интеллект и теория вычислений.

В пособии учитывается динамическая связь развития современных технических и программных средств, что отражено в преемственности с другими профилирующими дисциплинами – "Архитектура ЭВМ", "Операционные системы", "Компьютерные сети", "Алгоритмы и структура данных", "Языки программирования", "Технология программирования и разработка программного обеспечения", "Базы данных", "Разработка и эксплуатация информационных систем".

Пособие содержит материалы как для аудиторной работы, так и для самостоятельной работы студентов с последующим анализом на практических занятиях. В конце каждого раздела учебного пособия приводятся вопросы для самопроверки. Ответы на предлагаемые вопросы вынесены в конец каждой главы. Помимо вопросов для самопроверки, каждая глава содержит подборку упражнений, относящихся к содержанию всей главы, разработанных для закрепления пройденного материала и использования для домашних и практических занятий.

Обширный материал пособия предоставляет преподавателю возможность свободной и гибкой организации процесса обучения с учетом индивидуальных особенностей как группы, так и отдельных студентов. Обсуждаемый материал расширяется дополнительными аспектами, даются ссылки на материал, обсуждение которого будет проводиться позднее. Разделы глав, помеченные звездочкой, содержат дополнительный материал для использования на факультативных или индивидуальных занятиях.

ВВЕДЕНИЕ

Термин "информация" происходит от латинского informatio, что в переводе означает изложение, разъяснение. В обыденной жизни под этим словом понимают сведения, передаваемые людьми устным, письменным или другим образом. В научных и официальных источниках этот термин трактуется по-разному. Так, например, ст. 2 Федерального закона от 20 февраля 1995 г. № 24-ФЗ "Об информации, информатизации и защите информации" (с изменениями от 10 января 2003 г.) дает следующие определения терминов, связанных с информацией:

− *информация* – сведения о лицах, предметах, фактах, событиях, явлениях и процессах независимо от формы их представления;

− *информатизация* – организационный социально-экономический и научно-технический процесс создания оптимальных условий для удовлетворения информационных потребностей и реализации прав граждан, органов государственной власти, органов местного самоуправления, организаций, общественных объединений на основе формирования и использования информационных ресурсов;

− *документированная информация* (документ) – зафиксированная на материальном носителе информация с реквизитами, позволяющими ее идентифицировать;

− *информационные процессы* – процессы сбора, обработки, накопления, хранения, поиска и распространения информации;

− *информационная система* – организационно упорядоченная совокупность документов (массивов документов) и информационных технологий, в том числе с использованием средств вычислительной техники и связи, реализующих информационные процессы;

− *информационные ресурсы* – отдельные документы и отдельные массивы документов, документы и массивы документов в информационных системах (библиотеках, архивах, фондах, банках данных, других информационных системах).

В наиболее общем виде понятие информации можно выразить следующим образом. Информация – это отражение предметного мира с помощью знаков и сигналов.

В теории информации под этим термином понимается такое сообщение, которое содержит факты, неизвестные ранее потребителю и дополняющие его представление об изучаемом или анализируемом объекте (процессе, явлении). Другими словами, информация – сведения, которые должны снять в той или иной степени существующую у потребителя до их получения неопределенность, расширить его понимание объекта полезными (для потребителя) сведениями. По Шеннону, информация – это снятая неопределенность.

Наряду с информацией часто употребляется понятие "данные". Данные могут рассматриваться как признаки или записанные наблюдения, которые по каким-то причинам не используются, а только хранятся. В том случае, если появляется возможность использовать эти данные для уменьшения неопределенности о чем-либо, данные превращаются в информацию. Поэтому можно утверждать, что информацией являются используемые данные.

В процессе обработки информация может менять структуру и форму. Признаками структуры являются элементы информации и их взаимосвязь. Различают содержательную и формальную структуры.

Содержательная структура естественно ориентирована на содержание информации, а формальная – на форму представления информации. Формы представления информации также различны. Основные из них – символьная (основанная на ис-

пользовании символов – букв, цифр, знаков), текстовая (использует тексты – символы, расположенные в определенном порядке), графическая (различные виды изображений), звуковая.

В зависимости от области знаний различают научную, техническую, производственную, правовую и другую информацию. Каждый из видов информации имеет свои особые смысловые нагрузку и ценность, требования к точности и достоверности, преимущественные технологии обработки, формы представления и носители (бумажные, магнитные и др.).

Информация – это неубывающий ресурс жизнеобеспечения. Деятельность отдельных людей, групп, коллективов и организаций сейчас все в большей степени начинает зависеть от их информированности и способности эффективно использовать имеющуюся информацию. Прежде чем предпринять какие-то действия, необходимо провести большую работу по сбору и переработке информации, ее осмыслению и анализу.

Отыскание рациональных решений в любой сфере требует обработки больших объемов информации, что подчас невозможно без привлечения специальных технических средств. Возрастание объема информации особенно стало заметно в середине XX в. Лавинообразный поток информации хлынул на человека, не давая ему возможности воспринять эту информацию в полной мере. В ежедневно появляющемся новом потоке информации ориентироваться становилось все труднее. Подчас выгоднее стало создавать новый материальный или интеллектуальный продукт, нежели вести розыск аналога, сделанного ранее.

Увеличение информации и растущий спрос на нее обусловили появление отрасли, связанной с автоматизацией обработки информации – информатики.

Термин "информатика" (informatics) введен французскими учеными примерно в начале 70-х годов и означал "наука о преобразовании информации". В 1963 г. советский ученый Ф.Е. Темников одновременно с зарубежными авторами определяет информатику как науку об информации вообще, состоящую из трех основных частей – теории информационных элементов, теории информационных процессов и теории информационных систем. Это был первый важный поворот в судьбе понятия "информатика". Он оставался долго лишь историческим фактом. Попытка обосновать новое понятие, доказать его необходимость не была успешной и в должной мере не оценена в силу того, что публикация была осуществлена в мало известном, специальном журнале ("Известия вузов. Электромеханика", 1963, № 11). Так или иначе тогда понятие "информатика" еще не получило в нашей стране заметного распространения. Хотя в научной литературе уже в этот период часто встречались трактовки "информатики через призму взглядов Темникова". Так, в 1968 г. напечатана работа А.И. Михайлова, А.И. Черного и Р.С. Гиляровского "Основы информатики", в которой подробно рассмотрены понятия научно-технической информации и методы ее обработки.

Французский же вариант этого термина постепенно приобретал все большую популярность, чему, несомненно, способствовал тот факт, что Франция становилась одним из лидеров в области развития информационной технологии и техники.

На Международном конгрессе в Японии в 1978 г. дается широкое определение информатики. Вот это определение: "Понятие информатики охватывает области, связанные с разработкой, созданием, использованием и материально-техническим обслуживанием систем обработки информации, включая машины, оборудование, математическое обеспечение, организационные аспекты, а также комплекс промышленного, коммерческого, административного, социального и политического воздействия".

В 1982 г. выходит монография академика В.М. Глушкова "Основы безбумажной информатики". А год спустя годичное Общее собрание Академии наук СССР принимает решение о создании отделения информатики. С этого момента идеи информатики получили прописку не только в науке, но также и среди специалистов-практиков.

Каково тогда было понимание информатики? В названной монографии академика В.М. Глушкова нет прямого определения информатики как новой науки. Но исходя из содержания этой книги и материалов АН о создании нового отделения, можно сделать следующий вывод: информатика – это совокупность средств всей современной информационной теории, техники и технологии, суммарное, комплексное обозначение этой области знаний. По-другому говоря, информатика как наука вбирает сегодня самые разные по своей сущности и природе информационные идеи, средства и процессы, связанные с удовлетворением информационных потребностей общества в настоящем и будущем.

1. ХРАНЕНИЕ ДАННЫХ

Данные в современных компьютерах представляются в виде комбинации двоичных цифр 0 и 1, по-английски – *binary digit* или сокращенно *bit* (*бит*). Смысловое значение комбинаций двоичных цифр изменяется от одного приложения к другому. Для запоминания двоичных цифр (бит) машине требуется некоторое устройство, которое может пребывать в одном из двух состояний. Такими устройствами могут быть переключатель (включен/выключен), реле (открыто/закрыто), флажок на флагштоке (поднят/опущен) и т.д., в которых одно состояние представляет значение 0, а другое – значение 1. В главе рассмотрены способы хранения значений двоичных разрядов в современных машинах и представление данных с помощью комбинации битов.

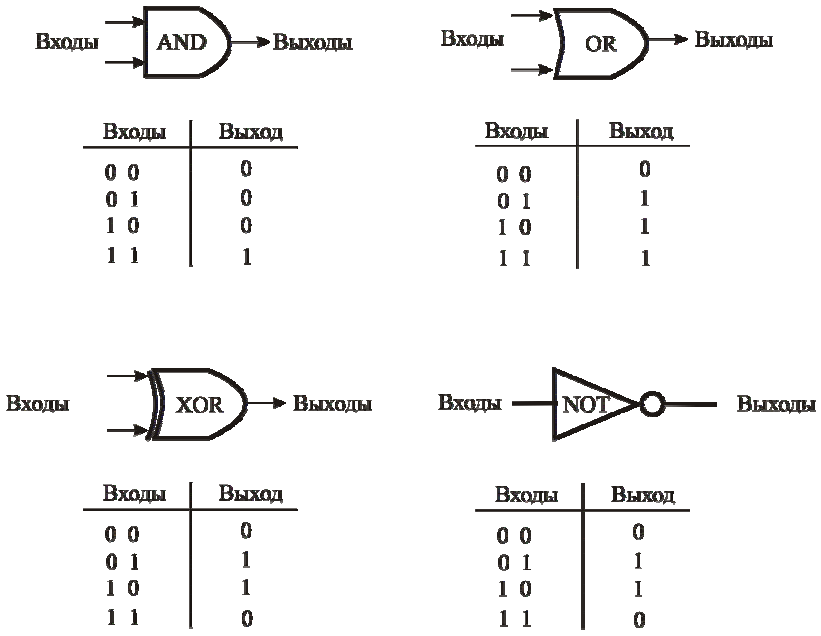
1.1. ОСНОВНАЯ ПАМЯТЬ

В компьютере для хранения данных используется большой набор электронных схем, каждая из которых способна запомнить одну двоичную цифру (бит). Это хранилище битов принято называть *основной* (или *оперативной*) *памятью* (main memory). Электронные схемы основной памяти реализуются в современных вычислительных машинах на базе триггеров или конденсаторов.

Вентили, триггеры и конденсаторы. Устройство, которое выдает результат булевой операции (см. Приложение А), после введения 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 цепи, в которых цифры 0 и 1 представляются разными уровнями электрического напряжения. Во многих случаях вполне достаточно представить вентили в их символической форме, как это показано на рис. 1.1. Обрати-

те 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.

Вентили, подобные показанным на рис. 1.1, представляют собой строительные блоки, из которых конструируются компьютеры. Один важный этап этого направления представлен в электрической схеме, показанной на рис. 1.2. Это один из возможных вариантов схем определенного класса, называемых триггерами. *Триггер* – это схема, которая постоянно выдает выходное значение 0 или 1; оно не меняется до тех пор, пока одиночный импульс от другой схемы не переведет ее в противоположное состояние. Другими словами, выходное значение будет переключаться из одного состояния в другое только под воздействием внешних стимулов.



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

Входы	Выход
0 0	0
0 1	1
1 0	1
1 1	0

Рис. 1.1. Схематическое представление вентилей AND, OR, XOR, NOT и таблицы их входных и выходных данных

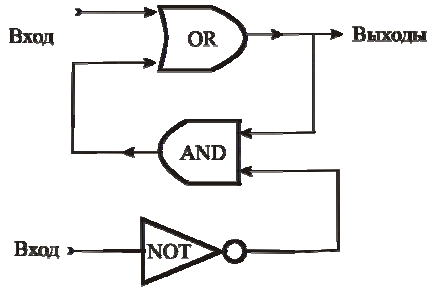
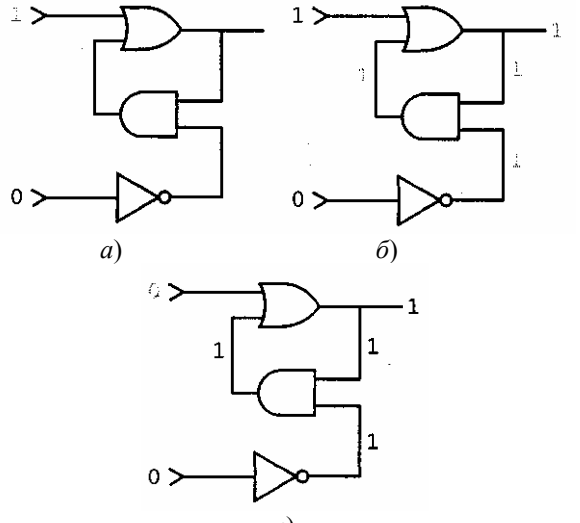


Рис.1.2. Схема простого триггера

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.



а) *б*) *в*)

Рис. 1.3. Установка входного значения триггера равного 1:

а – единица поступает на верхний вход; *б* – это вызывает появление единицы на выходе вентиля OR, что, в свою очередь, вызывает появление единицы на выходе вентиля AND; *в* – наличие единицы на выходе вентиля AND удерживает вентиль OR от изменения его состояния и после снятия единичного сигнала с верхнего входа

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. 1.3, *а*). Это приведет к тому, что выходное значение вентиля OR станет равно 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 равно 1, а это значит, что на втором входе вентиля OR также появится значение 1 (рис. 1.3, *б*). Это гарантирует, что выходное значение вентиля OR останется равным 1 даже в том случае, если значение на верхнем входе триггера вновь станет равно 0 (рис. 1.3, *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.

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.

Для нас значение триггерной схемы состоит в том, что она является идеальным механизмом для хранения двоичных данных (битов) внутри компьютера. Величина, сохраняемая в триггере, определяется его выходным значением. Другие схемы легко могут изменять это значение, посылая импульсы на входы триггера. Подобным же образом другие схемы могут реагировать на хранимое в триггере значение посредством использования выходного значения триггера как одного из своих входных значений.

Конечно, существуют и другие варианты построения триггеров. Один из них изображен на рис. 1.4. Если поэкспериментировать с этой схемой, то можно обнаружить, что, несмотря на совершенно иную внутреннюю структуру, ее внешние свойства полностью аналогичны свойствам схемы, представленной на рис. 1.2. Это первый пример большого значения абстрактных инструментов. При разработке схемы триггера инженер рассматривает несколько альтернативных способов его построения с использованием вентилей в качестве компоновочных блоков. Как только триггеры и другие базовые схемы будут разработаны, инженер сможет использовать их в качестве строительных блоков для создания более сложных схем. Таким образом, разработка общей схемы компьютера приобретает иерархическую структуру, в которой на каждом уровне в качестве абстрактных инструментов используются компоненты, созданные на предыдущих уровнях.

Другим современным средством запоминания битов является конденсатор. Он состоит из двух маленьких металлических пластин, расположенных параллельно друг другу на небольшом расстоянии. Если положительный полюс источника напряжения соединить с одной пластиной конденсатора, а отрицательный – с другой, то электрические заряды из этого источника равномерно распределятся по пластинам.

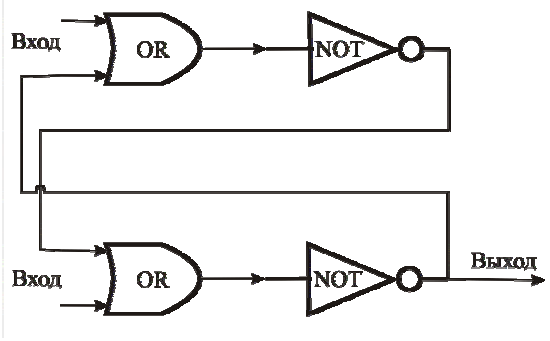


Рис. 1.4. Другой способ построения триггера

Заряды сохранятся на пластинах конденсатора и после отключения источника напряжения. Если впоследствии пластины соединить проводником, то по нему потечет электрический ток и заряды нейтрализуются. Таким образом, конденсатор может пребывать в заряженном или разряженном состоянии; одно из них вполне может представлять значение 0, а другое – значение 1. С помощью современных технологий на тонких пластинах, называемых чипами, можно размещать миллионы крошечных конденсаторов вместе с необходимыми схемами электрических соединений. Благодаря этому в настоящее время конденсаторы широко используются для хранения битов в вычислительных машинах.

Триггеры, конденсаторы – это примеры запоминающих систем с различной степенью продолжительности хранения информации. Заряды в крошечных конденсаторах настолько недолговечны, что способны быстро исчезать сами по себе, даже если машина находится в рабочем состоянии. Поэтому заряд конденсатора необходимо регулярно возобновлять с помощью специальной схемы, называемой цепью регенерации. Принимая во внимание кратковременность хранения данных, созданную по такой технологии компьютерную память именуют *динамической памятью* (dynamic memory).

Структура основной памяти. Запоминающие схемы основной памяти машины организованы в небольшие блоки (доступные как единое целое), которые называются ячейками памяти (cell). Как правило, размер ячейки памяти составляет восемь бит. Наборы из восьми бит получили такую популярность, что для их обозначения сейчас широко используется специальный термин байт (byte).

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 машине 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*. Бит на другом конце именуют *младшим*, или *битом с наименьшим весом*. Таким образом, содержимое ячейки памяти размером один байт можно представить себе так, как показано на рис. 1.5.

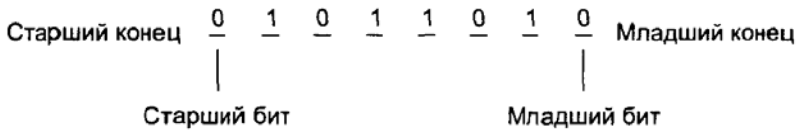


Рис. 1.5. Организация ячейки памяти в 1 байт

Микрокомпьютеры, используемые, например в микроволновых печах, имеют основную память, которая измеряется всего лишь несколькими сотнями ячеек, тогда как компьютеры, предназначенные для хранения и обработки большого количества информации, имеют миллиарды ячеек основной памяти. Размер основной памяти машины часто измеряется единицами в 1 048 576 отдельных ячеек. (Величина 1 048 576 – это число, равное 2^{20}, и это значение более удобно в качестве единицы измерения в компьютере, чем число 1 000 000). Для обозначения этой единицы измерения используется термин *мега*. Аббревиатура *Мбайт* обычно употребляется как сокращение для термина *мегабайт*. Следовательно, память емкостью 4 Мбайт содержит 4 194 304 (4 × 1 048 576) ячейки, каждая размером 1 байт. Другими единицами измерения памяти являются *килобайт* (сокращенно кбайт), который равен 1024 байт (2^{10} байт), и *гигабайт* (сокращенно Гбайт), который равен 1024 Мбайт, или 2^{30} байт.

Для идентификации отдельных ячеек основной памяти машины каждой ячейке присваивается уникальное имя, называемое *адресом ячейки* (address). 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. Адреса ячеек в машине с памятью 4 Мбайт будут представлены числами 0, 1, 2, ..., 4194304. Следует отметить, что такая система адресации не только позволяет однозначно идентифицировать каждую ячейку памяти (рис. 1.6), но и упорядочивает их, делая правомочными такие выражения, как "следующая ячейка" или "предыдущая ячейка".

В состав основной памяти машины, помимо электрической цепи, фиксирующей значения битов, входит и другая цепь, 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.

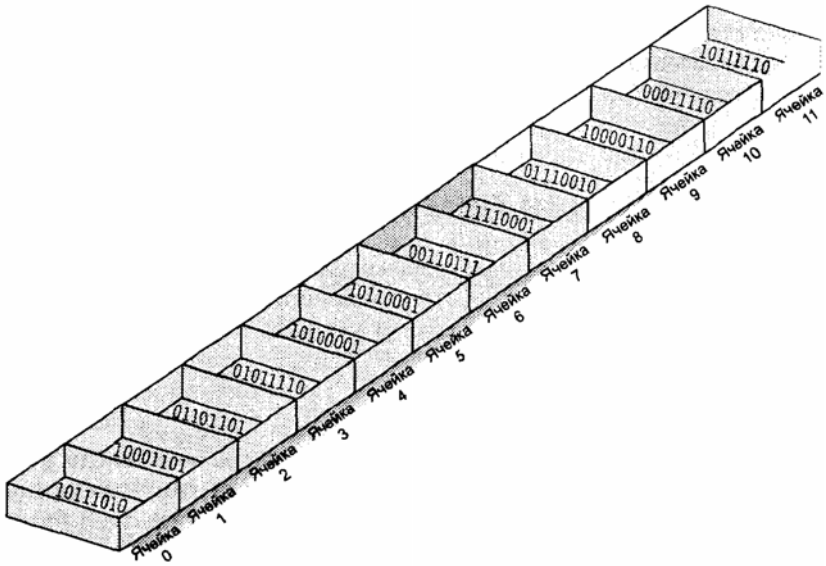


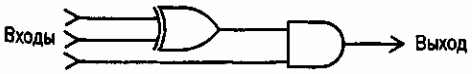
Рис. 1.6. Образное представление ячеек памяти, упорядоченных по адресам

В этих устройствах длинные строки битов приходится обрабатывать как единый блок. Если память типа RAM создается с использованием технологии динамической памяти, то в этом случае ее называют *динамической памятью с произвольной выборкой* (Dynamic RAM, DRAM).

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 один байт, для сохранения строки из 16 бит можно просто воспользоваться двумя последовательными ячейками памяти.

Вопросы для самопроверки

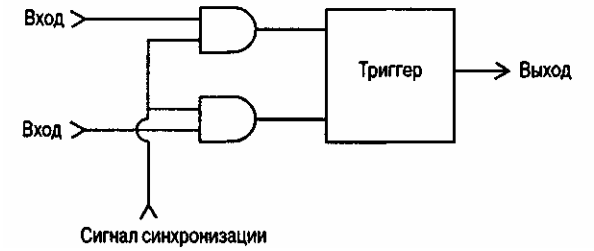
1. При каких значениях на входах представленной ниже схемы на ее выходе появится значение 1?



2. Выше утверждалось, что при поступлении значения 1 на нижний вход триггера, показанного на рис. 1.2 (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. 1.4, подается значение 0. Опишите последовательность событий, которые будут происходить в элементах этого триггера при кратковременном поступлении на его верхний вход значения 1.

4. Довольно часто необходимо согласовывать действия различных частей схемы. Это достигается путем подачи импульсного сигнала (называемого *сигналом синхронизации*) в те части схемы, работу которых требуется согласовать. Изменение значения сигнала синхронизации с нуля на единицу вызывает активизацию различных компонентов схемы. Ниже приведен пример одной из частей подобной схемы, включающей триггер, изображенный на рис. 1.2. При каких значениях сигнала синхронизации триггер будет защищен от воздействия значений, поступающих на входы этой схемы? При каких значениях сигнала синхронизации триггер будет реагировать на значения, поступающие на входы этой схемы?



5. 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?

6. Предположим, что требуется поменять местами значения, хранящиеся в ячейках памяти с номерами 2 и 3. Найдите ошибку в следующей последовательности действий.

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

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

Предложите последовательность действий, которая позволит корректно поменять местами содержимое указанных ячеек.

7. Какое количество битов содержится в памяти компьютера, размер которой равен 4 кбайт?

1.2. ЗАПОМИНАЮЩИЕ УСТРОЙСТВА БОЛЬШОЙ ЕМКОСТИ

В связи с невозможностью постоянного хранения данных и ограниченным объемом основной памяти компьютера большинство машин обеспечивается устройствами дополнительной памяти, которые называются *массовой памятью*, или *запоминающими устройствами большой емкости* (mass storage system). Преимущества таких устройств, по сравнению с основной памятью компьютера, состоят в долговременности хранения данных, большей емкости и, в большинстве случаев, автономности, т.е. возможности извлечения носителя информации из машины, например в целях архивирования.

Основным недостатком устройств массовой памяти является то, что они обычно требуют механических перемещений носителя или устройства считывания. Поэтому время доступа к информации у этих устройств существенно больше по сравнению с основной памятью машины, которая является электронной.

Магнитные диски. Одним из наиболее распространенных типов запоминающих устройств большой емкости, применяемых в наше время, являются устройства, которые используют в качестве носителя информации *магнитные диски* (magnetic disk). Устройства считывания – *головки чтения/записи* (headers) – размещаются над и/или под диском таким образом, что во время вращения диска каждая головка описывает над ним круг, называемый *дорожкой* (track), расположенной на верхней и/или нижней поверхности диска. 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* (cylinder).

Так как дорожка может содержать больше информации, чем обычно требуется одновременно обрабатывать, все дорожки поделены на зоны, или *секторы* (sectors), в которых информация записывается в виде непрерывной последовательности битов (рис. 1.7). Каждая дорожка внутри дисковой системы содержит одинаковое количество секторов, а каждый сектор, в свою очередь, – одинаковое число двоичных разрядов. (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.)

Таким образом, мы выяснили, что дисковое запоминающее устройство состоит из множества отдельных секторов, каждый из которых

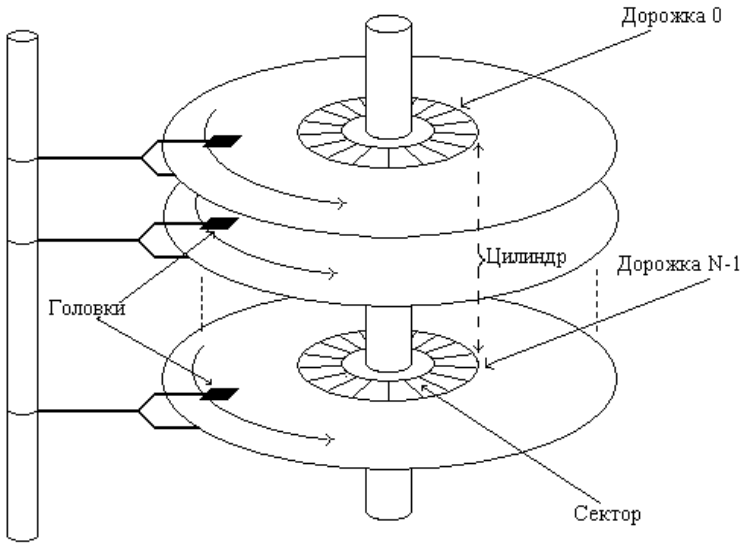


Рис. 1.7. Дисковое запоминающее устройство

может быть независимо считан как одна строка битов. Количество дорожек на поверхности диска, а также количество секторов на дорожках могут значительно отличаться в разных дисковых устройствах. Размеры секторов обычно не превышают нескольких килобайт. Чаще всего размер сектора составляет 512 или 1024 байта.

Расположение дорожек и секторов не является постоянной характеристикой, зафиксированной в физической структуре диска. На самом деле они маркируются магнитным способом с помощью процесса, который называется *форматированием* (formatting) (или инициализацией) диска. Этот процесс обычно осуществляется той фирмой, которая производит дисковые устройства, и на рынок поступают уже отформатированные диски. Большинство компьютерных систем тоже могут форматировать диски. Поэтому в случае повреждения формата диска он может быть переформатирован, однако это приведет к уничтожению всей информации, которая прежде была записана на данном устройстве.

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 единственного пластикового диска, который называется *дискетой*, или *гибким диском* (floppy disk). (Современные гибкие диски размером $3^1/_2$ дюйма имеют жесткие пластиковые корпуса, а не гибкие упаковки, в отличие от своих более старых аналогов диаметром $5^1/_4$ дюйма, которые упаковывались в бумажные конверты.) Дискеты легко вставляются и вынимаются из устройств, а также достаточно удобны в хранении. Поэтому они часто используются как автономные хранилища информации. Универсальная дискета размером $3^1/_2$ дюйма имеет емкость, достаточную для хранения 1,44 Мбайт информации. Однако существуют и дискеты с существенно большей емкостью. Примером может служить дисковое устройство типа Zip компании Iomega Corporation, где на одной жесткой дискете может записываться несколько сотен мегабайт информации.

Дисковые системы большой емкости способны хранить многие гигабайты информации. Такие устройства включают от пяти до десяти жестких дисковых пластин, смонтированных на общей оси. Поскольку используемые в таких устройствах диски являются жесткими, их называют системами с *жестким диском* (hard disk), в отличие от гибких дисков, обсуждавшихся выше. Чтобы увеличить скорость вращения дисков, головки чтения/записи в таких системах размещены так, что они не соприкасаются с поверхностью диска, а как бы "плавают" над поверхностью с магнитным покрытием. Расстояние между головкой и диском настолько мало, что даже отдельная частица пыли может застрять между ними и вызвать их повреждение (явление, известное как *разрушение головки*). Поэтому устройства жестких дисков герметически упаковывают в коробки и запечатывают непосредственно на том предприятии, где они изготовляются.

Для оценки производительности дисковой системы используется несколько параметров:

1) *время установки* (seek time) – время, которое требуется для перемещения головки чтения/записи с одной дорожки на другую;

2) *задержка вращения* (rotation delay), или *время ожидания* (latency time) – 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* (access time) – сумма времени установки и времени ожидания;

4) *скорость передачи* (transfer rate) – скорость, с которой данные могут передаваться дисковому устройству или считываться с него.

Устройства с жесткими дисками имеют намного лучшие характеристики в сравнении с устройствами, использующими гибкие диски. Так как головки чтения/записи не соприкасаются с поверхностью жесткого диска, скорость вращения достигает от 3000 до 7000 оборотов в минуту, тогда как скорость вращения гибких дисков составляет только 300 оборотов в минуту. Поэтому устройства с жесткими дисками имеют более высокую скорость передачи, измеряемую обычно в мегабайтах в секунду, тогда как скорость передачи данных гибких дисков измеряется в килобайтах в секунду.

Поскольку работа дисковых устройств требует физического перемещения носителя, жесткие и гибкие диски проигрывают в скорости по сравнению с электронными схемами. Это неудивительно, так как задержки в электронных схемах измеряются в наносекундах (миллиардная доля секунды) и меньше, тогда как время установки, ожидания и доступа дисковых устройств измеряется в миллисекундах (тысячная доля секунды). Таким образом, время, требуемое для считывания информации с дисковых устройств, кажется просто вечностью в сравнении со скоростью работы электронных схем.

Компакт-диски. Еще одной популярной технологией хранения данных является использование *компакт-дисков* (compact disk – 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 (compact disk digital audio – компакт-диск с цифровой звукозаписью). Компакт-диски, используемые в настоящее время для хранения компьютерных данных, похожи на своих аудио предшественников, за исключением того, что для них применяется формат CD-ROM (compact disk – read only memory или компакт-диск – постоянное запоминающее устройство). Различие между форматами CD-DA и CD-ROM состоит в способе интерпретации полей данных. Например, в формате CD-DA определенные поля предназначены для хранения информации о времени воспроизведения, тогда как в формате CD-ROM это пространство используется для хранения произвольных данных.

В отличие от устройств с магнитными дисками, где запись данных осуществляется на концентрических дорожках, информация на компакт-дисках записывается на единственной дорожке, которая закручивается спиралью на поверхности диска подобно желобку на старых грампластинках. (Но в отличие от старых грампластинок дорожка на компакт-диске записывается в направлении от центра к краю.) Эта дорожка разделена на части, которые называют секторами (рис. 1.8). Секторы содержат одинаковое количество данных, и у каждого есть своя личная маркировка. Сектор в формате CD-ROM содержит 2 кбайт информации, а сектор того же размера в формате CD-DA содержит данные, обеспечивающие воспроизведение музыки в течение 1/75 секунды.

Рис. 1.8. Особенности хранения данных на компакт-дисках

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. Это означает, что на витке во внешней части спирали хранится большее количество информации, чем на витке в ее внутренней части. Поэтому за один оборот диска будет считываться больше секторов, когда лазерный луч сканирует внешнюю часть спиральной рожки, и меньше секторов, когда луч будет сканировать внутреннюю часть рожки. 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.

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 (например, как в системе резервирования авиабилетов), подход, используемый в устройствах магнитных дисков (отдельные концентрические дорожки, каждая из которых содержит одинаковое количество секторов), оказывается эффективнее спирального метода записи, используемого в компакт-дисках.

Емкость компакт-диска в формате CD-ROM составляет немного более 600 Мбайт. Однако уже появились новые дисковые форматы, например DVD (Digital Versatile Disk – цифровой универсальный диск). В этом формате емкость каждого носителя составляет порядка 10 Гбайт. На таких компакт-дисках можно хранить мультимедиа-презентации, в которых аудио- и видеоинформация комбинируется в целях более интересной и содержательной дачи материала. Главная задача разработки стандарта DVD состоит в представлении инструментальных средств для записи на компакт-диски полнометражных кинофильмов.

Еще одним вариантом в технологии компакт-дисков является формат CD-WORM (Compact Disk – Write Once, Read Many или компакт-диск с однократной записью и многократным считыванием). Он позволяет записывать данные на компакт-диск после его изготовления, а не во время этого процесса. Эти устройства чрезвычайно удобны для архивирования, а также для производства записей на компакт-дисках в небольших количествах.

Магнитная лента. В более ранних типах запоминающих устройств большой емкости используется *магнитная лента* (magnetic types) (рис. 1.9). В этом случае информация записывается на магнитное покрытие тонкой пластиковой ленты, которая для хранения наматывается на бобину. Чтобы получить доступ к записанным на ней данным, магнитная лента устанавливается на устройство, называемое лентопротяжным механизмом. Это устройство позволяет считывать, записывать и перематывать магнитную ленту под управлением компьютера. По своим размерам лентопротяжные механизмы могут варьироваться от небольших кассетных блоков, называемых стриммерами (в них применяются кассеты, подобные видеокассетам), до более старых и громоздких катушечных устройств.

В современных стриммерных устройствах лента разделена на сегменты, которые маркируются магнитным способом в процессе форматирования (данный способ подобен методу, применяемому для дисковых носителей информации). Каждый

из сегментов содержит несколько дорожек, расположенных вдоль ленты параллельно друг другу. К каждой такой дорожке доступ можно получить независимо от других. Это означает, что лента в сущности состоит из совокупности отдельных строк битов, напоминающих секторы на диске.

Основным недостатком стриммерных устройств является то, что для доступа к информации может потребоваться достаточно много

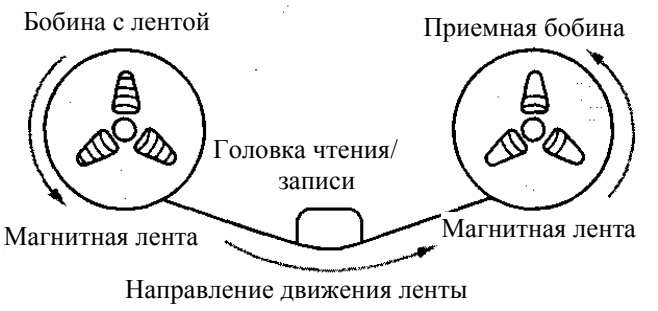


Рис. 1.9. Запоминающее устройство на магнитной ленте

времени, поскольку это связано с перемоткой ленты с одной бобины на другую. Поэтому лентопротяжные устройства характеризуются существенно большим временем доступа к информации, чем устройства с магнитными дисками, в которых для доступа к различным секторам достаточно короткого перемещения головки чтения/записи. Именно по этой причине лентопротяжные устройства не приобрели широкой популярности в качестве основных носителей информации. Однако если речь идет об архивировании данных, то большая емкость, надежность и невысокая стоимость ленточных устройств позволяют считать их хорошим выбором среди прочих современных устройств хранения данных.

Сохранение и считывание файлов. Информация в массовой памяти хранится в виде больших именованных блоков, которые принято называть *файлами* (files). Типичный файл может содержать некоторый текстовый документ, фотографию, программу или совокупность данных о персонале какой-либо компании. Физические особенности устройств массовой памяти требуют, чтобы файлы сохранялись и считывались отдельными блоками из большого количества байтов. Например, каждый сектор на магнитном диске должен обрабатываться как одна непрерывная строка битов. Блок данных, соответствующий физическим характеристикам запоминающего устройства, называется *физической записью* (physical record). Поэтому файл, записанный в массовую память, обычно состоит из множества физических записей.

Помимо разделения на физические записи, любой файл обычно подразумевает некоторое естественное разграничение представленной в нем информации. Например, файл с информацией о персонале компании будет состоять из множества элементов, каждый из которых содержит сведения об отдельном человеке. Такие блоки данных, естественным образом образующиеся при создании файла, называют *логическими записями* (logical records).

Размер логической записи редко совпадает с размером физической записи, который определяется типом устройства массовой памяти. Поэтому несколько логических записей могут помещаться в одну физическую запись, и наоборот, логическая запись при необходимости может разделяться на несколько физических (рис. 1.10). В результате считывание данных файла с устройства массовой памяти обычно связано с восстановлением его логических записей из физических. Типичный способ решения этой проблемы состоит в выделении некоторой области основной памяти, достаточно большой для размещения нескольких физических записей файла, и использовании ее в качестве промежуточного хранилища для перегруппировки информации. В результате обмен данными между этой промежуточной областью и устройством массовой памяти может осуществляться блоками данных, соответствующими физическим записям, тогда как для программ находящаяся в

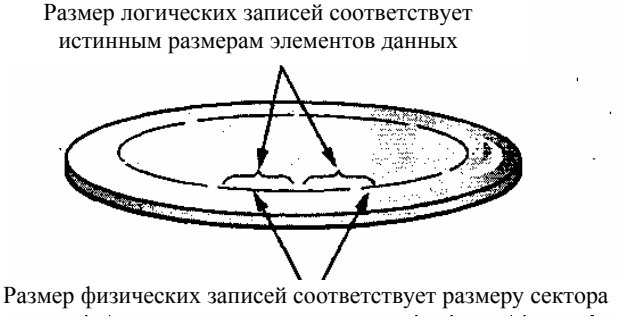


Рис. 1.10. Представление логических и физических записей на диске

основной памяти информация может быть представлена в виде логических записей. Используемая подобным образом область основной памяти называется *буфером* (buffer).

Использование буфера поясняет относительную роль основной и массовой памяти в системе. Основная память используется для хранения данных в целях их обработки, тогда как массовая память является постоянным хранилищем информации. Таким образом, обновление сохраняемой в массовой памяти информации предполагает передачу информации в основную память, изменение ее, а затем возврат обновленной информации в массовую память.

Можно сделать заключение, что основная память, магнитные диски, компакт-диски и магнитная лента представляют различные уровни возможности прямого доступа к данным в порядке ее уменьшения. Используемая в основной памяти система адресации допускает быстрый произвольный доступ к отдельным байтам данных. Магнитные диски обеспечивают прямой доступ только к целым секторам данных. Кроме того, время, затрачиваемое на считывание сектора, включает также

время установки и время ожидания. Компакт-диски тоже поддерживают произвольный доступ к отдельным секторам, однако величина задержки для компакт-дисков значительно больше по сравнению с магнитными дисками. Это обусловлено тем, что в этом случае требуется дополнительное время для того, чтобы найти спиральную дорожку и настроить скорость вращения диска. Наконец, устройства с магнитной лентой не позволяют получать прямой доступ к информации. Современные лентопротяжные системы маркируют фрагменты ленты, что позволяет ссылаться на различные сегменты по отдельности, однако сама физическая организация данных обуславливает то, что поиск сегмента потребует достаточно много времени.

Вопросы для самопроверки

1. Какие преимущества дает устройствам с жестким диском большая скорость вращения диска по сравнению с гибким 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 ленточном устройстве?
4. Предположим, что логические записи длиной 450 байт должны храниться на диске, размер секторов которого составляет 512 байт. Приведите аргумент в защиту решения о размещении только одной логической записи в каждой физической записи, даже несмотря на то, что 62 байта в каждом секторе останутся свободными.

1.3. ПРЕДСТАВЛЕНИЕ ЦЕЛЫХ ЧИСЕЛ

В разделе рассматриваются две системы двоичного представления (binary notation) целых чисел, которые наиболее часто используются в компьютерном оборудовании. В основе этих представлений лежит двоичная система счисления, о которой говорится в приложении А, но они обладают некоторыми особенностями, которые делают их более подходящими для вычислительной машины. Одна из этих особенностей дает возможность кодирования и положительных, и отрицательных чисел. Другой особенностью является использование фиксированного числа битов для представления числового значения. Эта особенность может приводить при вычислениях к ошибкам особого рода, обсуждаемым в данном разделе.

Для хранения дробей используется своя система представления, которая дополнительно позволяет закодировать положение точки в дробном числе. Особенности представления дробных чисел описываются в следующем разделе.

Двоичный дополнительный код. Наиболее распространенной системой представления целых чисел в современных компьютерах является представление в *двоичном дополнительном коде* (two's complement notation). Эта система использует фиксированное число битов для представления числового значения. В современном оборудовании принято использовать представление, при котором каждому значению отводится 32 бита. Такой подход позволяет хранить большой диапазон чисел, однако его очень трудно изобразить наглядно. Поэтому мы сосредоточим наше внимание на коротких системах представления.

Запись чисел в десятичной системе счисления	Запись чисел в двоичной системе счисления	Запись чисел в трехразрядном двоичном дополнительном коде
3	11	011
2	10	010
1	1	001
0	0	000
-1	-1	111
-2	-10	110
-3	-11	101
-4	-100	100

а)

Запись чисел в десятичной системе счисления	Запись чисел в двоичной системе счисления	Запись чисел в четырехразрядном двоичном дополнительном коде
7	111	0111
6	110	0110
5	101	0101
4	100	0100
3	11	0011
2	10	0010
1	1	0001
0	0	0000
-1	-1	1111
-2	-10	1110
-3	-11	1101
-4	-100	1100
-5	-101	1011
-6	-110	1010
-7	-111	1001

б)

Рис. 1.11. Схемы кодирования в двоичном дополнительном коде

Два представления в двоичном дополнительном коде изображены на рис. 1.11. В этих вариантах 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, ...

Для преобразования битовых комбинаций, представляющих положительные и отрицательные числа, имеющие одно и то же значение по модулю, достаточно копировать исходную комбинацию справа налево до тех пор, пока не будет встречена единица, а затем последовательно заменять значения оставшихся битов их дополнениями (рис. 1.12). *Дополнением двоичной ком-*

бинации (complement) называется такая комбинация, которая получается в результате изменения всех нулей в исходном значении на единицы. а всех единиц на нули. Например, двоичные комбинации 0110 и 1001 являются дополнениями друг другу.

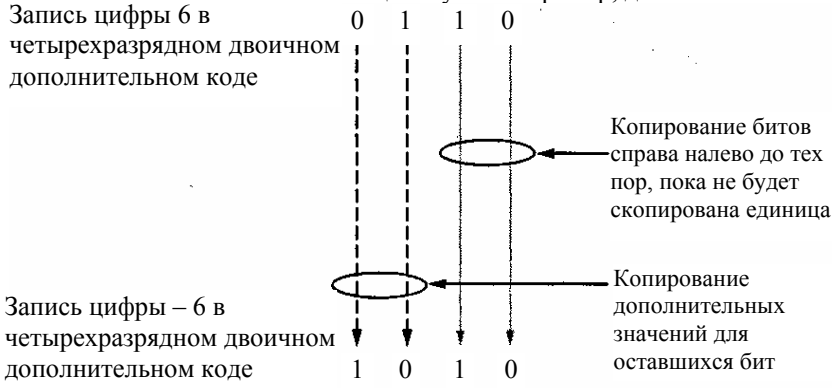


Рис. 1.12. Представление числа –6 в четырехразрядном дополнительном коде

Рассмотрим пример преобразования значений этого кода в десятичное представление.

Пример. Определить десятичное значение комбинации 1010.

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 двойная комбинация представляет число –6.

Сложение в двоичном дополнительном коде. Для сложения чисел в двоичном дополнительном коде применяется такой же алгоритм, как для двоичного сложения, только в этом случае все коды, включая результат операции, будут иметь одинаковую длину. Это означает, что, если в результате сложения появляется дополнительный бит с левого края, он будет отсечен. Именно поэтому 0101 и 0010 в сумме дают 0111, а сумма 0111 и 1011 равна 0010 (0111 + 1011 = 10010, которая усекается до 0010).

На рис. 1.13 представлены примеры сложения чисел в двоичном дополнительном коде. Обратите внимание, что за счет преобразования исходных данных в двоичные дополнительные коды можно вычислить результат, как для сложения и вычитания, 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.

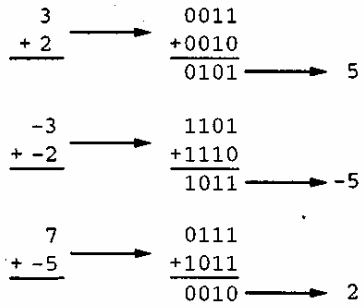


Рис. 1.13. Сложение чисел в двоичном дополнительном коде

Поэтому при использовании двоичного дополнительного кода необходимо реализовать электронные схемы только для осуществления операций сложения и отрицания (операция вычитания 7 – 5 аналогична операции сложения 7 + (–5)). Этого будет достаточно для выполнения как операций сложения, так и вычитания. Электронные схемы сложения и отрицания представлены в приложении Б.

Ошибка переполнения. Одной из проблем, которые существуют в любом представлении в двоичном дополнительном коде, является ограничение на размер чисел, представимых данным количеством битов. Например, если мы используем 4-битовый двоичный дополнительный код, то у числа 9 не будет соответствующей записи. Это означает, что мы не получим правильного ответа, выполнив операцию сложения 5 + 4. На самом деле мы получим ответ – 7. Такая ошибка называется *переполнением* (overflow), она возникает тогда, когда нужно сохранить число, не попадающее в диапазон чисел, которые могут быть представлены в двоичном дополнительном коде.

Переполнение может возникнуть, когда нужно сложить два положительных или два отрицательных числа. В обоих случаях можно проверить, есть ли ошибка, посмотрев на знаковый разряд полученного результата, т.е. переполнение возникло, если сумма двух положительных чисел имеет код отрицательного числа или сумма двух отрицательных чисел имеет код положительного числа.

В настоящее время для хранения чисел в двоичном дополнительном коде обычно применяются битовые комбинации длиной 32 бита, что позволяет без возникновения переполнения обрабатывать числа от –2 147 483 648 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, километры, а не метры).

Двоичная нотация с избытком*. Другой способ представления целочисленных значений называется *двоичной нотацией с избытком* (excess notation). В отличие от двоичного дополнительного кода в этой нотации отрицательные числа представляются комбинациями со знаковым битом, равным 0, а положительные числа – комбинациями со знаковым битом, равным 1 (см. рис. 1.14).

Таблица кодов, изображенная на рис. 1.14, *б*, называется двоичной нотацией с избытком восемь. Для того чтобы понять, почему это так, сначала переведем коды из таблицы в десятичную систему счисления,как обычный двоичный код, и сравним полученные значения со значениями в таблице.

Запись чисел в десятичной системе счисления	Запись чисел в двоичной системе счисления	Запись чисел в двоичной нотации с избытком четыре
3	11	111
2	10	110
1	1	101
0	0	100
-1	-1	011
-2	-10	010
-3	-11	001
-4	-100	000

а)

Запись чисел в десятичной системе счисления	Запись чисел в двоичной системе счисления	Запись чисел в двоичной нотации с избытком восемь
7	111	1111
6	110	1110
5	101	1101
4	100	1100
3	11	1011
2	10	1010
1	1	1001
0	0	1000
-1	-1	0111
-2	-10	0110
-3	-11	0101
-4	-100	0100
-5	-101	0011
-6	-110	0010
-7	-111	0001
-8	-1000	0000

б)

Рис. 1.14. Схемы кодирования в двоичной нотации с избытком

В каждом случае вы обнаружите, что полученный результат превосходит код в представлении с избытком на восемь. Например, последовательность 1100 в двоичной системе является записью числа 12, но в представлении с избытком она является кодом 4; последовательность 0000 в двоичной системе является записью числа 0, а в представлении с избытком – кодом 8. Точно так же 5-битовое представление с избытком будет называться представлением с избытком 16, так как, например, последовательность 10000 будет кодом 0, а не 16, как в двоичной записи. Вы можете убедиться, что 3-битовое представление с избытком является представлением с избытком четыре (рис. 1.14, *а*).

Вопросы для самопроверки

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) 26; в) 217; г) 13; д) 21; 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. Какое наибольшее и наименьшее число может быть записано, 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) $+\underline{\begin{matrix}0101\\0010\end{matrix}}$; б) $+\underline{\begin{matrix}0011\\0001\end{matrix}}$; в) $+\underline{\begin{matrix}0101\\1010\end{matrix}}$; г) $+\underline{\begin{matrix}1110\\0011\end{matrix}}$; д) $+\underline{\begin{matrix}1010\\1110\end{matrix}}$.

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) $+\underline{\begin{matrix}0100\\0011\end{matrix}}$; б) $+\underline{\begin{matrix}0101\\0110\end{matrix}}$; в) $+\underline{\begin{matrix}1010\\1010\end{matrix}}$; г) $+\underline{\begin{matrix}1010\\0111\end{matrix}}$; д) $+\underline{\begin{matrix}0111\\0001\end{matrix}}$.

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) $+\underline{\begin{matrix}6\\1\end{matrix}}$; б) $-\underline{\begin{matrix}3\\2\end{matrix}}$; в) $-\underline{\begin{matrix}4\\6\end{matrix}}$; г) $+\underline{\begin{matrix}2\\4\end{matrix}}$; д) $-\underline{\begin{matrix}1\\5\end{matrix}}$.

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; е) 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; е) –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.

1.4. ПРЕДСТАВЛЕНИЕ ДРОБНЫХ ЧИСЕЛ*

В отличие от хранения целых чисел, для чисел с дробной частью требуется хранить не только двоичное представление числа и его знак, но и позицию разделительной точки. Общепринятым способом хранения дробей является представление с плавающей точкой.

Двоичная нотация с плавающей точкой. Для представления дробных значений используют способ, который называется *двоичной нотацией с плавающей точкой* (floating-point notation). На рис. 1.15 представлены компоненты дробного числа, состоящего из 8 бит и записанного в двоичной нотации с плавающей точкой (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).

Пример. Представить битовую комбинацию 01101011, записанную в двоичной нотации с плавающей точкой в десятичном формате.

Знаковый бит этого числа равен 0, поле порядка числа имеет значение 110, а поле мантиссы — значение 1011. Вначале выделим мантиссу и поместим плавающую точку слева от нее, как показано ниже:

.1011

Далее выделим значение в поле порядка числа (110) и интерпретируем его как целое трехразрядное число, записанное в двоичной нотации с избытком. Таким образом, в поле порядка числа закодировано целое число 2. Это означает, что плавающую точку в полученном ранее значении следует переместить на два бита вправо (при отрицательном порядке плавающая точка перемещается влево), после чего будет получен окончательный результат:

10.11

Это значение является двоичным представлением числа $2^3/_4$. 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 $2^3/_4$.

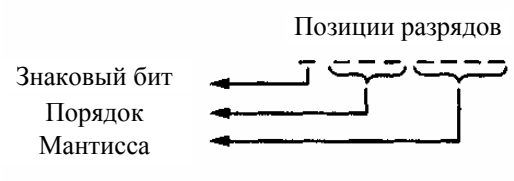


Рис. 1.15. Компоненты числа в двоичной нотации с плавающей точкой

Пример. Представить битовую комбинацию 10111100, записанную в двоичной нотации с плавающей точкой в десятичном формате.

Выделив мантиссу, получим следующее значение:

.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 отрицательным, поскольку его знаковый бит равен 1. Из этого следует, что битовая комбинация 10111100 в двоичной нотации с плавающей точкой представляет число $-\,^3/_8$.

Для представления чисел в двоичной нотации с плавающей точкой необходимо следовать описанному выше процессу, но уже в обратном порядке.

Пример. Представить в двоичной нотации с плавающей точкой число $1^1/_8$.

Сначала число $1^1/_8$ необходимо записать в его двоичном представлении: 1.001. Затем эта битовая комбинация копируется в поле мантиссы слева направо, начиная с самой левой единицы в двоичном представлении числа:

_ _ _ _ 1 0 0 1

Определим число разрядов, а также направление, в котором будет перемещаться плавающая точка для получения исходного значения двоичного числа. Здесь можно увидеть, что точка в комбинации .1001 должна быть перемещена на один бит вправо; в результате будет получено исходное значение 1.001. Таким образом, порядок числа равен 1 или 101 в двоичной нотации с избытком четыре. Окончательное значение в байте будет выглядеть следующим образом:

0 1 0 1 1 0 0 1

При заполнении поля мантиссы имеется один тонкий момент: правило требует копировать битовую комбинацию двоичного представления числа в поле мантиссы слева направо, начиная с *крайней левой единицы*. Данное правило исключает возможность различного представления одного и того же значения, и, кроме того (что является, пожалуй, самым важным), если "плавающая" точка расположена в мантиссе перед первой значащей цифрой (т.е. если используется нормализованная запись числа – см. Приложение А), то при фиксированном количестве разрядов, отведенных под мантиссу, обеспечивается запись максимального количества значащих цифр числа, т.е. максимальная точность представления числа в машине.

Пример. Представить в двоичной нотации с плавающей точкой число $^3/_8$. Двоичным представлением числа $^3/_8$ является битовая комбинация .011. В этом случае мантисса должна иметь следующее значение:

_ _ _ _ 1 1 0 0

Любой другой вариант, например представленный ниже, недопустим:

_ _ _ _ 0 1 1 0

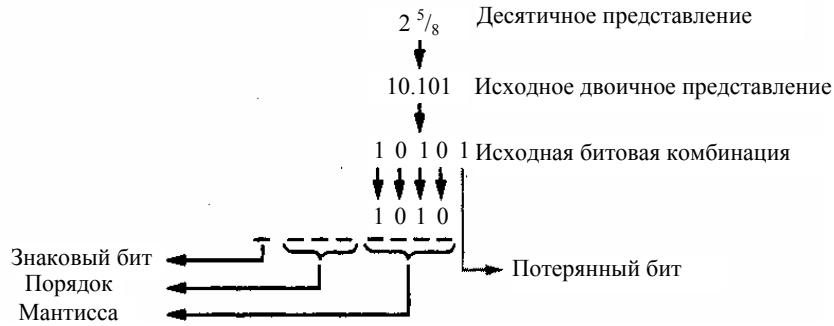
Ошибка усечения. При операциях с числами, записанными в двоичной нотации с плавающей точкой, могут возникать ошибки, аналогичные ошибкам переполнения.

Пример. 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 самая правая единица в двоичном представлении, имеющая весовое значение $^1/_8$, теряется (рис. 1.16). Если не обратить на это внимание и продолжить заполнение поля порядка числа и знакового бита, будет получена комбинация 01101010, которая на самом деле представляет число $2^1/_2$, а не $2^5/_8$.

Это явление называется *ошибкой усечения* (truncation error), или *ошибкой округления* (round-off error). Оно означает, что некоторая часть кодируемого числа теряется, поскольку размер поля мантиссы оказывается недостаточным.

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.



1.16. Схема кодирования числа $2^5/_8$

Существует еще одна причина появления ошибок усечения, с которой каждый из нас уже встречался при изучении десятичной системы счисления. Это пробле*ма 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 мы не использовали для их представления.

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 мы не использовали 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 могут возникнуть и при сложении очень больших и маленьких чисел. 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.

Пример. Сложить следующие три числа, представленные в однобайтовых кодах двоичной нотации с плавающей точкой $2^1/_2 + {}^1/_8 + {}^1/_8$.

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. Это число не может быть представлено точно (см. предыдущий пример), 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 ответ правильный.

Поэтому общее правило суммирования большого количества чисел требует начинать операцию сложения с самых малых чисел, предполагая, что в результате будет получено достаточно большое промежуточное значение, которое затем можно безопасно сложить с оставшимися большими числами.

Вопросы для самопроверки

1. Декодируйте приведенные ниже битовые комбинации с помощью формата с плавающей точкой, описанного 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) $-4^3/_4$.

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 большим.

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?

1.5. ПРЕДСТАВЛЕНИЕ ТЕКСТА, ИЗОБРАЖЕНИЙ И ЗВУКА

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

В ранний период развития компьютерной технологии было разработано много подобных кодов, причем каждый из них использовался в различных элементах оборудования. Это 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-кии"), который приобрел очень большую популярность. В этом коде комбинации двоичных разрядов длиной семь бит используются для представления строчных и прописных букв английского алфавита, знаков пунктуации, цифр от 0 до 9, а также кодов управления передачей информации (перевод строки, возврат каретки и 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 код, размер которого 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. 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 с одной программы в другую, особенно если эти программы были разработаны разными фирмами.

Ниже приведен неполный список ASCII-кодов символов. В этом списке к исходным семиразрядным двоичным кодам слева приписаны нули – для получения восьмибитовых кодов, общепринятых в настоящее время.

Символ	ASCII-код	Символ	ASCII-код	Символ	ASCII-код
(пробел)	00100000	?	00111111	~	01011110
!	00100001	@	01000000	_	01011111
“	00100010	A	01000001	a	01100001
#	00100011	B	01000010	b	01100010
$	00100100	C	01000011	c	01100011
%	00100101	D	01000100	d	01100100
&	00100110	E	01000101	e	01100101
‘	00100111	F	01000110	f	01100110
(	00101000	G	01000111	g	01100111
)	00101001	H	01001000	h	01101000
*	00101010	I	01001001	i	01101001
+	00101011	J	01001010	j	01101010
,	00101100	K	01001011	k	01101011
–	00101101	L	01001100	l	01101100
.	00101110	M	01001101	m	01101101
/	00101111	N	01001110	n	01101110
0	00110000	O	01001111	o	01101111
1	00110001	P	01010000	p	01110000
2	00110010	Q	01010001	q	01110001
3	00110011	R	01010010	r	01110010
4	00110100	S	01010011	s	01110011
5	00110101	T	01010100	t	01110100
6	00110110	U	01010101	u	01110101
7	00110111	V	01010110	v	01110110
8	00111000	W	01010111	w	01110111
9	00111001	X	01011000	x	01111000
:	00111010	Y	01011001	y	01111001
;	00111011	Z	01011010	z	01111010
<	00111100	[	01011011	{	01111011
=	00111101	\	01011100	}	01111101
	>	00111110	]	01011101	

На рис. 1.17 показано, как в этой кодировке приветствие *Hello* представляется с помощью следующей комбинации битов: 01001000 01100101 01101100 01101100 01101111 00101110.

Несмотря на то что ASCII – это один из наиболее широко используемых кодов, сегодня растет популярность кодов с более широкими возможностями, которые способны представлять документы на разных языках. Одним из них является Unicode, который был разработан в результате объединенных усилий нескольких ведущих фирм-производителей программного и аппаратного обеспечения. В этом коде для представления каждого символа используется уникальная комбинация из 16 двоичных разрядов. В результате кодировка Unicode включает 65 536 различных двоичных кодов, что вполне достаточно даже для представления всех широко употребляемых китайских и японских алфавитов. *Международная организация по стандартизации* (International Organization for Standardization, часто именуемая ISO, от греческого isos – одинаковый) разработала код, способный соперничать даже с кодировкой Unicode. Здесь для выражения символов используются комбинации из 32 бит, в результате чего этот код позволяет представить более 17 миллионов символов. Будущее покажет, какой из двух кодов приобретет большую популярность.

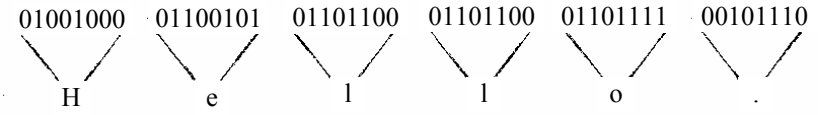


Рис. 1.17. Представление слова Hello, в кодах ASCII

Несмотря на то что метод хранения информации в виде закодированных символов достаточно удобен, он оказывается неэффективным при записи чисто числовой информации. Например, пусть в память требуется записать число 25. Если воспользоваться символами в кодах ASCII, то для записи этого числа потребуется один байт на каждый символ, а всего – 16 бит (более того, самое большое число, которое мы сможем представить с помощью 16 битов, – это 99). Используя двоичную систему счисления, можно в одном байте сохранить любое целое число в диапазоне от 0 до 255 (от 00000000 до 11111111), а в двух байтах – уже любое целое число в диапазоне от 0 до 65 535. Это намного лучше, нежели сохранять в двух байтах только целые числа от 0 до 99, как при использовании для кодировки числа однобайтовых символов в кодах ASCII.

Представление изображений. Наиболее распространенные из существующих методов представления изображений можно разделить на две большие категории: *растровые методы* (bitmap techniques) и *векторные методы* (vector techniques). При растровом методе изображение представляется как совокупность точек, называемых *пикселями* (pixel, сокращение от picture element – элемент изображения). Говоря упрощенно, изображение кодируется в виде длинных строк битов, которые представляют цвет пикселей в изображении. Для черно-белых изображений каждый пиксель представляется 1 битом. При этом каждый бит равен 0 или 1, в зависимости от того, является ли соответствующий пиксель черным или белым.

Для передачи градаций серого цвета в черно-белых изображениях каждый пиксель кодируется комбинацией из 8 бит. Это позволяет передать 256 значений серого цвета.

Большинство периферийных устройств современных вычислительных машин, например факсимильные аппараты, видеокамеры или сканеры, преобразует цветные изображения в графические файлы с растровым форматом. Чаще всего эти устройства записывают цвет каждого пикселя, раскладывая его на три составляющие – красную (Red, R), зеленую (Green, G) и синюю (Blue, B), соответствующие трем первичным цветам. Такая система кодирования называется системой *RGB* по первым буквам основных цветов.

Для передачи интенсивности каждого цвета обычно используется один байт (или 24 двоичных разрядов на 1 пиксель). Это позволяет определить 16,5 млн. различных цветов. Такой режим представления цветной графики называется *полноцветным* (*True Color*).

При уменьшении количества двоичных разрядов до 16 на 1 пиксель удается передать 65 тыс. различных цветов. Такой режим представления цветной графики называется *High Color*.

Аналогичный трехкомпонентный пиксельный подход к передаче графической информации используется и при выводе изображений на экраны мониторов современных компьютеров. Экраны этих устройств содержат десятки тысяч пикселей, каждый из которых состоит из трех компонентов (красного, зеленого и синего), что можно заметить, воспользовавшись увеличительным стеклом, а иногда даже невооруженным глазом.

Формат "три байта на пиксель" означает, что для хранения изображения, в котором 1280 рядов по 1024 пикселя (фотография обычного размера), потребуется несколько мегабайт памяти, что существенно превышает размер стандартной дискеты. В разделе 1.6 будут рассмотрены два наиболее распространенных формата для сжатия подобных изображений до более приемлемых размеров (это форматы GIF и JPEG).

Одним из недостатков растровых методов 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, что также часто встречается и при фотографировании на пленку. Векторные методы позволяют избежать проблем масштабирования, характерных для растровых методов. В этом случае изображение представляется в виде совокупности линий и кривых. Вместо того, чтобы заставлять устройство воспроизводить заданную конфигурацию пикселей, составляющих изображение, ему передается подробное описание того, как расположены образующие изображение линии и кривые. На основе этих данных устройство, в конечном счете, и создает готовое изображение. С помощью подобной технологии описываются различные шрифты, поддерживаемые современными принтерами и мониторами. Они позволяют изменять размер символов в широких пределах и по этой причине получили название *масштабируемых шрифтов*. Например, технология True Type, разработанная компаниями Microsoft и Apple Computer, описывает способ отображения символов в тексте. Для подобных целей предназначена и технология PostScript (разработанная компанией Adobe Systems), позволяющая описывать способ отображения символов, а также других, более общих графических данных. Векторные методы также широко применяются в автоматизированных системах проектирования (computer-aided design, CAD), которые отображают на экране мониторов чертежи сложных трехмерных объектов и предоставляют средства

манипулирования ими. Однако векторная технология не позволяет достичь фотографического качества изображений объектов, как при использовании растровых методов. Именно поэтому в современных цифровых фотокамерах используются растровые методы представления изображения.

Представление звука. При наиболее распространенном способе кодирования звуковой информации амплитуда сигнала измеряется через равные промежутки времени и записываются полученные значения. Например, последовательность 0, 1.5, 2.0, 1.5, 2.0, 3.0, 4.0, 3.0, 0 описывает волну звука, амплитуда которой сначала увеличивается, затем немного уменьшается, затем снова повышается и, наконец, падает до 0 (рис. 1.18). Этот способ кодирования, в котором частота дискретизации составляет 8000 отсчетов в секунду, используется не первый год в дальней телефонной связи. Голос на одном конце канала кодировался в виде числовых значений, отражавших амплитуду звукового сигнала, восемь тысяч раз в секунду. Эти значения затем передавались по каналам связи и использовались для воспроизведения звука.

Может показаться, что 8000 отсчетов в секунду – это большая частота дискретизации, но она все же недостаточна для высокой точности воспроизведения музыки. Для получения качественного звучания на современных музыкальных компакт-дисках используется частота дискретизации, равная 44 100 отсчетов в секунду. Для данных, полученных при каждом отсчете, отводится 16 битов памяти (или 32 бита для стереозаписей). Следовательно, для хранения одной секунды звучания требуется более миллиона битов.

В музыкальных синтезаторах, компьютерных играх и звуковом фоне, сопровождающем Web-страницы, широко используется более экономная система кодирования, которая называется *цифровым интерфейсом музыкальных инструментов* (Musical Instrument Digital Interface – MIDI).

При использовании стандарта MIDI не требуется столько места в памяти, как при дискретизации звукового сигнала, так как эта система кодирует указания, как следует порождать музыку, а не сам звуковой сигнал.

Рис. 1.18. Звуковой сигнал, представленный последовательностью 0, 1.5, 2.0, 1.5, 2.0, 3.0, 4.0, 3.0, 0

Точнее, MIDI кодирует информацию о том, какой инструмент должен играть, какую ноту и какова продолжительность звучания этой ноты. Это означает, что для кларнета, играющего ноту ре в течение двух секунд, потребуются три байта, а не более двух миллионов битов, как в случае дискретизации сигнала с частотой 44 100 отсчетов.

Можно сказать, что стандарт MIDI скорее похож на нотную запись, которую читает исполнитель, чем на само исполнение. Издержки метода – музыкальная запись в стандарте MIDI может звучать по-разному в исполнении различных музыкальных синтезаторов.

Вопросы для самопроверки

1. Ниже приведено сообщение, представленное в виде символов ASCII с использованием восьми битов на один символ. Какой текст этого сообщения?

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 в кодировке ASCII?

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

а) Where are you?

б) "How" Cheryl asked.

в) 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, а другому – значение 0, а затем покажите, как будет выглядеть код ASCII для буквы b, если представить ее таким способом.

5. 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?

6. В чем состоят преимущества векторной графики по сравнению с растровой? Каковы преимущества растровой графики?

7. Предположим, что стереозапись одного часа музыки закодирована с частотой дискретизации 44100 отсчетов в секунду. Как размер закодированного сигнала соотносится с емкостью компакт-диска?

1.6. СЖАТИЕ ДАННЫХ*

При записи или передаче данных часто бывает полезно (а иногда просто необходимо) сократить размер обрабатываемых данных. Технология, позволяющая достичь этой цели, называется *сжатием данных*. В этом разделе рассмотрены некоторые общие методы сжатия данных, а также конкретные приемы, разработанные специально для сжатия изображения.

Универсальные методы сжатия данных. Существует множество методов сжатия данных, каждый из которых характеризуется собственной областью применения, в которой он дает наилучшие или, наоборот, наихудшие результаты. Метод *кодирования длины серий* (run-length encoding) дает наилучшие результаты, если сжимаемые данные состоят из длинных последовательностей одних и тех же значений. В сущности, такой метод кодирования как раз и состоит в замене подобных последовательностей кодовым значением, определяющим 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.

В некоторых случаях информация 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* (relative encoding). 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.

Еще один метод сжатия данных предполагает применение *частотно-зависимого кодирования* (frequency-dependent encoding), при котором длина битовой комбинации, представляющей элемент данных, обратно пропорциональна частоте использования этого элемента. Такие коды входят в группу *кодов переменной длины* (variable-length codes), т.е. элементы данных в этих кодах представляются битовыми комбинациями различной длины. Если взять английский текст, закодированный с помощью частотно-зависимого метода, то чаще всего встречающиеся символы (e, t, a, i) будут представлены короткими битовыми комбинациями, а те знаки, которые встречаются реже (z, q, x), – более длинными битовыми комбинациями. В результате мы получим более короткое представление всего текста, чем при использовании обычного кода, подобного Unicode или ASCII. Построение алгоритма, который обычно используется при разработке частотно-зависимых кодов, приписывают Дэвиду Хоффману (David Huffman), поэтому такие коды часто называются *кодами Хоффмана* (Huffman codes). Большинство используемых сегодня частотно-зависимых кодов является кодами Хоффмана.

Хотя обсуждавшиеся выше методы кодирования были представлены как технологии сжатия данных общего назначения, тем не менее, каждый из них имеет собственную сферу применения. В противоположность этому, системы, основанные на использовании *метода кодирования Lempel-Ziv* (названного в честь его создателей Абрахама Лемпеля (Abraham Lempel) и Джэкоба Зива (Jacob Ziv), действительно являются системами сжатия данных общего назначения. Многие пользователи Internet (глава 3), несомненно, уже встречали и даже использовали такие универсальные программы сжатия данных произвольного типа, как zip и unzip, в которых применяется технология Lempel-Ziv.

Системы кодирования по методу Lempel-Ziv используют технологию *кодирования с применением адаптивного словаря* (adaptive dictionary encoding). В данном контексте термин "словарь" означает набор строительных блоков, из которых создается сжатое сообщение. Если сжатию подвергается английский текст, то строительными блоками могут быть символы алфавита. Если потребуется уменьшить размер данных, которые хранятся в компьютере, то компоновочными блоками могут стать нули и единицы. В процессе адаптивного словарного кодирования содержание словаря может изменяться. Например, при сжатии английского текста может оказаться целесообразным добавить в словарь окончание ing и артикль the. В этом случае место, занимаемое будущими копиями окончания ing и артикля the, может быть уменьшено за счет записи их как одиночных ссылок вместо сочетания из трех разных ссылок. Системы кодирования по методу Lempel-Ziv используют изощренные и весьма эффективные методы адаптации словаря в процессе кодирования (или сжатия). В частности, в любой момент процесса кодирования словарь будет состоять из тех комбинаций, которые уже были закодированы (сжаты).

В качестве примера давайте рассмотрим, как можно выполнить сжатие сообщения с использованием конкретной системы метода Lempel-Ziv, известной как LZ77. Процесс начинается практически с переписывания начальной части сообщения, однако в определенный момент осуществляется переход к представлению будущих сегментов с помощью триплетов, каждый из которых будет состоять из двух целых чисел и следующего за ними одного символа текста. Каждый триплет описывает способ построения следующей части сообщения. Например, пусть распакованный текст имеет следующий вид (символы греческого алфавита здесь использованы для того, чтобы данному примеру не придавался никакой определенный смысл):

αβααβρβ (5, 4, α)

Строка αβααβρβ является уже распакованной частью сообщения. Для того чтобы разархивировать остальной текст сообщения, необходимо сначала расширить строку, присоединив к ней ту часть, которая в ней уже встречается (рис. 1.19). Первый номер в триплете указывает, сколько символов необходимо отсчитать в обратном направлении в строке, чтобы найти первый символ добавляемого сегмента.

а)

α β α α β ρ β

б)

α β α α β ρ β α α β ρ

в)

α β α α β ρ β α α β ρ α

г)

Рис. 1.19. Процесс распаковки сообщения αβααβρβ (5, 4, α):
а – отсчет 5-ти символов назад; *б* – определение сегмента из 4-х символов, который должен быть добавлен в конец строки; *в* – копирование сегмента из 4-х символов в конец сообщения; *г* – добавление в конец сообщения символа, указанного в триплете

В данном случае необходимо отсчитать в обратном направлении 5 символов, и мы попадем на второй слева символ α уже распакованной строки. Второе число в триплете задает количество последовательных символов справа от начального, которые составляют добавляемый сегмент. В нашем примере это число 4, и это означает, что добавляемым сегментом будет ααβρ. Копируем его в конец строки и получаем новое значение распакованной части сообщения:

αβααβρβααβρ

Наконец, последний элемент (в нашем случае это символ α) должен быть помещен в конец расширенной строки, в результате чего получаем полностью распакованное сообщение:

αβααβρβααβρα

Теперь предположим, что сжатая версия текста имеет такой вид:

αβααβρβ (5,4, α)(0,0,δ)(8,6, β)

Вначале распакуем первый триплет, в результате чего получим сообщение следующего вида:

αβααβρβααβρα (0,0,δ) (8,6, β)

Теперь распакуем второй триплет и получим следующий результат:

αβααβρβααβραδ (8,6, β)

Обратите внимание, что второй триплет (0,0, δ) использовался только потому, что символ δ еще не встречался в этом тексте. И наконец, распакуем третий триплет и получим полностью распакованное сообщение:

αβααβρβααβραδρβααβρβ

Чтобы запаковать сообщение с использованием системы LZ77, сначала необходимо записать начальный сегмент текста, а затем искать в нем наиболее длинный сегмент, соответствующий очередному фрагменту оставшейся части сообщения. Это будет комбинация, описываемая первым триплетом. Все последующие триплеты строятся по тому же методу.

Может показаться, что приведенные примеры не демонстрируют значительного сжатия, поскольку все триплеты описывают лишь небольшие сегменты сообщения. Однако при работе с длинными битовыми комбинациями есть основания полагать, что достаточно длинные сегменты данных будут представлены единственными триплетами, что приведет к значительному сжатию данных.

Сжатие изображений. В разделе 1.5 было показано, что растровый формат, используемый в современных цифровых преобразователях изображений, предусматривает кодирование изображения в формате по три байта на пиксель, что приводит к созданию громоздких, неудобных в работе растровых файлов. Специально для этого формата было разработано множество схем сжатия, предназначенных для уменьшения места, занимаемого подобными файлами 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. С помощью таблицы, называемой цветовой палитрой, каждый из допустимых цветовых оттенков пикселя ассоциируется с некоторой комбинацией цветов "красный-зеленый-синий". Изменяя используемую палитру, можно изменять цвета, появляющиеся в изображении.

Обычно один из цветов палитры в формате GIF воспринимается как обозначение "прозрачности". Это означает, что в закрашенных этим цветом участках изображения отображается цвет того фона, на котором оно находится. Благодаря этому и относительной простоте использования изображений формат GIF получал широкое распространение в тех компьютерных играх, где множество различных картинок перемещается по экрану.

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. Именно по этой причине данный стандарт используется производителями современных цифровых фотокамер. Следует ожидать, что он окажет немалое влияние на область цифрового представления изображений и в будущем.

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. В этом режиме экономия места достигается посредством запоминания различий между последовательными пикселями, а не яркости каждого пикселя в отдельности. Согласно теории, в большинстве случаев степень различия между соседними пикселями может быть закодирована более короткими битовыми комбинациями, чем собственно значения яркости отдельных пикселей. Существующие различия кодируются с помощью кода переменной длины, который применяется в целях дополнительного сокращения используемой памяти.

К сожалению, при использовании режима "без потерь" создаваемые файлы растровых изображений настолько велики, что они с трудом обрабатываются методами современной технологии, а потому и применяются на практике крайне редко. Большинство существующих приложений использует другой стандартный метод формата JPEG – режим "базовых строк". В этом режиме каждый из пикселей также представляется тремя составляющими, но в данном случае это уже один компонент яркости и два компонента цвета. Грубо говоря, если создать изображение только из компонентов яркости, то мы увидим черно-белый вариант изображения, так как эти компоненты отражают только уровень освещенности пикселя.

Смысл подобного разделения между цветом и яркостью объясняется тем, что человеческий глаз более чувствителен к изменениям яркости, чем цвета. Рассмотрим, например, два равномерно окрашенных синих прямоугольника, которые абсолютно идентичны, за исключением того, что на один из них нанесена маленькая яркая точка, тогда как на другой – маленькая зеленая точка той же яркости, что и синий фон. Глазу проще будет обнаружить яркую точку, а не зеленую. Режим "базо-

вых строк" стандарта JPEG использует эту особенность, кодируя компонент яркости каждого пикселя, но усредняя значение цветовых компонентов для блоков, состоящих из четырех пикселей, и записывая цветовые компоненты только для этих блоков. В результате окончательное представление изображения сохраняет внезапные перепады яркости, однако оставляет размытыми резкие изменения цвета. Преимущество этой схемы состоит в том, что каждый блок из четырех пикселей представлен только шестью значениями (четыре показателя яркости и два – цвета), а не двенадцатью, которые необходимы при использовании схемы из трех показателей на каждый пиксель.

Дополнительной экономии места можно достичь с помощью записи информации, определяющей изменения компонентов яркости и цвета, а не их абсолютных значений. В этом случае, как и в режиме "без потерь" формата JPEG, отправной точкой является тот факт, что при сканировании изображения уровень различий между соседними пикселями может быть закодирован с использованием меньшего количества битов, чем при записи самих характеристик отдельных пикселей. (В действительности эти изменения кодируются с помощью математического метода, называемого дискретным косинусным преобразованием и применяемого к блокам размером 8 × 8 пикселей.) Полученная в результате битовая комбинация дополнительно сжимается с использованием кодов переменной длины. В результате применение режима "базовых строк" формата JPEG позволяет получать цветные изображения приемлемого качества, размер которых находится в соотношении 1 : 20 с размером растровых файлов, в которых для представления каждого пикселя используется трехбайтовая схема, используемая в большинстве существующих сканеров.

То, что режим "базовых строк" формата JPEG позволяет существенно сократить размеры файлов за счет незначительного и практически незаметного снижения качества изображения, сделало этот формат очень популярным среди пользователей. Однако в некоторых случаях использование других методов дает лучшие результаты. Например, формат GIF позволяет лучше представлять изображения, состоящие из блоков одного цвета с четкими границами (как, например, в цветной мультипликации).

В заключение следует отметить, что сейчас в области сжатия данных проводятся интенсивные и обширные исследования. Мы обсудили лишь два из множества существующих методов сжатия изображений. А ведь, помимо них, имеются еще многочисленные методы сжатия звука и видеоизображений. Например, метод, подобный режиму "базовых строк" формата JPEG, был разработан входящей в состав ISO ассоциацией Motion Picture Experts Group (MPEG) и принят в качестве стандарта кодирования (или сжатия) движущихся изображений. Суть этого стандарта состоит в записи начальной картинки последовательности изображений с помощью метода, подобного режиму "базовых строк" формата JPEG, после чего для кодирования оставшейся части изображений в их последовательности применяются методы относительного кодирования.

Как уже отмечалось в разделе 1.5, одна секунда музыкального звучания, оцифрованная с частотой дискретизации 44 100 отсчетов в секунду, требует более одного миллиона битов в памяти. Подобные затраты памяти приемлемы для записи музыки на компакт-дисках, однако в сочетании с видеозаписью (для получения движущихся озвученных изображений) эти требования превышают возможности современной технологии. Поэтому ассоциация Motion Picture Experts Group разработала методы сжатия звука, позволяющие существенно снизить требования к использованию памяти. Одним из таких форматов является MP3 (MPEG-1, Audio Layer-3), позволяющий сжимать аудиоинформацию в соотношении 12 : 1. При использовании этого формата музыкальные записи сжимаются до таких размеров, которые позволяют эффективно пересылать их по Internet.

Вопросы для самопроверки

1. Ниже представлен текст сообщения, сжатый с использованием метода LZ77. Как будет выглядеть распакованное сообщение?

101101011 (7, 5, 0) (12, 10, 1) (18, 13, 0)

2. Несмотря на то что мы не очень подробно рассматривали алгоритм кодирования данных по методу LZ77, все же попытайтесь выполнить сжатие следующего сообщения:

ββαβββααβαβααβαβααβαβααα

3. Выше утверждалось, что формат GIF позволяет лучше представлять цветные мультипликационные изображения, чем формат JPEG. Объясните, почему это действительно так.

4. Какое наибольшее количество байтов потребуется для представления изображения размером 1024 × 1024 пикселей, если использовать формат GIF? Что можно сказать относительно использования режима "базовых строк" формата JPEG?

5. Какие особенности человеческого глаза используются в режиме "базовых строк" формата JPEG?

1.7. 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, вероятнее всего, полученная, в конце концов, 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. 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, большую 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, необхо-

димо создать систему, в которой любая битовая комбинация будет содержать нечетное количество единиц. Обычно это достигается путем добавления к уже существующему коду дополнительного бита, который называется *битом четности* или *контрольным битом* (parity bit), чаще всего помещаемого в старший конец комбинации. В результате восьмиразрядный код ASCII превращается в девятиразрядный, а шестнадцатиразрядная битовая комбинация в двоичном дополнительном коде становится семнадцатиразрядной. В каждом случае значение бита четности устанавливается равным 0 или 1, исходя из требования, чтобы вся битовая комбинация в целом содержала нечетное количество единиц. Как показано на рис. 1.20, символ A в коде ASCII будет представлен как 101000001 (бит четности равен 1), тогда как символ F в этом же коде 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 система будет построена указанным образом, то появление битовой комбинации

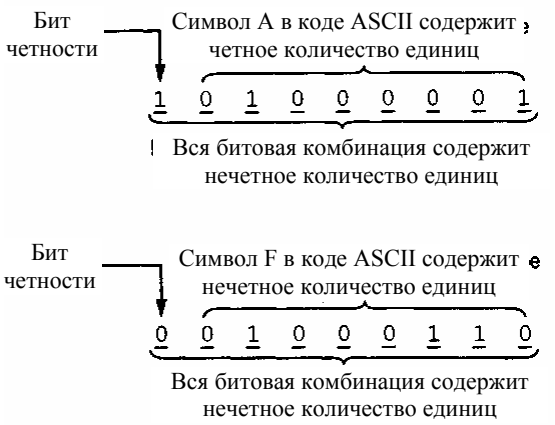


Рис. 1.20. Представление символов A и F в коде ASCII с использованием контрольного бита

с четным количеством единиц будет свидетельствовать об ошибке и сигнализировать, что обрабатываемые данные являются неверными. Описанная выше система контроля четности называется *проверкой на нечетность* (odd parity), поскольку все обрабатываемые битовые комбинации должны содержать нечетное количество единиц. Кроме того, существует способ, именуемый *проверкой на четность* (even parity). 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 внешне 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 исходного.

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 коды с исправлением ошибок, позволяющие не только выявлять ошибки, но и исправлять их.

В конце концов, интуиция подсказывает, что мы не в состоянии исправить ошибку в полученном сообщении, если зара-

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

Рис. 1.21. Код с исправлением ошибок

нее не знаем, о чем там идет речь. Тем не менее, существует довольно простой код, позволяющий исправлять возникающие ошибки (рис. 1.21).

Для того чтобы понять принцип действия этого кода, сначала необходимо определить *дистанцию Хэмминга* (Hamming

distance) между двумя кодовыми комбинациями, которая будет равна количеству битов, отличающихся 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 А и В (рис. 1.21) равна четырем, а дистанция Хэмминга между кодами букв В и С 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.

Если в любой комбинации, показанной на рис. 1.21, 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. При декодировании некоторого сообщения достаточно просто сравнивать каждую полученную 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. 1.22. По содержанию этой таблицы можно сделать заключение, что поступивший символ – это буква D, так как ее битовая комбинация в наибольшей степени соответствует полученной.

Как видите, при использовании кода, представленного на рис. 1.22, 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.

Символ	Дистанция между полученной комбинацией битов и кодом его символа	
A	2	
B	4	
C	3	
D	1	Наименьшая дистанция
E	3	
F	5	
G	2	
H	4	

Рис. 1.22. Декодирование битовой комбинации 010100 с помощью кода, представленного на рис. 1.21

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 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 CD-DA 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 20 000 компакт-дисков.

Вопросы для самопроверки

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

а) 10101101; б) 10000001; в) 00000000; г) 11100000; д) 11111111.

2. Могли бы Вы не заметить ошибки в байтах, представленных в первом вопросе? 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 предложения в коде ASCII с использованием контроля по нечетности и помещением контрольного бита в старший конец кода каждого символа:

а) Where are you?

б) "How?" Chery lasked.

в) 2 + 3 = 5.

5. Используйте представленный на рис. 1.21 код с исправлением ошибок для декодирования следующих сообщений:

а) 001111 100100 001100

б) 010001 000000 001011

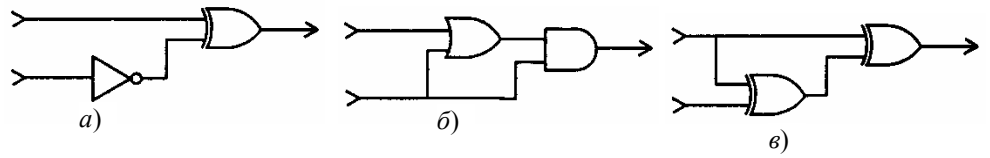
в) 011010 110110 100000 011100

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.

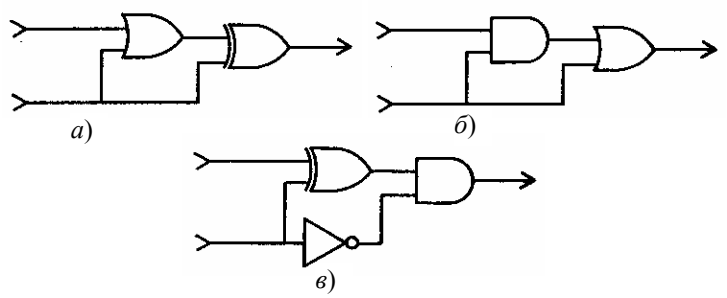
Упражнения.

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

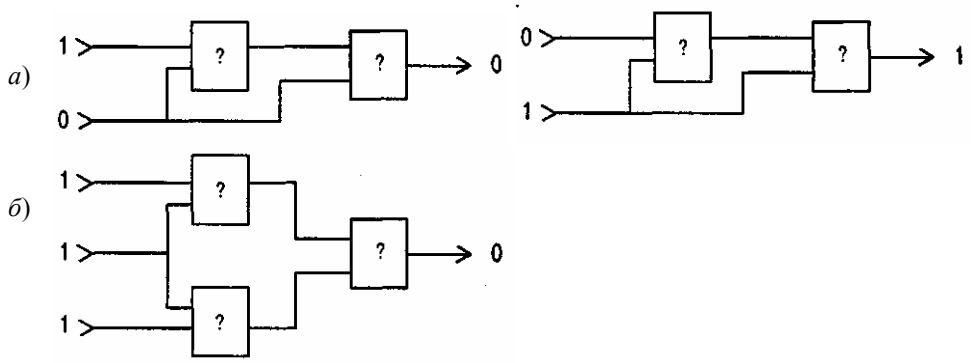
1. Какое выходное значение будет иметь каждая из приведенных ниже схем, если предположить, что на их верхний вход подано значение 1, а на нижний – значение 0.



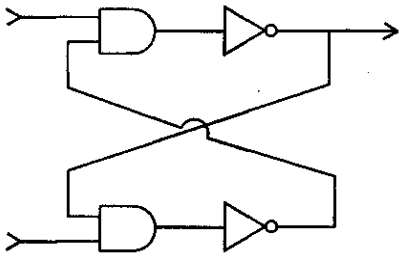
2. Для каждой из приведенных ниже схем укажите комбинации входных значений, при которых их выходное значение будет равно 1.



3. В каждой из приведенных ниже схем прямоугольники представляют вентили одного и того же типа. Основываясь на показанных входных и выходных значениях этих схем, определите в каждом случае тип используемого вентиля (AND, OR или XOR).



4. Предпол*в*) им, что на оба входа приведенной ниже цепи поданы значения 1. Опишите, что произойдет, если на верхний вход временно подать значение 0. Что произойдет, если временно подать значение 0 на нижний вход?



6. 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
01	AB
02	53
03	D6
04	02

Этап 1. Переместите содержимое ячейки с адресом 03 в ячейку с адресом 00.

Этап 2. Поместите число 01 в ячейку с адресом 02.

Этап 3. Переместите значение, сохраняемое по адресу 01, в ячейку с адресом 03.

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

7. Какие комбинации битов представлены следующими шестнадцатеричными обозначениями?

а) BC; б) 67; в) 9A; г) 10; д) 3F.

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

а) FF; б) 7F; в) 8F; г) 1F.

9. Представьте следующие битовые комбинации в шестнадцатеричной системе счисления:

а) 101010101010; б) 110010110111; в) 000011101011.

10. Предположим, что на экране монитора отображены 24 строки, содержащие по 80 символов каждая. Сколько байтов машинной памяти потребуется, чтобы записать в нее все содержимое экрана, представив каждый символ с помощью соответствующего кода ASCII (по одному символу на байт)?

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 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 дисковым носителям информации.

б) Назовите два преимущества, которые имеют дисковые носители информации по отношению к основной памяти машины.

13. Предположим, вам необходимо подготовить на компьютере курсовую работу объемом приблизительно 40 машинописных листов. В компьютере имеется дисковое устройство чтения дискет диаметром $3^1/_2$ дюйма, емкость которых составляет 1,44 Мбайт. Поместится ли ваша курсовая работа на одной такой дискете? Если да, то сколько таких документов может поместиться на одной дискете? Если нет, то сколько дискет потребуется для сохранения вашей курсовой работы?

14. Предположим, что на жестком диске Вашего компьютера, емкость которого составляет 5 Гбайт, осталось только 100 Мбайт дискового пространства. Допустим, вы намерены заменить его жестким диском большей емкости (10 Гбайт) и на время модернизации предполагаете сохранить всю имеющуюся на жестком диске информацию на дискетах размером $3^1/_2$ дюйма. Разумно ли это? Поясните свой ответ.

15. Если каждый сектор диска содержит 512 байт, то сколько секторов потребуется для записи одной печатной страницы текста, если для хранения одного символа этого текста необходим один байт?

16. Обычная дискета диаметром $3^1/_2$ дюйма имеет емкость 1,44 Мбайт. Достаточно ли одной такой дискеты для сохранения 400 страниц текста, если каждая страница содержит 3500 символов?

17. Если дискета, которая содержит 16 секторов на каждой дорожке и 512 байт в каждом секторе, вращается со скоростью 300 оборотов в минуту, то какой приблизительно будет скорость передачи данных через головку чтения/записи устройства?

18. Если настольный компьютер содержит дисковод, описанный в упражнении 17, и выполняет десять операций за одну микросекунду (миллионную долю секунды), то сколько операций он может выполнить между прохождением двух последовательных байтов данных через головку чтения/записи дисковода?

19. Если дискета вращается со скоростью 300 оборотов в минуту, а компьютер способен выполнить 100 операций в микросекунду (миллионная доля секунды), то сколько операций сможет выполнить этот компьютер за время ожидания доступа диска?

20. Сравните время ожидания доступа обычной дискеты, описанной в упражнении 19, и время ожидания доступа жесткого диска со скоростью вращения 60 оборотов в секунду.

21. Каким будет среднее время доступа жесткого диска, вращающегося со скоростью 60 оборотов в секунду, если его время поиска равно 10 миллисекундам?

22. 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.

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

01010111	01101000	01100001
01110100	00100000	01100100
01101111	01100101	01110011
00100000	01101001	01110100
00100000	01110011	00110001
01111001	00111111	

24. 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.

6865786164656369 6D616C

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

а) 100/5 = 20.

б) To be or not to be?

в) The total cost is $7.25.

26. Представьте ответ из упражнения 25 в шестнадцатеричной системе счисления.

27. Перечислите двоичные представления для целых чисел от 6 до 16.

28. а) Запишите число 13, представив цифры 1 и 3 в коде ASCII;

б) Запишите число 13 в двоичной системе счисления.

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

30. Преобразуйте приведенные ниже двоичные числа в эквивалентные десятичные значения:

а) 111; б) 0001; в) 11101;

г) 10001; д) 10111; е) 000000;

ж) 100; з) 1000; и) 10000;
к) 11001; л) 11010; м) 11011.

31. Представьте следующие десятичные числа в двоичном представлении:

а) 7; б) 12; в) 16; г) 15; д) 33.

32. Представьте в десятичной системе счисления приведенные ниже числа, записанные в двоичном дополнительном коде:

а) 10000; б) 10011; в) 01104; г) 01111; д) 10111.

33. Представьте приведенные ниже десятичные числа в двоичном дополнительном коде разрядностью 7 битов:

а) 12; б) –12; в) –1; г) 0; д) 8.

34. 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) $+\begin{array}{r}00101\\ \underline{01000}\end{array}$; б) $+\begin{array}{r}01111\\ \underline{00001}\end{array}$; в) $+\begin{array}{r}11111\\ \underline{00001}\end{array}$; г) $+\begin{array}{r}10111\\ \underline{11010}\end{array}$; д) $+\begin{array}{r}00111\\ \underline{00111}\end{array}$;

е) $+\begin{array}{r}00111\\ \underline{01100}\end{array}$; ж) $+\begin{array}{r}11111\\ \underline{11111}\end{array}$; з) $+\begin{array}{r}01010\\ \underline{10101}\end{array}$; и) $+\begin{array}{r}01000\\ \underline{01000}\end{array}$; к) $+\begin{array}{r}01010\\ \underline{00011}\end{array}$.

35. 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) $+\begin{array}{r}7\\ \underline{1}\end{array}$; б) $-\begin{array}{r}7\\ \underline{1}\end{array}$; в) $-\begin{array}{r}12\\ \underline{4}\end{array}$; г) $-\begin{array}{r}8\\ \underline{7}\end{array}$; д) $+\begin{array}{r}12\\ \underline{4}\end{array}$; е) $+\begin{array}{r}4\\ \underline{11}\end{array}$.

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

а) 10000; б) 10011; в) 01101; г) 01111; д) 10111.

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

а) 0; б) 3; в) –3; г) –1; д) 1.

38*. Какие из перечисленных ниже битовых комбинаций не являются допустимыми значениями в двоичной нотации с избытком шестнадцать?

а) 01001; б) 101; в) 010101; г) 00000;
д) 1000; е) 000000; ж) 1111.

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

а) 11.001; б) 100.1101; в) 0.0101; г) 1.0; д) 10.01.

40*. Запишите приведенные ниже числа в двоичной системе счисления:

а) 5 ¾; б) $^1/_{16}$; в) $7\ ^7/_8$; г) $1^1/_4$; д) $6^5/_8$.

41*. Декодируйте следующие битовые комбинации в формате с плавающей точкой:

а) 01011100; б) 11001000; в) 00101010; г) 10111001.

42*. Закодируйте приведенные ниже числа, используя восьмиразрядный формат с плавающей точкой. Укажите на ошибки усечения:

а) $^1/_2$; б) $7^1/_2$; в) $-3^3/_4$; г) $^3/_{32}$; д) $^{31}/_{32}$.

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

44*. Какое наилучшее приближение для значения числа $^1/_{10}$ может быть достигнуто в восьмиразрядном формате с плавающей точкой?

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

46*. Используйте восьмиразрядный формат с плавающей точкой для вычисления следующей операции суммирования: $^1/_8 + ^1/_8 + ^1/_8 + 2^1/_2$; при условии, что она выполняется слева направо. Что изменится в полученном результате, если вычисления производить справа налево?

47*. Какой ответ получит машина при вычислении приведенных ниже примеров суммирования, если в ней используется восьмиразрядный формат с плавающей точкой?

а) $1^1/_2 + ^3/_{16} =$

б) $3^1/_4 + 1^1/_8 =$

в) $2^1/_4 + 1^1/_8 =$

48*. Для каждого приведенного ниже примера преобразуйте битовые комбинации (интерпретируя их как числа в формате с плавающей точкой) в десятичные значения, выполните сложение, а затем закодируйте ответ в тот же формат с плавающей точкой. Укажите на ошибки усечения.

а) $+\begin{array}{r}01011100\\ \underline{01101000}\end{array}$; б) $+\begin{array}{r}01101010\\ \underline{00111000}\end{array}$; в) $+\begin{array}{r}01111000\\ \underline{00011000}\end{array}$; г) $+\begin{array}{r}01011000\\ \underline{01011000}\end{array}$.

49*. 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?

50*. 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; однако не обязательно в указанном порядке. Определите это число и укажите, какая из битовых комбинаций принадлежит к каждому из перечисленных форматов.

51*. Каждая из приведенных ниже строк битов представляет одно и то же число, записанное в различных системах кодирования, обсу-ждавшихся нами выше. Определите каждое из чисел и укажите использованную в каждом случае систему кодирования.

а) 111110100 011 1011

б) 11111101 01111101 11101100

в) 10100010 01101000

52*. Какие из приведенных ниже чисел невозможно точно представить в формате с плавающей точкой?

а) $6\ ^1/_2$; б) 9; в) $1^3/_{16}$; г) $^{17}/_{32}$; д) $^{15}/_{16}$.

53*. Если увеличить длину битовой строки, используемой для представления целых чисел в двоичной системе, от четырех до восьми бит, 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?

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

55*. Приведенное ниже сообщение было сжато по методу LZ77. Распакуйте это сообщение.

0100101 (4, 3, 0) (8, 7, 1) (17, 9, 1) (8, 6, 1)

56*. Ниже приведена часть сообщения, закодированного по методу LZ77. Исходя из содержащейся в данном представлении информации, определите длину исходного сообщения.

αβγ (_,3, β)(_,6, γ)

57*. Запишите последовательность инструкций, поясняющих способ сжатия сообщений по методу LZ77.

58*. Запишите последовательность инструкций, поясняющих способ распаковки сообщений, сжатых по методу LZ77.

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

а) 100/5 = 20;

б) To be or not to be?

в) The total cost is $7.25.

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

11011 01011 10110 00000 11111 10101

10001 00100 01110

61*. 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?

62*. Для декодирования приведенных ниже слов используйте код с исправлением ошибок, представленный на рис. 1.21:

а) 111010 110110;

б) 101000 100000 001100;

в) 011101 000110 000000 010100;

г) 010010 001000 001110 101111;

д) 000000 110111 100110;

е) 010011 000000 101001 100110.

63*. Используя вентили, разработайте такую схему с четырьмя входами и одним выходом, чтобы значение на выходе было равно нулю или единице, в зависимости от того, является ли четырехразрядная комбинация на входах этой схемы четной или нечетной.

Ответы на вопросы для самопроверки

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. 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. Триггер будет закрыт, когда значение сигнала синхронизации будет равно 0, и открыт, когда значение сигнала синхронизации будет равно 1.

5. В первом случае после завершения операции ячейка памяти с адресом 6 будет содержать значение 5. Во втором случае эта ячейка будет иметь значение 8.

6. В результате выполнения первой же операции предыдущее значение в ячейке с адресом 3 будет удалено, поскольку новое значение записывается поверх него. Следовательно, после выполнения второй операции предыдущее значение из ячейки с адресом 3 не будет перенесено в ячейку с адресом 2. В результате обе ячейки будут содержать значение, которое исходно хранилось в ячейке с адресом 2. Правильная процедура должна быть следующей.

Шаг 1. Переместить содержимое ячейки с адресом 2 в ячейку с адресом 1.

Шаг 2. Переместить содержимое ячейки с адресом 3 в ячейку с адресом 2.

Шаг 3. Переместить содержимое ячейки с адресом 2 в ячейку с адресом 3.

7. 32 768 бит.

Раздел 1.2

1. Более высокая скорость считывания и передачи данных.

2. Здесь следует вспомнить, что механическое движение намного медленнее электронных процессов, протекающих в компьютере. Это вынуждает нас минимизировать количество перемещений магнитных головок. Если необходимо полностью заполнять поверхность данными перед тем, как перейти к следующей поверхности, придется перемещать магнитную головку каждый раз, когда очередная дорожка будет закончена. 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 объем данных постоянно то увеличивается, то уменьшается. Если бы информация хранилась на ленте, это привело бы к бесконечному перезаписыванию данных. При этом остаток записей постоянно записывался бы туда и обратно вследствие сдвига записей при обновлении данных или их удалении. Однако при хранении данных на диске каждое изменение происходит только в том блоке данных, который располагается в соответствующем секторе. Следовательно, при обновлении данных понадобится намного меньше перезаписей.

4. Распределение логических записей по разным секторам означает, что для получения полной логической записи понадобится прочитать с диска несколько секторов. Затраты времени на чтение этих дополнительных секторов легко может перевесить преимущества экономии дискового пространства.

Раздел 1.3

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) 0011 (переполнение); в) 0100 (переполнение); г) 0001; д) 1000 (переполнение).

7. а) 0110; б) 0011; в) 0100; г) 0010; д) 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) б, 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 –> 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 бит), 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 1.4

1. а) $^{5}/_{8}$; б) $^{31}/_{4}$; в) $^{9}/_{32}$; г) $-1^{1}/_{2}$; д) $-^{11}/_{64}$.

2. а) 01101011; б) 01111010 (ошибка округления); в) 01001100; г) 11101110; д) 11111000 (ошибка округления).

3. 01001001 ($^{9}/_{16}$) больше 00111101 ($^{13}/_{32}$). Ниже приведен простой способ определения, какой из двоичных кодов представляет собой большее число.

Случай 1. Если знаковые биты чисел разные, тогда из двух чисел больше то, у которого знаковый бит 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, у которого в этой позиции стоит 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, у которого в этой позиции стоит 0, представляет собой большее число.

Простота этого способа сравнения двух чисел является одной из причин, по которой экспоненты чисел с плавающей точкой представляются в двоичной нотации с избытком, а не в дополнительном коде.

4. Большим значением было бы $7^{1}/_{2}$, 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 1.5

1. Computer science (Компьютерные науки).

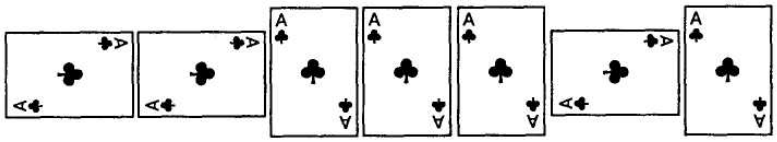
2. Две эти битовые комбинации практически одинаковы, за исключением того, что шестой бит, считая от младших разрядов к старшим, всегда равен 0 для прописных букв и 1 – для строчных.

3. а) 01010111 01100101 01100101 01110101
01101000 00100000 00100000 00111111
01100101 01100001 01111001
01110010 01111001 01101111;

б) 00100010 01001000 01101111 01110111
00111111 00100010 00100000 01000011
01101000 01100101 01110010 01111001
01101100 00100000 01100001 01110011
01101011 01100101 01100100 00101110;

в) 00110010 00101011 00110011 00111101
00110101 00101110.

4.



5. В 24 битах можно хранить три символа в кодировке ASCII, 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.

6. Векторные методы лучше реагируют на изменения масштаба, чем растровые. Простые чертежи в этом формате занимают меньше памяти. С другой стороны, векторные методы не могут обеспечить того же фотографического качества, что и растр.

7. При записи с частотой 441 000 значений в секунду один час звучания потребует 635 040 000 байт для хранения. Это практически заполнит компакт-диск, емкость которого 700 Мбайт.

Раздел 1.6

1. В шестнадцатеричном представлении сообщение будет иметь вид
B5E95EFA56.

2. Ответы могут быть различными. Один из возможных вариантов – $\beta\beta\alpha\beta\beta\beta\alpha\alpha\beta\alpha$ (5, 5, β) (10, 7, α).

3. Цветные мультфильмы состоят из блоков сплошного цвета с резкими контурами. Кроме того, количество используемых цветов ограничено.

4. Может существовать до 1 049 576 пикселей, для кодирования цвета каждого из которых потребуется 1 байт. Таким образом, максимальный размер изображения в формате GIF – 1 Мбайт. Этот размер можно значительно уменьшить за счет применения при кодировании изображений в формате GIF дополнительных алгоритмов сжатия информации, хотя их эффективность зависит от сложности изображения. При кодировании в формате GIF простого изображения размер файла обычно не превосходит нескольких килобайт. Если же используется стандарт JPEG, каждый блок пикселей размером 2×2 элемента

требует определения только 6 компонентов. Если предположить, что одному компоненту соответствует один байт, то для кодирования изображения размером 1024 × 1024 пикселей потребуется максимум 1,5 Мбайт дискового пространства, однако с помощью дополнительных методов сжатия размер файла изображения можно будет сократить до 100 кбайт и меньше.

5. Стандарт JPEG использует преимущество того, 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.

Раздел 1.7

1. Пункты *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) 001010111 001101000 101100101
101110010 101100101 100100000
001100001 101110010 101100101
000100000 001111001 101101111
001110101 100111111;

б) 100100010 101001000 101101111
101110111 100111111 100100010
000100000 001000011 001101000
101100101 101110010 001111001
101101100 001000000 001000001
001110011 001101011 101100101
001100100 100101110;

в) 000110010 100101011 100110011
000111101 100110101 100101110.

5. а) BED; б) CAB; в) HEAD.
6. Одно из решений таково:

A 0 0 0 0 0
B 1 1 1 0 0
C 0 1 1 1 1
D 1 0 0 1 1

2. ОБРАБОТКА ДАННЫХ

В главе 1 мы познакомились с методами хранения данных и организацией памяти компьютера. Кроме способности хранить данные, компьютер должен обладать способностью обрабатывать их так, как это предписано алгоритмом. Это значит, что машина должна иметь средства выполнения операций над данными и средства контроля последовательности этих операций. Такие задачи выполняются устройством, которое называется центральным процессором. Изучению именно этого устройства и связанных с ним вопросов посвящена данная глава.

2.1. ЦЕНТРАЛЬНЫЙ ПРОЦЕССОР

Состав центрального процессора. Электронные цепи типичного компьютера, предназначенные для выполнения различных операций с данными (например, сложения или вычитания), обычно не связаны с ячейками основной памяти напрямую. Все эти цепи размещаются в изолированной части компьютера, которая называется *центральным процессором* (Central Processing Unit – CPU), или ЦП (часто просто процессор). Данное устройство состоит из двух частей: *арифметико-логического* блока (arithmetic/logic unit), включающего схемы для обработки данных, и *блока управления* (control unit), который содержит схемы, координирующие деятельность всей машины.

Для временного запоминания информации в ЦП имеются ячейки, называемые *регистрами* (registers), которые похожи на ячейки основной памяти. Их можно разделить на регистры *общего назначения* (general-purpose registers) и *специальные регистры* (special-purpose registers). Обсуждение специальных регистров приводится в разделе 2.3; в этом разделе мы ограничимся изучением работы регистров общего назначения.

Регистры общего назначения используются для временного хранения данных, обрабатываемых в ЦП. В них сохраняются входные данные для схем арифметико-логического блока. Кроме того, 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 цепи в этом блоке, а также указать арифметико-логическому блоку тот регистр, в который должен быть помещен результат.

Полезно будет рассмотреть назначение регистров с точки зрения общих функций памяти компьютера. Регистры предназначены для хранения тех данных, с которыми машине необходимо работать непосредственно сейчас, основная память используется для хранения тех данных, которые понадобятся для работы в ближайшем будущем, а массовая память применяется для хранения данных, с которыми в ближайшее время вряд ли потребуется работать.

Во многих машинах к этой иерархической структуре присоединен дополнительный уровень, который называется сверхоперативной памятью (кэш). *Кэш* – это раздел высокоскоростной памяти с временем доступа, сравнимым со временем доступа к регистрам центрального процессора. Часто кэш непосредственно входит в состав ЦП. В эту специальную область памяти машина стремится скопировать именно ту часть основной памяти, в которой содержатся данные, необходимые для работы на данный момент. В этом случае обмен данными будет осуществляться не между регистрами и основной памятью, как это обычно бывает, а между регистрами и кэшем. Затем, в подходящий момент, все выполненные изменения одновременно передаются в основную память машины.

Интерфейс между ЦП и основной памятью. Для передачи битовых комбинаций между ЦП и основной памятью машины эти устройства соединяются группой проводов (рис. 2.1), которая называется *шиной* (bus). Именно через эту шину центральный процессор извлекает (или считывает) данные из основной памяти, направляя в нее адрес необходимой ячейки памяти вместе с сигналом считывания. Аналогичным образом ЦП помещает (или записывает) данные в память, указав адрес ячейки назначения и записываемую информацию, сопровождаемые сигналом записи.

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. 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 (рис. 2.2).

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

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

Машинные команды. Показанная на рис. 2.2 последовательность этапов представляет собой пример команд, которые должен уметь выполнять центральный процессор любой машины. Такие команды называются *машинными командами* (machine instruction). Полный список машинных команд относительно невелик. Одной из наиболее удивительных особенностей компьютерных наук является то, что 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 комфортность эксплуатации машины или скорость ее работы, однако никак не влияет на основные ее свойства. Подробнее об этом речь пойдет в главе 11.

При изучении системы машинных команд полезно будет разделить их на три категории: команды передачи данных, арифметические и логические команды, а также команды управления.

Команды передачи данных включают 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. Как и в случае с основной памятью, наиболее типичной является ситуация, когда перемещаемые данные сохраняются и в месте их исходного расположения. Процедура выполнения команд передачи данных больше напоминает копирование информации с одного места в другое, а не обычное их перемещение. Поэтому чаще всего употребляемые названия команд *пересылка* (transfer) или *перемещение* (move) следует считать выбранными неверно. Более подходящими названиями для этих команд можно считать *копирование* (copy) или *дублирование* (clone). Поскольку мы коснулись терминологии, то следует указать, что для передачи данных между ЦП и основной памятью существуют специальные термины. Запрос на заполнение регистра общего назначения содержимым ячейки памяти обычно называют командой *загрузки* (load), а запрос на передачу содержимого регистра в ячейку основной памяти – командой *сохранения* (store).

Вторую, очень важную группу команд этой категории составляют команды связи с устройствами, выходящими за рамки интерфейса ЦП – основная память. 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 2.6 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.

К *арифметическим и логическим командам* относятся те команды, которые указывают блоку управления на необходимость запросить 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 обычные логические операции AND, OR и XOR, которые уже рассматривались в главе 1. В этой главе мы обсудим эти операции более подробно.

В основном они используются для манипуляции отдельными битами некоторого регистра общего назначения; при этом состояние остальных регистров остается неизменным.

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 эти биты просто отбрасываются, а при операции вращения биты, покидающие пределы регистра с одного конца, помещаются во вновь вставляемые позиции на другом конце регистра. (Иногда последняя операция называется *циклическим сдвигом*.)

Команды управления 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 группа 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 в регистр число из основной памяти.
Этап 2. Загрузить в другой регистр еще одно число из основной памяти.
Этап 3. Если второе число равно нулю, перейти к этапу 6.
Этап 4. Разделить содержимое первого регистра на содержимое второго и записать результат в третий регистр.
Этап 5. Запомнить содержимое третьего регистра в основной памяти.
Этап 6. Завершить выполнение операции.

Рис. 2.3. Деление чисел, сохраняемых в основной памяти

Вопросы для самопроверки

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. Почему термин "перемещение" следует считать неправильным для обозначения операции перемещения данных из одного места в другое?
4. В тексте команда перехода 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) адресуемой команды позднее будет изменено, потребуется найти и исправить все команды перехода на эту команду. Предложите другой способ записи команд перехода, не содержащий явного указания имени адресуемой команды.
5. Как Вы считаете, команда "Если 0 равен 0, то перейдите к этапу 7" является условным или безусловным переходом? Поясните Ваш ответ.

2.2. МАШИННЫЙ ЯЗЫК

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. Подобную систему можно сравнить с музыкальной шкатулкой, которая всегда играет одну и ту же мелодию. Однако нам необходимо устройство, которое должно обладать гибкостью, не уступающей возможностям плейера компакт-дисков. Один из подходов к достижению необходимой гибкости в ранних электронных машинах состоял в том, что блок управления машиной конструировался с учетом возможности его перекоммутации. В этом случае в блок управления входила коммутационная панель, напоминающая 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, может быть закодирована 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. С тех пор эта *концепция хранимой программы* (stored-program concept) считается стандартным подходом к решению данной проблемы, применяемым и в настоящее время. Прежде эта идея считалась сложной, поскольку многие относили данные и программы к совершенно разным категориям. Это как раз тот случай, когда за деревьями не удавалось увидеть леса.

Представление машинных команд в виде битовых комбинаций. Для реализации концепции хранимой программы центральный процессор разрабатывается так, чтобы распознавать определенные битовые комбинации как представления конкретных команд. Весь набор выполняемых команд вместе с системой их кодирования называют *машинным языком* (machine language), поскольку он представляет собой средство передачи алгоритмов машине.

Кодированное представление машинной команды обычно состоит из двух частей: *поля кода операции* (op-code field – сокращение от operation code field) и *поля операндов* (operand field). Битовая комбинация, помещаемая в поле кода операции, определяет ту элементарную операцию (например, сохранения, сдвига, XOR или перехода), выполнение которой предусматривается данной командой. Битовые комбинации в поле операндов предоставляют более детальную информацию о той операции, которая задана в поле кода операции. Например, в случае команды сохранения 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.

Пример машинного языка. Теперь давайте посмотрим, как можно закодировать команды в типичной вычислительной машине. Машина, которая используется в нашем примере, подробно описана в приложении В и схематично показана на рис. 2.4. Она состоит из 16 регистров общего назначения и 256 ячеек основной памяти, каждая из которых имеет длину восемь битов. Для целей адресации присвоим регистрам номера от 0 до 15, а адреса ячеек основной памяти установим равными от 0 до 255, исключая те случаи, когда мы будем работать с этими числами, представленными в двоичной системе. В последнем случае битовые комбинации будем представлять в шестнадцатеричной системе счисления, поэтому регистры общего назначения будут иметь номера от 0 до F, а ячейки памяти – адреса от 00 до FF.

Весь язык машины, описанной в приложении В, состоит из 12 команд, каждая из которых представлена 16-битовым кодом, записанным четырьмя шестнадцатеричными цифрами (рис. 2.5). Код операции для

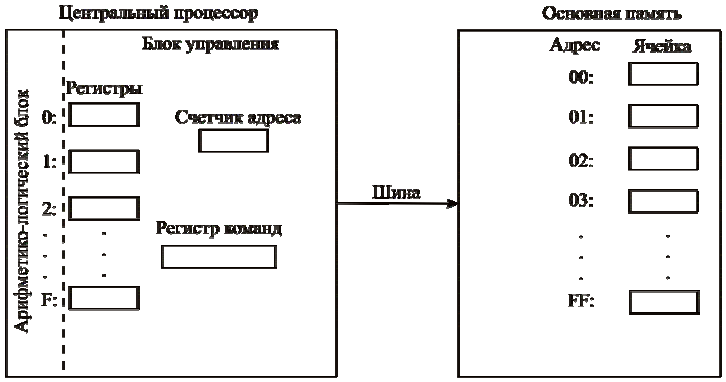


Рис. 2.4. Архитектура машины, описанной в приложении В

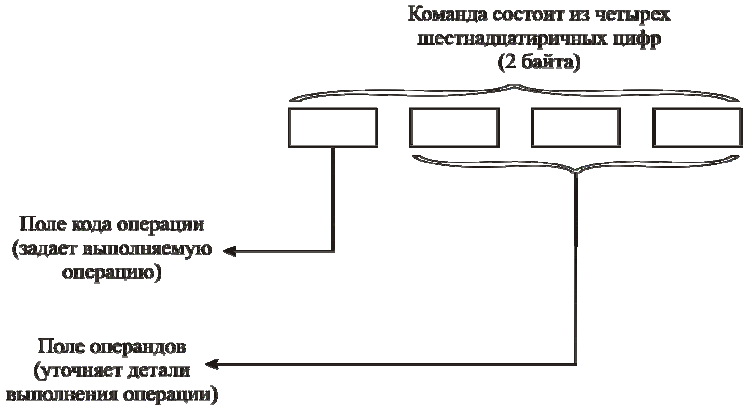


Рис. 2.5. Формат команды вычислительной машины, описанной в приложении В

каждой команды размещается в первых ее четырех битах и представляется одной шестнадцатеричной цифрой от 1 до C. В частности, как можно увидеть в таблице приложения, команда, начинающаяся с цифры 3, является командой сохранения, команда, начинающаяся с шестнадцатеричной цифры A, является командой циклического сдвига.

Поле операнда каждой команды состоит из трех шестнадцатеричных цифр (12 бит) и во всех случаях (кроме команды остановки, для которой не требуется никаких уточнений) содержит дополнительные сведения, необходимые для выполнения команды, заданной кодом операции. Следовательно, если первая шестнадцатеричная цифра команды равна 1 (код операции считывания ячейки памяти), то следующая шестнадцатеричная цифра команды указывает общий регистр, в который требуется загрузить считанное из основной памяти значение, а последние две шестнадцатеричные цифры задают адрес ячейки памяти, из которой требуется считать данные. Например, команда 1347 (шестнадцатеричное число) воспринимается машиной как "Загрузить в регистр 3 содержимое ячейки памяти с адресом 47". Если код операции представлен шестнадцатеричной цифрой 7 (операция OR над содержимым двух регистров общего назначения), то следующая шестнадцатеричная цифра указывает номер регистра, в который следует поместить результат операции, а две последние шестнадцатеричные цифры поля операндов задают номера тех регистров, над содержимым которых необходимо выполнить операцию OR. В результате команда 70C5 понимается как инструкция "Выполнить операцию OR с содержимым регистров C и 5, а результат поместить в регистр 0".

Существует тонкое отличие между двумя командами загрузки. Код операции 1 (шестнадцатеричный) относится к команде загрузки регистра общего назначения содержимым ячейки основной памяти, тогда как код операции 2 (шестнадцатеричный) – к команде загрузки регистра общего назначения указанным числовым значением. Различие заключается в том, что поле операндов в команде первого типа содержит адрес, тогда как в команде второго типа поле операндов содержит ту битовую комбинацию, которую требуется загрузить в регистр.

В машине есть две команды сложения: 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.

Интересное решение принято в отношении команды перехода (код операции – шестнадцатеричная цифра B). Первая шестнадцатеричная цифра поля операндов указывает, какой регистр общего назначения следует сравнить с регистром 0. Если указанный регистр содержит ту же битовую комбинацию, что и регистр 0, то машина выполняет переход посредством выбора следующей команды по тому адресу, который указан в двух последних шестнадцатеричных цифрах поля операндов. В противном случае программа продолжает нормальную последовательность выполнения команд. По сути, это пример команды условного перехода. Однако если первая шестнадцатеричная цифра поля операндов будет равна 0, то данная команда запрашивает сравнить регистр 0 с регистром 0. Так как содержимое любого регистра всегда равно самому себе, то в этом случае переход всегда выполняется. Следовательно, команда, код которой начинается с шестнадцатеричных цифр B0, воспринимается как команда безусловного перехода.

Закончим этот раздел примером закодированной последовательности команд, приведенной на рис. 2.2. Предположим, что суммируемые числа представлены в дополнительном двоичном коде и хранятся в ячейках памяти с адресами 6C и 6D, а

сумму этих чисел необходимо поместить в ячейку памяти с адресом 6E.

Этап 1	156C
Этап 2	166D
Этап 3	5056
Этап 4	306E
Этап 5	C000

Вопросы для самопроверки

1. Запишите программу, приведенную как пример в конце данного раздела, в виде собственно битовых комбинаций.

2. Ниже представлены команды, записанные на машинном языке, которые описаны в приложении В. Приведите текстовую формулировку этих команд.

а) 368A; б) BADE; в) 803C; г) 40F4.

3. В чем состоит различие между командами 15AB и 25AB, записанными на машинном языке, представленном в приложении В?

4. Ниже дано текстовое представление нескольких машинных команд. Запишите эти команды на машинном языке, представленном в приложении В.

а) Загрузить в регистр 3 шестнадцатеричное число 56.

б) Сдвинуть содержимое регистра 5 на три бита вправо.

в) Передать управление команде, расположенной по адресу F3, если содержимое регистра 7 равно содержимому регистра 0.

г) Выполнить операцию AND над содержимым регистров A и 5 и поместить ее результат в регистр 0.

2.3. ВЫПОЛНЕНИЕ ПРОГРАММЫ

Машинный цикл. Компьютер выполняет хранимую в его памяти программу посредством копирования команд из основной памяти в блок управления (по мере необходимости). Как только команда попадает в блок управления, она декодируется, после чего выполняется. Порядок, в котором команды выбираются из памяти, соответствует порядку их размещения в памяти, за исключением случаев выполнения команды перехода. Чтобы представить себе общий процесс выполнения команд, необходимо познакомиться с тем, как блок управления функционирует внутри процессора. Этот блок включает два специализированных регистра: *счетчик адреса* (program counter) и *регистр команд* (instruction register) (см. рис. 2.4). Счетчик адреса содержит адрес следующей выполняемой команды, т.е. он предназначен для наблюдения за ходом выполнения программы. Регистр команды используется для хранения кода выполняемой команды.

Блок управления работает в режиме постоянного повторения алгоритма, называемого *машинным циклом* (machine cycle), который состоит из трех этапов: выборки, декодирования и выполнения (рис. 2.6). На этапе выборки блок управления извлекает из основной памяти ту команду, которая должна выполняться следующей. Блок управления знает, где именно в памяти находится требуемая команда, поскольку ее адрес содержится в счетчике адреса. Блок управления помещает считанную команду в регистр команд, а затем увеличивает значение в счетчике адреса так, чтобы он содержал адрес следующей команды.

Когда команда поступает в регистр команды, блок управления декодирует ее. Эта процедура включает и разбиение поля операндов на соответствующие составляющие части, исходя из кода операции данной команды.

Рис. 2.6. Схема машинного цикла

Затем блок управления выполняет команду посредством активизации соответствующей схемы, предназначенной для выполнения поставленной задачи. Например, если команда представляет собой операцию загрузки данных из основной памяти, блок управления выполняет загрузку требуемых данных; если же команда предусматривает выполнение арифметической операции, то блок управления активизирует соответствующую схему в арифметико-логическом блоке, указывая в качестве входных регистры, заданные в команде.

Когда обработка команды будет завершена, блок управления вновь начинает выполнение алгоритма машинного цикла с первого этапа. 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 команда перехода. Давайте рассмотрим выполнение команды с кодом B258, которая имеет следующий смысл: "Выполнить переход к команде, сохраняемой по адресу 58, если содержимое регистра 2 идентично содержимому регистра 0". В этом случае этап выполнения в машинном цикле начинается со сравнения содержимого регистров 2 и 0. Если оно различно, то этап выполнения этой команды завершается и начинается выполнение нового машинного цикла.

Если же содержимое указанных регистров одинаково, то машина на этом этапе поместит в счетчик адреса значение 58. Теперь на этапе выборки следующего машинного цикла блок управления обнаружит в счетчике адреса значение 58, поэтому следующей будет выполнена команда, расположенная по этому адресу.

Для координации действий, выполняемых на протяжении машинного цикла, необходимо обеспечить синхронизацию работы различных схем машины. С этой целью на соответствующие электронные схемы подается импульсный сигнал, который называется сигналом синхронизации. Амплитуда этого сигнала изменяется между уровнями 0 и 1, а различные электронные схемы машины разрабатываются таким образом, чтобы они приводились в действие тем или иным фронтом импульса синхронизирующего сигнала. В результате тактовая частота этот сигнала фактически определяет ту скорость, с которой центральный процессор выполняет свой машинный цикл.

Пример выполнения программы. Давайте проследим за выполнением всех машинных циклов, предусматриваемых программой, текст которой представлен в конце раздела 2.2. Вначале необходимо разместить программу в какой-либо области памяти машины. Например, будем считать, что текст нашей программы занесен в последовательные ячейки памяти, начиная с адреса A0 (шестнадцатеричное представление). На рис. 2.7 представлена таблица, отражающая содержимое этой области памяти. Если записать программу таким способом, можно заставить машину выполнить нашу программу, поместив адрес ее первой команды (A0) в счетчик адреса и запустив машину в работу.

Адрес	Содержимое
A0	15
A1	6C
A2	16
A3	6D
A4	50
A5	56
A6	30
A7	6E
A8	C0
A9	00

Рис. 2.7. Программа сложения чисел, записанная в памяти машины, начиная с адреса A0

Блок управления начинает с извлечения из основной памяти команды, записанной по адресу A0, и помещает ее (156C) в регистр команды, выполнив тем самым этап выборки первого машинного цикла. Обратите внимание, что длина команды составляет шестнадцать разрядов (2 байта). Поэтому выбираемая команда должна занимать две ячейки памяти с адресами A0 и A1. Конструкция блока управления разработана с учетом этой особенности, поэтому он выбирает содержимое обеих ячеек и помещает данные в регистр команды, размер которого составляет 16 бит. Затем блок управления добавляет число 2 к значению в счетчике адреса, чтобы содержимое этого регистра представляло собой адрес следующей команды. По завершении этапа выборки первого машинного цикла в счетчике адреса и регистре команд будут содержаться следующие данные.

Счетчик адреса: A2
Регистр команд: 156C

На следующем этапе блок управления анализирует команду, помещенную в регистр команд, и приходит к заключению, что это команда загрузки в регистр 5 содержимого ячейки памяти с адресом 6C. Загрузка осуществляется на этапе выполнения данного машинного цикла, после чего блок управления начнет новый машинный цикл.

Новый цикл начинается с выборки команды 166D из ячеек памяти с адресами A2 и A3. Блок управления помещает эту команду в регистр команд и увеличивает значение счетчика адреса, после чего оно становится равным A4. После завершения очередного этапа выборки в счетчике адреса и регистре команд будут следующие данные.

Счетчик адреса: A4
Регистр команд: 166D

Блок управления декодирует команду 166D и определяет, что в регистр 6 необходимо загрузить содержимое ячейки памяти с адресом 6D, после чего выполняет команду, и в регистр 6 действительно загружается требуемое значение.

Поскольку в данный момент счетчик адреса имеет значение A4, блок управления считывает следующую команду, которая начинается с указанного адреса. В результате в регистр команд помещается значение 5056, а счетчик адреса получает новое значение A6. Блок управления декодирует содержимое очередной команды и выполняет ее посредством активизации электронной схемы сложения чисел в дополнительном двоичном коде, указав этой схеме использовать как входные регистры 5 и 6.

На этапе выполнения данной команды арифметико-логический блок выполняет сложение и записывает результат в регистр 0 (как было указано блоком управления), после чего сообщает блоку управления о том, что требуемые действия выполнены. После этого блок управления начинает следующий машинный цикл. И вновь, используя текущее значение в счетчике адреса, он выбирает следующую команду из двух смежных ячеек памяти с адресами A6 и A7 (теперь это команда 306E) и модифицирует значение счетчика адреса, установив его равным A8. Затем считанная из памяти команда декодируется и выполняется. В результате сумма двух чисел помещается в ячейку памяти с адресом 6E.

Следующая выбираемая команда расположена в памяти, начиная с адреса A8. После ее извлечения значение счетчика адреса увеличивается до AA. Содержимое регистра команд равно C000 и расшифровывается как команда останова. В результате работа машины останавливается на этапе выполнения, и это означает, что выполнение нашей 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. Обычно человек отмечает в списке выполненные им инструкции галочками – машина же для этого использует счетчик адреса. Определив, какая из инструкций должна выполняться следующей, человек знакомится с ней и принимает решение, что именно нужно сделать. Затем он выполняет требуемые действия и переходит к следующей инструкции. Аналогичным образом поступает и машина – она выполняет очередную команду, помещенную в регистре команд, и запускает следующий цикл выборки.

Особенности совместного хранения данных и программ. В основной памяти компьютера можно разместить множество программ одновременно, выделив для каждой различные области памяти. Тем, какая из этих программ начнет выполняться при запуске машины, можно легко управлять, просто соответствующим образом установив 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, что – программой. Если в счетчике

адреса вместо адреса требуемой программы будет установлен адрес данных, то компьютер не сможет предпринять никаких иных действий, кроме как считать битовые комбинации данных так, как если бы они были командами, и попытаться выполнить их. Полученный результат непредсказуем и будет зависеть от того, с какими именно данными работала машина.

Тем не менее, нельзя сказать, что мы поступаем неверно, придавая и программам, и данным одинаковую форму. Благодаря этому 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.

Вопросы для самопроверки

1. Предположим, что в памяти машины, описанной в приложении В, в ячейках с адресами от 00 до 05 содержатся битовые комбинации, приведенные ниже (шестнадцатеричное представление).

Адрес	Содержимое
00	14
01	02
02	34
03	17
04	C0
05	00

Если запустить машину, предварительно установив в счетчике адреса значение 00, то какая битовая комбинация окажется в ячейке памяти с адресом 17 (шестнадцатеричное представление), когда работа машины будет остановлена?

2. Предположим, что в памяти машины, описанной в приложении В, в ячейках с адресами от B0 до B8 содержатся битовые комбинации, приведенные ниже (шестнадцатеричное представление).

Адрес	Содержимое
B0	13
Bl	B8
B2	A3
B3	02
B4	33
B5	B8
B6	C0
B7	00
B8	0F

а) Если в начале работы в счетчик адреса помещается значение B0, то какая битовая комбинация будет содержаться в регистре 3 после выполнения первой команды?

б) Какая битовая комбинация будет находиться в ячейке памяти с адресом B8 после выполнения команды останова?

3. Предположим, что в памяти машины, описанной в приложении В, в ячейках с адресами от A4 до B1 содержатся битовые комбинации, приведенные ниже (шестнадцатеричное представление).

Адрес	Содержимое
A4	20
A5	00
A6	21
A7	03
A8	22
A9	01
AA	B1
AB	B0
AC	50
AD	02
AE	B0
AF	AA
B0	C0
B1	00

Отвечая на следующие вопросы, исходите из того, что в начале работы счетчик адреса содержит значение A4.

а) Какое значение будет находиться в регистре 0 после первого выполнения команды, расположенной в ячейке с адресом AA?

б) Что будет находиться в регистре 0 после второго выполнения команды, расположенной в ячейке с адресом AA?

в) Сколько раз должна быть выполнена команда, расположенная в ячейке с адресом AA, прежде чем машина остановится?

4. Предположим, что в памяти машины, описанной в приложении В, в ячейках с адресами от F0 до F9 содержатся битовые комбинации, приведенные ниже (шестнадцатеричное представление).

Адрес	Содержимое
F0	20
F1	C0

F2	30
F3	F8
F4	20
F5	00
F6	30
F7	F9
F8	FF
F9	FF

Если в начале работы машины счетчик адреса содержит значение F0, то какими будут действия машины, если она дойдет до команды, записанной в ячейке с адресом F8?

2.4. АРИФМЕТИЧЕСКИЕ И ЛОГИЧЕСКИЕ КОМАНДЫ

Как мы уже говорили ранее, группа арифметических и логических команд состоит из таких команд, которые требуют выполнения некоторых арифметических операций, логических 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. они 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.

$$\begin{array}{r} 10011010 \\ \text{AND}\ \underline{11001001} \\ 10001000 \end{array}$$

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.

$$\begin{array}{r} 10011010 \\ \text{OR}\ \underline{11001001} \\ 11011011 \end{array} \qquad \begin{array}{r} 10011010 \\ \text{XOR}\ \underline{11001001} \\ 01010011 \end{array}$$

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, можем 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.

$$\begin{array}{r} 00001111 \\ \text{AND}\ \underline{10101010} \\ 00001010 \end{array}$$

Подобное применение операции AND является примером процедуры, называемой *маскированием* (masking). Первый операнд, или маска (mask), определяет, какая часть другого операнда подвергается изменению. В случае операции AND результатом маскирования будет частичная копия второго операнда, в которой нули заполняют те позиции, которые не подлежат дублированию.

Такая операция полезна при работе с *битовыми отображениями* (bit map), т.е. строкой битов, в которой каждый бит представляет наличие или отсутствие определенного объекта. Мы уже встречались с битовыми отображениями при изучении растрового представления изображений, в котором каждый бит ассоциируется с отдельным пикселем. Другим 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, тогда как 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, где 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 те 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.

$$\begin{array}{r} \text{OR}\ 11110000 \\ \underline{10101010} \\ 11111010 \end{array}$$

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. Например, обратите внимание на взаимосвязь между вторым операндом и строкой результата.

$$\begin{array}{r} \text{XOR}\ 11110000 \\ \underline{10101010} \\ 01010101 \end{array}$$

Операция XOR между любым байтом и байтом, содержащим все 1, дает в результате байт дополнения к исходному байту.

Операции сдвига. Операции сдвига и вращения (циклического сдвига) позволяют перемещать биты в регистре и часто используются для решения проблем выравнивания, например при подготовке значения байта к последующим операциям маскирования или манипулирования значением мантиссы в представлениях с плавающей точкой. Классификация этих операций производится по направлению движения (вправо или влево), а также с учетом того, является сдвиг циклическим или нет. 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 или влево. На том конце байта, в направлении которого происходит сдвиг, крайний бит выйдет за его пределы и будет потерян, на другом конце образуется пустое место, в которое потребуется ввести некоторое значение. То, что произойдет с удаляемым битом и что будет вставлено в освободившуюся позицию – именно это и определяет отличия 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 также иногда называют вращением. Если выполнить циклический сдвиг байта вправо восемь раз подряд, то получим ту же битовую комбинацию, которая существовала вначале, тогда как семь циклических сдвигов вправо идентичны одному циклическому сдвигу влево.

Другим вариантом решения является удаление бита, выходящего за пределы байта, и помещение в освобождающиеся позиции исключительно значения 0. Подобный вариант называют *логическим сдвигом* (logical shift). 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 два. 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 освобождающаяся 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* (arithmetic shift).

Арифметические операции. Несмотря на то что выше уже шла речь об арифметических операциях сложения, вычитания, умножения и деления, все же необходимо сделать ряд замечаний. Как уже говорилось, все операции этой группы чаще всего могут быть реализованы с помощью единственной операции сложения и действия отрицания. Поэтому некоторые малогабаритные компьютеры содержат в своем наборе команд только операции сложения или 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 гипотетической машины, описанной в приложении В. При операциях сложения операнды могут быть представлены в двоичном дополнительном коде, и тогда операция их сложения будет выполняться как обычное двоичное суммирование. Если же операнды будут представлены как числа в формате с плавающей точкой, то при суммировании потребуется выделить мантиссу каждого из чисел. После этого эти значения должны быть сдвинуты вправо или влево в зависимости от значения в поле порядка. Затем проверяются знаковые биты и выполняется операция сложения. Полученный результат вновь переводится в формат с плавающей точкой. Как видите, хотя обе описанные выше операции считаются операциями сложения, действия машины по их выполнению будут существенно отличаться. Более того, с точки зрения самой машины, между этими двумя операциями вообще нет никакой связи.

Вопросы для самопроверки

1. Выполните приведенные ниже операции:

а) $\text{AND}\begin{array}{r} 010011011 \\ \underline{101001011} \end{array}$; б) $\text{AND}\begin{array}{r} 10000011 \\ \underline{11101100} \end{array}$; в) $\text{AND}\begin{array}{r} 11111111 \\ \underline{00101101} \end{array}$;

г) $\text{OR}\begin{array}{r} 010011011 \\ \underline{101001011} \end{array}$; д) $\text{OR}\begin{array}{r} 10000011 \\ \underline{11101100} \end{array}$; е) $\text{OR}\begin{array}{r} 11111111 \\ \underline{00101101} \end{array}$;

ж) XOR $\begin{matrix}010011011\\101001011\end{matrix}$; з) XOR $\begin{matrix}10000011\\11101100\end{matrix}$; и) XOR $\begin{matrix}11111111\\00101101\end{matrix}$.

2. Предположим, что требуется выделить средние три бита в семиразрядной строке посредством помещения нулей в оставшиеся ее четыре бита, не изменяя значения трех выделяемых битов. Какую маску и операцию следует использовать в этом случае?

3. Предположим, что нужно заменить значение средних трех битов в семиразрядной строке на обратное, не изменяя при этом значения оставшихся четырех битов. Какую маску и 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 цифровой. 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?

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:

а) AB; б) 5C; в) B7; г) 35.

9. На сколько разрядов влево нужно сдвинуть восьмиразрядную строку, 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 считать, что эти значения представляют числа в формате с плавающей точкой, речь о котором шла в главе 1?

11. Используя машинный язык, описываемый в приложении В, напишите программу, которая поместит единицу в старший бит ячейки памяти с адресом A7, оставив значения всех остальных битов этой ячейки без изменения.

12. Используя машинный язык, описываемый в приложении В, напишите программу, которая скопирует значения четырех средних битов ячейки памяти с адресом E0 в младшие четыре бита ячейки памяти с адресом E1, при этом в старшие четыре бита этой ячейки должны быть занесены нули.

2.5. ВЗАИМОДЕЙСТВИЕ ЦП С ПЕРИФЕРИЙНЫМИ УСТРОЙСТВАМИ*

Основная память и центральный процессор образуют центральное звено компьютера. Ниже мы рассмотрим, как это центральное звено взаимодействует с различными периферийными устройствами, такими, как дисковые накопители, принтеры, а также другими компьютерами.

Взаимодействие через контроллеры. Взаимодействие между машиной и другими устройствами обычно осуществляется через промежуточное устройство, называемое *контроллером* (controller). Если в качестве примера взять персональный компьютер, то контроллер будет представлять собой ту монтажную плату, которая вставляется в разъем на основной *монтажной плате компьютера* (motherboard – материнской плате). С помощью кабелей платы контроллеров соединяются с периферийными устройствами, установленными в самом компьютере, а соединение с внешними устройствами осуществляется через промежуточные разъемы, установленные на задней стенке корпуса компьютера.

Каждый контроллер обеспечивает взаимодействие с определенным видом устройства. Некоторые из них разработаны для взаимодействия с монитором, другие отвечают за взаимодействие с дисководами, а есть контроллеры, поддерживающие взаимодействие компьютера с устройствами чтения компакт-дисков. Поэтому иногда вместе с новым периферийным устройством приходится покупать и новый контроллер. 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. Подобные контроллеры часто представляют собой небольшие специализированные компьютеры с собственной основной памятью и центральным процессором, который выполняет программу, управляющую всеми действиями данного контроллера.

Когда контроллер вставляется в один из разъемов на материнской плате компьютера, он электрически подключается к шине, соединяющей ЦП компьютера и его основную память (рис. 2.8). В месте своего подключения каждый контроллер осуществляет непрерывное наблюдение за сигналами, посылаемыми из ЦП машины, и отвечает на те, которые адресованы непосредственно ему.

В частности, центральный процессор способен взаимодействовать с контроллерами, подключенными к шине, так же, как он взаимодействует с оперативной памятью. Для того чтобы послать цепочку битов контроллеру, прежде всего ее нужно поместить в один из регистров общего назначения, после чего выполнить команду, подобную команде сохранения, чтобы "сохранить" код в контроллере. Точно так же, для того чтобы получить цепочку битов от контроллера, исполняется команда, похожая на команду загрузки. В некоторых компьютерах предусмотрены дополнительные коды операции для этих действий. Команды с такими кодами называются командами ввода-вывода. Команды ввода-вывода находят контроллер с помощью системы адресации, подобной системе адресации оперативной памяти. А именно, каждому контроллеру соответствует уникальный набор адресов (адреса ввода–вывода), которые используются в командах ввода–вывода для указания контроллера-адресата.

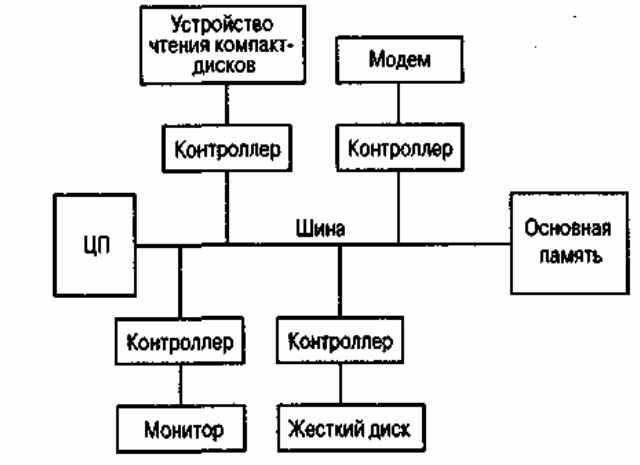


Рис. 2.8. Подключение контроллеров к шине компьютера

Набор адресов, соответствующих контроллеру, называется *портом* (port), так как они представляют собой "место", через которое информация входит в компьютер и выходит из него. Поскольку адреса ввода–вывода могут иметь такой же вид, как адреса ячеек оперативной памяти, шины компьютеров снабжены сигналом, который показывает, передается сообщение в оперативную память или в контроллер. Следовательно, на команду ввода–вывода отослать содержимое регистра определенному контроллеру центральный процессор будет реагировать так же, как на команду отослать цепочку битов в определенную ячейку памяти, только при этом он выставит сигнал, который сообщит устройствам, подключенным к шине, что цепочка битов предназначена для такого-то контроллера, а не для оперативной памяти.

Альтернативой включению в машинный язык специальных кодов операций для команд ввода–вывода является использование команд загрузки и сохранения, которые уже существуют в языке для коммуникации с оперативной памятью. В этом случае контроллер отвечает только на определенный уникальный набор адресов (который также называется портом), а оперативная память игнорирует эти ячейки. Таким образом, когда центральный процессор посылает сообщение шине о том, что нужно сохранить цепочку битов по адресу, приписанному к контроллеру, ее и получает контроллер, а не оперативная память. Точно так же, если центральный процессор пытается прочитать данные из такого адреса, указанного в команде загрузки, то он получит последовательность битов из контроллера, а не из памяти. Такая система связи называется *отображением ввода/вывода в память* (memory-mapped I/O), потому что устройства ввода–вывода компьютера представляются как различные ячейки памяти (рис. 2.9).

Поскольку контроллер подключен к шине компьютера, он может сам связываться с оперативной памятью в течение тех наносекунд, когда центральный процессор не использует шину. Подобный тип доступа контроллера к основной памяти называется *прямым доступом к памяти* (DMA – direct memory access) и является важным средством повышения производительности компьютера. Если в компьютере контроллер дисковых устройств обладает прямым доступом к памяти, то ЦП может посылать ему представленные в виде битовых комбинаций запросы, требующие считать с диска определенный сектор и поместить прочитанные данные в указанный блок ячеек основной памяти.

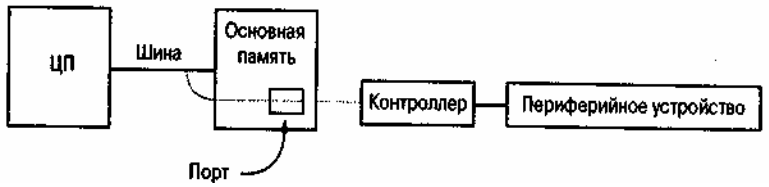


Рис. 2.9. Концептуальная схема отображения ввода–вывода в память

Такой блок ячеек называется *буфером*. В общем случае буфер представляет такое местоположение, где одна система может оставить данные, к которым позднее сможет получить доступ другая система. Пока контроллер будет выполнять затребованную операцию считывания данных, ЦП может продолжать обработку других заданий. Это означает, что в одно и то же время будут выполняться два разных действия. ЦП будет выполнять программу, а контроллер в это время 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.

Наконец, следует заметить, что 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 оно способно посылать данные обратно в компьютер. В самом деле, компьютер способен подготавливать символы и передавать их принтеру намного быстрее, чем принтер сможет их печатать. Если компьютер будет передавать данные на принтер вслепую, то принтер быстро начнет отставать, что может привести к потере данных. Поэтому такой процесс, как печать документа, предусматривает постоянный двусторонний диалог, в процессе которого компьютер и периферийное устройство обмениваются информацией о текущем состоянии устройства.

Такой диалог часто предусматривает использование *слова состояния* (status word) устройства, т.е. определенной битовой комбинации, которая генерируется периферийным устройством и посылается его контроллеру. Биты в слове состояния отражают текущее состояние устройства. Если вернуться к примеру с принтером, то значение младшего бита слова состояния может указывать, что в принтере закончилась бумага, тогда как следующий бит в этом слове указывает, готов ли принтер к приему очередной порции информации. В зависимости от выбранной системы контроллер либо сам реагирует на подобную информацию о состоянии устройства, либо передает ее на обработку в ЦП. В каждом случае либо программа в контроллере, либо программа, выполняемая центральным процессором, должна быть разработана так, чтобы задержать пересылку данных на принтер до тех пор, пока от него не будет получена соответствующая информация о состоянии.

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). В каждом случае максимальная скорость передачи данных зависит от типа используемой линии связи и способа передачи.

Существуют два основных способа передачи данных: параллельный и последовательный. Этими терминами обозначают способ передачи битов относительно друг друга. В случае *параллельной связи* (parallel communication) несколько битов передаются одновременно, каждый по отдельному проводнику (линии). Такая техника позволяет быстро передавать данные, но требует достаточно сложной линии связи.

В качестве примера можно привести внутреннюю шину компьютера и большинство каналов связи между компьютером и периферийными устройствами, такими как запоминающие устройства и принтеры.

В этих случаях скорость передачи данных измеряется в Мбит/с и выше.

Напротив, при *последовательной связи* (serial communication) за один раз передается только один бит. Такая техника передачи данных медленнее, но для нее требуется более простой канал связи, поскольку все биты передаются по одной линии, один за другим. Последовательная связь обычно используется для передачи информации между компьютерами, где более простой канал связи является более экономным.

Например, существующие телефонные линии являются системами последовательной связи, так как они передают тоны один за другим.

В процессе связи между компьютерами по этим линиям последовательности битов сначала с помощью *модема* (сокращение от "модулятор-демодулятор") преобразуются в слышимые звуки, затем эти звуки последовательно передаются по телефонной сети и снова трансформируются в цепочки битов модемом, находящимся в месте назначения.

На практике представление цепочек битов тонами разных частот (называемое частотной модуляцией) используется только для низкоскоростной связи, не более 1200 бит/с. Чтобы добиться скорости 2400 бит/с, 9600 бит/с и выше, модем комбинирует изменения частоты тона, его амплитуды и фазы (степени задержки передачи сигнала).

А для достижения еще более высоких скоростей передачи данных часто применяются способы сжатия данных, что позволяет получить скорость передачи до 57,6 кбит/с.

Эти скорости передачи данных являются пределом для современных телефонных линий с частотным диапазоном 3 Гц. Однако они не удовлетворяют современным запросам. Передача графических изображений со скоростью 57,6 кбит/с может стать невыносимо долгой, а передавать видеоизображение с такой скоростью вообще неразумно. Поэтому развиваются новые технологии, которые могли бы дать пользователям, использующим телефонные линии, более высокие скорости передачи данных. Одна из таких технологий – *цифровая абонентская линия* (digital subscriber line – DSL). Она использует тот факт, что существующие телефонные линии способны пропускать более широкий диапазон частот, чем тот, что применяется для передачи речевых сигналов. Скорость передачи в таких системах обычно составляет около 1,5 Мбит/с, но может достигать 6 Мбит/с в одном направлении, если в это время передача данных в противоположном направлении ограничена. Значение скорости зависит от используемой версии DSL и длины линии до операционного центра телефонной компании, которая обычно не превышает трех миль. В других технологиях, составляющих конкуренцию DLS, применяется кабель, который используется в системах кабельного телевидения, при этом скорость передачи данных достигает 40 Мбит/с. Также применяется оптическое стекловолокно, скорость передачи которого может составлять несколько гигабит в секунду.

Вопросы для самопроверки

1. Предположим, машина, описанная в приложении В, использует механизм отображения ввода/вывода в память, а адрес B5 определяет местоположение порта принтера, в который должны передаваться данные для вывода на печать.

а) Если регистр 7 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 рассказа, занимающего 300 страниц печатного текста, представленного символами в коде ASCII, если передача данных будет осуществляться со скоростью 57 600 бит/с?

2.6. ДРУГИЕ ТИПЫ АРХИТЕКТУРЫ КОМПЬЮТЕРОВ*

Для расширения кругозора целесообразно рассмотреть некоторые варианты архитектуры построения компьютеров, отличающиеся от той архитектуры, речь о которой шла в предыдущих разделах этой главы.

CISC- и RISC-архитектура компьютеров. Разработка машинного языка требует предварительного принятия многих решений. Одно из них состоит в выборе между построением сложной машины, способной декодировать и выполнять широкий спектр разнообразных команд, и созданием более простой машины, которая будет иметь ограниченный набор команд. К первому варианту относятся компьютеры с CISC-архитектурой (Complex Instruction Set Computer – компьютер со сложным набором команд), а ко второму – компьютеры с RISC-архитектурой (Reduced Instruction Set Computer – компьютер с ограниченным набором команд). CISC-компьютер проще программировать, поскольку единственная его команда позволяет решить задачу, выполнение которой в RISC-компьютере потребует длинной последовательности более простых команд. Однако CISC-компьютер сложнее сконструировать, и он обходится дороже как при создании, так и при эксплуатации. Более того, многие сложные команды найдут лишь ограниченное применение, вследствие чего могут оказаться просто балластом, создающим дополнительную бесполезную нагрузку.

Для уменьшения числа используемых электронных цепей CISC-процессоры обычно конструируются по двухуровневой схеме, при которой каждая машинная команда в действительности выполняется как последовательность простейших операций. В такие процессоры обычно встроен блок специальных ячеек памяти, называемых памятью микрокоманд, в которых записана программа, именуемая микропрограммой. Именно эта микропрограмма управляет выполнением сложных команд машинного языка. Кроме того, в микропрограмму можно вносить коррективы и тем самым изменять смысл команд машинного языка. Таким образом, помимо поддержки расширенного набора команд CISC-процессоров, реализуемой без использования сложных электронных схем, наличие микропрограммы позволяет настраивать один и тот же тип центрального процессора на поддержку различных специализированных команд машинного языка просто посредством внесения необходимых изменений в микропрограмму.

Сторонники RISC-архитектуры заявляют, что все эти преимущества не оправдывают издержки, связанные с поддержкой микропрограммирования. Они утверждают, что лучше спроектировать более простую машину с небольшим, но тщательно продуманным набором команд. Такой подход позволяет избежать усложнения конструкции, связанного с поддержкой памяти микропрограмм, что приводит к упрощению проектирования центрального процессора. Однако это означает, что программы на машинном языке для RISC-компьютеров будут значительно длиннее программ для CISC-компьютеров, так как для выполнения сложных операций потребуется несколько отдельных машинных команд, тогда как в CISC-архитектуре тот же результат достигается с помощью единственной команды.

В настоящее время на рынке представлены оба вида процессоров – как с CISC-, так и с RISC-архитектурой. Процессоры серии Pentium, выпускаемые компанией Intel, являются примером центральных процессоров с CISC-архитектурой, тогда как процессоры серии PowerPC, выпускаемые компаниями Apple Computer, IBM и Motorola, имеют RISC-архитектуру.

Конвейерная обработка. Скорость прохождения электронных импульсов по проводам не превышает скорости света. Поскольку скорость света составляет около 30 см в наносекунду (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. Несмотря на фантастический прогресс в этой области, все же рано или поздно будет достигнут теоретический предел.

Попытки решить эту проблему привели к тому, что конструкторы вычислительных машин заменили исходную концепцию скорости выполнения команд принципом пропускной способности. Этот термин означает общее количество работы, которое машина способна выполнить за определенный период времени, при этом продолжительность выполнения отдельного задания в расчет не принимается.

Приведем пример того, как можно повысить пропускную способность компьютера без увеличения скорости выполнения команд. В данном случае используется подход, называемый конвейерной обработкой, согласно которому 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, 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. Современные процессоры 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 машин.

Аргументом в пользу создания многопроцессорных машин может стать не что иное, как модель работы человеческого мозга. Современные технологии уже позволяют создавать электронные схемы, в которых есть столько же переключающих цепей, сколько нейронов в мозге человека (нейроны можно рассматривать как живые переключающие схемы). Несмотря на это, возможности современных компьютеров все еще значительно уступают возможностям человеческого мозга. Считается, что это происходит из-за неэффективного использования компонентов машин, вызванного недостатками архитектуры компьютеров. Действительно, если в компьютере установлено множество схем памяти и всего один центральный процессор, то большинство схем памяти в любой момент времени просто не используется. В противоположность этому, большая часть человеческого мозга в любой момент времени пребывает в активном состоянии. По этой причине сторонники параллельной обработки выступают в защиту машин, имеющих несколько устройств обработки данных. Они заявляют, что такое решение

способствует созданию конфигурации с более высокой степенью использования элементов.

По этому принципу было разработано большое количество машин. 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 послать 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 позволит 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 больше 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 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 подзадачи параллельно. В результате вся задача в целом может быть выполнена в многопроцессорной машине намного быстрее, чем в однопроцессорной.

При разработке и использовании многопроцессорных машин мы сталкиваемся с проблемой баланса нагрузки, т.е. динамического распределения задач между различными процессорами в целях повышения эффективности их работы. Эта проблема тесно связана с проблемой масштабирования, или разделения текущей задачи на несколько подзадач, количество которых совместимо с количеством доступных процессоров. Еще одна проблема заключается в сложности распределения выделенных задач. Действительно, если количество задач возрастает, то объем работы, связанной с их распределением и координацией взаимодействия между отдельными подзадачами, растет экспоненциально. Если имеется четыре задачи, то можно выделить шесть потенциальных пар задач, которым потребуется взаимодействовать друг с другом. Если имеется пять задач, то количество потенциальных каналов взаимодействия возрастает до десяти, а в случае с шестью задачами это количество увеличится до пятнадцати.

В главе 10 мы познакомимся с искусственными нейронными сетями, конструкция которых основана на наших знаниях о структуре человеческого мозга. Подобные конструкции представляют собой еще один тип многопроцессорной архитектуры, так как они состоят из множества элементарных процессоров или устройств обработки информации, выходные данные которых – это просто реакция такого устройства на поступившие входные данные. Все эти простые процессоры соединены между собой и образуют сеть, в которой выходные данные одних процессоров являются входными данными для других процессоров. Такая машина программируется посредством настройки степени влияния выходных данных каждого процессора на реакцию соединенных с ним процессоров. В какой-то степени этот подход имитирует способ, в соответствии с которым, как считается, происходит обучение нашего мозга. Точнее говоря, биологические сети нейронов головного мозга человека учатся реагировать определенным образом на заданные стимулы посредством управления химическим составом соединений (синапсов) между отдельными нейронами, что, в свою очередь, контролирует способность одного нейрона влиять на действия других нейронов.

Вопросы для самопроверки

1. Почему для работы машины с микропрограммным управлением требуются два счетчика адреса и два регистра команд?

2. 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 AA? При каких условиях конвейерная обработки на данном этапе программы не будет давать никаких преимуществ?

3. Какие противоречия должны быть разрешены при запуске программы, приведенной в вопросе 4 из раздела 2.3, в машине с конвейерной обработкой данных?

4. 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, что содержимое данной ячейки останется 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.

Упражнения

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

1. Дайте краткое определение следующим понятиям:

а) регистр; б) кэш-память; в) основная память; г) массовая память.

2. Предположим, что в ячейках памяти машины, описанной в приложении В, записан блок данных с адресами от B9 и до C1 включительно. Сколько ячеек памяти содержит этот блок? Перечислите их адреса.

3. Каким будет значение в счетчике адреса машины, описанной в приложении В, непосредственно после выполнения команды с кодом B0BA?

4. Предположим, что ячейки памяти с адресами от 00 до 05 в машине, описанной в приложении В, содержат следующие битовые комбинации.

Адрес	Содержимое
00	21
01	04
02	31
03	00
04	C0
05	00

Исходя из предположения, что исходно счетчик адреса содержал значение 00, запишите содержимое счетчика адреса, регистра команд и ячейки памяти по адресу 00 в конце фазы выборки каждого машинного цикла до тех пор, пока машина не остановится.

5. 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?

6. Ниже приведено несколько команд на машинном языке, описанном в приложении В. Дайте текстовое описание этих команд.

а) 407E; б) 9028; в) A302; г) B3AD; д) 2835.

7. Предположим, что в некотором машинном языке поле кода операции имеет длину четыре бита. Сколько различных типов команд может существовать в этом языке? Что можно сказать по этому поводу, если длина поля кода операции будет увеличена до 8 бит?

8. Запишите приведенные ниже команды на машинном языке, описанном в приложении В:

а) Загрузить в регистр 8 содержимое ячейки памяти с адресом 55.

б) Загрузить в регистр 8 шестнадцатеричное число 55.

в) Выполнить циклический сдвиг содержимого регистра 4 на три бит вправо.

г) Выполнить операцию AND над содержимым регистров F и 2, поместив результат операции в регистр 0.

д) Выполнить переход к команде, расположенной по адресу 31, если содержимое регистра 0 будет равно содержимому регистра В.

9. Распределите на три категории приведенные ниже команды (записанные на машинном языке, описанном в приложении В), исходя из того, изменит ли выполнение команды содержимое ячейки памяти с адресом 3B, позволит ли выполнение команды считать содержимое ячейки памяти с адресом 3B или же данная команда никак не зависит от содержимого ячейки памяти с адресом 3B: а) 153B; б) 253B; в) 353B; г) 3B3B; д) 403B.

10. Предположим, что в машине, описанной в приложении В, ячейки памяти с адресами от 00 до 03 содержат следующие битовые комбинации.

Адрес	Содержимое
00	23
01	02
02	C0
03	00

а) Дайте текстовую формулировку первой команды.

б) Если в начале работы машины содержимое счетчика адреса будет равно 00, какая битовая комбинация окажется в регистре 3, когда машина выполнит команду останова?

11. Предположим, что в машине, описанной в приложении В, ячейки памяти с адресами от 00 до 05 содержат следующие битовые комбинации.

Адрес	Содержимое
00	10
01	04
02	30
03	45
04	C0
05	00

Дайте ответ на поставленные ниже вопросы, полагая, что машина начинает работу со счетчиком адреса, равным 00.

а) Сформулируйте текстовое описание каждой команды.

б) Какая битовая комбинация будет находиться в ячейке памяти с адресом 45, после того как машина выполнит команду останова?

в) Какая битовая комбинация будет находиться в счетчике адреса, когда машина выполнит команду останова?

12. Предположим, что в машине, описанной в приложении В, ячейки памяти с адресами от 00 до 09 содержат следующие битовые комбинации.

Адрес	Содержимое
00	1A
01	02

02	2B
03	02
04	9C
05	AB
06	3C
07	00
08	C0
09	00

Будем считать, что машина начинает работу со счетчиком адреса, равным 00.

а) Какое значение будет находиться в ячейке памяти с адресом 00, когда машина выполнит команду останова?

б) Какая битовая комбинация будет находиться в счетчике адреса, когда машина выполнит команду останова?

13. Предположим, что в машине, описанной в приложении В, ячейки памяти с адресами от 00 до 0D содержат следующие битовые комбинации.

Адрес	Содержимое
00	20
01	03
02	21
03	01
04	40
05	12
06	51
07	12
08	B1
09	0C
0A	B0
0B	06
0C	C0
0D	00

Будем считать, что машина начинает работу со счетчиком адреса, равным 00.

а) Какая битовая комбинация будет находиться в регистре 1, когда машина выполнит команду останова?

б) Какая битовая комбинация будет находиться в регистре 0, когда машина выполнит команду останова?

в) Какая битовая комбинация будет находиться в счетчике адреса, когда машина выполнит команду останова?

14. Предположим, что в машине, описанной в приложении В, ячейки памяти с адресами от F0 до FD содержат следующие битовые комбинации (шестнадцатеричные).

Адрес	Содержимое
F0	20
F1	00
F2	21
F3	01
F4	23
F5	05
F6	83
F7	FC
F8	50
F9	01
FA	B0
FB	F6
FC	C0
FD	00

Если машина начнет работу со счетчиком адреса, имеющим значение F0, какое значение будет находиться в регистре 0, когда машина выполнит команду останова, расположенную в ячейке с адресом FC?

15. Если машина, описанная в приложении В, выполняет каждую команду за одну микросекунду (миллионная доля секунды), сколько времени займет выполнение программы, приведенной в задаче 14?

16. Предположим, что в машине, описанной в приложении В, в ячейках памяти с адресами от 00 до 05 содержатся следующие битовые комбинации (шестнадцатеричные).

Адрес	Содержимое
00	25
01	B0
02	35
03	04
04	C0
05	00

Когда машина выполнит команду останова, если она начнет работу при содержимом счетчика адреса 00?

17. Для каждого приведенного ниже задания напишите небольшую программу на машинном языке, описанном в приложении В, предназначенную для его выполнения. Исходите из того, что каждая программа будет записана в памяти, начиная с адреса 00.

а) Переместите значение, сохраняемое в ячейке памяти с адресом 8D, в ячейку с адресом В3.

б) Поменяйте местами значения, записанные в ячейках с адресами 8D и В3.

в) Если значение, записанное в ячейке памяти с адресом 45, равно 00, запишите значение СС в ячейку памяти с адресом 88, в противном случае запишите значение DD в ячейку памяти с адресом 88.

18. Одной из популярных среди любителей компьютеров игр является *core wars* (война сердечников). (Термин *core* использовался в устаревших технологиях создания основной памяти, где нули и единицы представлялись магнитными полями в небольших кольцевых сердечниках из магнитного материала.) Игра проводится между двумя программами, выступающими друг против друга, 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 (описанном в приложении В), реализующую стратегию обороны, т.е. имеющую минимально возможный размер.

б) Напишите игровую программу на машинном языке, описанном в приложении В, стратегия которой будет построена на стремлении избежать любых нападений другой программы посредством перемещения самой себя по различным адресам основной памяти. Точнее говоря, напишите программу, начинающуюся с адреса 00, которая при выполнении должна копировать себя в ячейки памяти, начиная с адреса 70, а затем передавать управление этой новой копии.

в) Модернизируйте программу из пункта б), чтобы она продолжала свое перемещение по новым адресам. В частности, пусть программа переместит себя по адресу 70, затем по адресу ЕО (70 + 70), а затем по адресу 60 (70 + 70 + 70) и т.д.

19. Напишите программу на машинном языке, описанном в приложении В, которая будет вычислять сумму чисел, представленных в двоичном дополнительном коде и сохраняемых в ячейках с адресами А1, А2, А3, А4. Результат должен быть записан в ячейку с адресом А5.

20. Предположим, что в машине, описанной в приложении В, ячейки памяти с адресами от 00 до 05 содержат следующие битовые комбинации (шестнадцатеричное представление).

Адрес	Содержимое
00	20
01	C0
02	30
03	04
04	00
05	00

Что произойдет, если машина начнет работу со счетчиком адреса, имеющим значение 00?

21. Что произойдет, если в машине, описанной в приложении В, ячейки памяти с адресами 06 и 07 будут содержать битовые комбинации В0 и 06 и машина начнет работу со счетчиком адреса, имеющим значение 06?

22. Предположим, что приведенная ниже программа написана на машинном языке, описанном в приложении В, и записана в память машины, начиная с ячейки с адресом 30 (шестнадцатеричное представление). Какое задание будет выполнено программой после ее завершения?

```
2003
2101
2200
2310
1400
3410
5221
5331
3239
333B
B248
B038
C000
```

23. Опишите этапы выполнения машиной, описанной в приложении В, команды с кодом операции В. Представьте свой ответ в виде набора инструкций, указывающих центральному процессору, что и как нужно делать.

24. Опишите этапы выполнения машиной, описанной в приложении В, команды с кодом операции 5. Представьте свой ответ в виде набора инструкций, указывающих центральному процессору, что и как нужно делать.

25. Опишите этапы выполнения машиной, описанной в приложении В, команды с кодом операции 6. Представьте свой ответ в виде набора инструкций, указывающих центральному процессору, что и как нужно делать.

26. Предположим, что регистры 4 и 5 в машине, описанной в приложении В, содержат битовые комбинации ЗС и С8, соответственно. Какая комбинация окажется в регистре 0 после выполнения следующих команд:

а) 5045; б) 6045; в) 7045; г) 8045; д) 9045.

27. Используя машинный язык, описанный в приложении В, напишите программы для выполнения приведенных ниже заданий.

а) Скопируйте битовую комбинацию, записанную в ячейке памяти с адресом 66, в ячейку с адресом BB.

б) Присвойте четырем младшим битам ячейки памяти с адресом 34 значение 0, не изменяя при этом значения остальных ее битов.

в) Скопируйте четыре младших бита из ячейки памяти с адресом A5 в четыре младших бита ячейки с адресом A6, не изменяя при этом значения остальных битов ячейки с адресом A6.

г) Скопируйте четыре младших бита из ячейки памяти с адресом A5 в четыре старших бита этой же ячейки. (В результате первые четыре бита в ячейке с адресом A5 будут идентичны ее последним четырем битам.)

28. Выполните указанные операции:

а) $\underset{101001}{\overset{111000}{\text{AND}}}$; б) $\underset{101010}{\overset{000100}{\text{AND}}}$; в) $\underset{010101}{\overset{000100}{\text{AND}}}$; г) $\underset{000100}{\overset{110101}{\text{AND}}}$;

д) $\underset{101001}{\overset{111000}{\text{OR}}}$; е) $\underset{101010}{\overset{000100}{\text{OR}}}$; ж) $\underset{010101}{\overset{000100}{\text{OR}}}$; з) $\underset{110101}{\overset{111011}{\text{OR}}}$;

и) $\underset{101001}{\overset{111000}{\text{XOR}}}$; к) $\underset{101010}{\overset{000100}{\text{XOR}}}$; л) $\underset{010101}{\overset{000100}{\text{XOR}}}$; м) $\underset{110101}{\overset{111011}{\text{XOR}}}$.

29. Укажите значение маски и тип логической операции, необходимые для выполнения указанных ниже действий.

а) Поместите значение 0 в четыре средних бита восьмибитовой комбинации, не изменяя состояния других ее битов.

б) Получите двоичное дополнение для комбинации из восьми битов.

в) Определите двоичное дополнение для старшего бита восьмибитовой ячейки памяти без изменения состояния остальных ее битов.

г) Поместите значение 1 в старший 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.

30. 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.

31. Укажите последовательность логических операций (вместе с соответствующими масками), которые при применении ее 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.

32. 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.

33. Каким будет результат выполнения циклического сдвига на один бит вправо для следующих байтов, представленных в шестнадцатеричной системе? Ответы также должны быть представлены в шестнадцатеричной системе: а) 3F; б) 0D; в) FF; г) 77.

34. Напишите программу на машинном языке, описанном в приложении В, которая изменит содержимое ячейки памяти с адресом 8C на обратное.

35*. Успеет ли принтер, печатающий 40 знаков в секунду, вывести строку из символов кода ASCII (каждый имеет бит четности), последовательно поступающих со скоростью 300 бит/с? А если скорость передачи будет 1200 бит/с?

36*. Предположим, что человек вводит с клавиатуры по 30 слов в минуту, причем каждое слово состоит из пяти символов. Если считать, что машина каждую микросекунду (миллионная доля секунды) выполняет одну команду, то сколько команд выполнит эта машина, пока с клавиатуры будут введены два символа?

37*. Сколько битов в секунду должна передавать клавиатура в компьютер, чтобы успевать за пользователем, выполняющим ввод со скоростью 30 слов в минуту? (Предположим, что каждый символ шифруется в коде ASCII и имеет бит четности, а каждое слово состоит из пяти символов.)

38*. Какую скорость передачи информации, выраженную в битах в секунду, будет иметь система связи, способная передавать любую последовательность из восьми различных состояний с максимальной скоростью 300 состояний в секунду?

39*. Предположим, что машина, описанная в приложении В, для работы с принтером использует метод отображения ввода/вывода в память. Также предположим, что адрес FF используется для передачи символов в принтер, а адрес FE – для получения информации о состоянии принтера. В частности, будем считать, что младший бит ячейки с адресом FE 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 00, которая будет ожидать, пока принтер сообщит о готовности принять очередной символ, а затем отправит ему символ, представленный битовой комбинацией в регистре 5.

40*. Напишите программу на машинном языке, описанном в приложении В, которая будет помещать значения 00 во все ячейки основной памяти с адресами от A0 до C0 и одновременно будет достаточно небольшого размера, чтобы помещаться в ячейках памяти с адресами от 00 до 13 (шестнадцатеричными).

41*. Предположим, что машина, имеющая на жестком диске
500 Мбайт свободной памяти, принимает данные по телефонной линии связи со скоростью 14 400 бит/с. Сколько времени ей понадобится, чтобы заполнить данными все свободное место на жестком диске?

42*. Предположим, что линия связи используется для последовательной передачи данных со скоростью 14 400 бит/с. Сколько битов данных будет искажено, если возникнет радиопомеха общей длительностью 0,01 секунды?

43*. Предположим, что имеются 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?

44*. Кратко опишите основные отличия между CISC- и RISC-архитектурами.

45*. Кратко опишите отличия между основной памятью и памятью микропрограмм.

46*. Опишите два подхода к увеличению пропускной способности машины.

47*. Объясните, как среднее значение для некоторого набора чисел может быть вычислено быстрее в многопроцессорной машине, чем в машине, имеющей только один процессор.

Ответы на вопросы для самопроверки

Раздел 2.1

1. На малых машинах это действие часто представляет собой процесс, состоящий из следующих двух этапов: сначала содержимое первой ячейки копируется в регистр, а затем записывается в предназначенную для него ячейку памяти. На большинстве больших машин этот процесс выполняется за один этап.

2. Значение, которое должно быть записано; адрес ячейки, в которую должно быть записано значение и команда записи.

3. Термин *переместить* (move) часто 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.

4. 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 такие команды потребуется изменять, если между исходной командой перехода и командой, на которую следует перейти, позднее будут вставлены дополнительные команды.

5. 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 всегда 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 ...", то такую команду можно использовать как для условного, так и безусловного перехода.

Раздел 2.2

1. 156C = 0001010101101100
166D = 0001011001101101
5056 = 0101000001010110
306E = 0011000001101110

2. а) Сохранить содержимое регистра 6 в ячейке памяти с адресом 8A.

б) Перейти на команду в ячейке с адресом DE, если содержимое регистра A равно содержимому регистра 0.

в) Выполнить поразрядную операцию AND над содержимым регистров 3 и C, поместив результат в регистр 0.

г) Переместить содержимое регистра F в регистр 4.

3. Команда 15AB требует, чтобы центральный процессор запросил у схемы управления основной памятью содержимое ячейки с адресом AB. Извлеченное из памяти значение помещается в регистр 5. Команда 25AB не предусматривает такого запроса к памяти. Точнее говоря, в регистр 5 просто помещается значение AB.

4. а) 2356; б) A503; в) B7F3; г) 80A5.

Раздел 2.3

1. Шестнадцатеричное 34.

2. а) 0F; б) C3.

3. а) 00; б) 01; 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 шестнадцатеричное число C0 в ячейку памяти с адресом F8, а следующие две команды – значение 00 в ячейку с адресом F9. Таким образом, в то время, когда машина выберет команду из ячейки с адресом F8, там уже будет храниться команда прекращения работы C000.

Раздел 2.4

1. а) 00001011; б) 10000000; в) 00101101; г) 11101011; д) 11101111; е) 11111111; ж) 11100000; з) 01101111; и) 11010010.

2. 0011100 с операцией AND.

3. 0011100 с операцией 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, операция XOR может рассматриваться как операция 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) 57; б) B8; в) 6F; д) 6A.

9. 5.

10. В двоичном дополнительном коде – 00110110; в виде числа с плавающей точкой – 01011110. Дело в том, что процедуры сложения двух чисел отличаются в зависимости от интерпретации заданных двоичных кодов.

11. Одно из решений будет следующим:

12A7 (загрузка в регистр 2 содержимого ячейки с адресом A7)

2380 (загрузка в регистр 3 значения 80)

7023 (операция OR над содержимым регистров 2 и 3 с помещением результата в регистр 0)

30A7 (запись содержимого регистра 0 в ячейку памяти с адресом A7)

C000 (прекращение выполнения программы)

12. Одно из решений будет таковым:

15E0 (загрузка в регистр 5 содержимого ячейки памяти с адресом E0)

A502 (циклический сдвиг содержимого регистра 5 на два бита вправо)

260F (загрузка в регистр 6 значения OF)

8056 (поразрядная операция AND над содержимым регистров 5 и 6 с помещением результата в регистр 0)

30E1 (запись содержимого регистра 0 в ячейку памяти с адресом E1)

C000 (прекращение выполнения программы)

Раздел 2.5

1. а) 37B5.

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 посылать 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 Мбайт/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 ASCII, содержит около 1 Мбайт, или 8 000 000 бит информации. Таким образом, для передачи всего романа со скоростью 57 600 бит/с потребуется приблизительно 139 секунд (или $2^1/_3$ минуты).

Раздел 2.6

1. Один набор регистров используется для извлечения, декодирования и выполнения микрокоманд, тогда как другой применяется для извлечения, декодирования и выполнения команд машинного языка в соответствии с микропрограммой.

2. Конвейер может содержать команды B1B0 (выполняемая в данный момент), 5002 и, возможно, даже B0AA. Если значение в регистре 0 равно значению в регистре 1, выполняется переход к ячейке с адресом B0 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.

3. Если не предпринять меры предосторожности, информация из ячеек с адресами F8 и F9 будет извлечена в качестве команды еще до того, как предыдущая часть программы получит возможность изменить содержимое этих ячеек.

4. 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.

3. ОПЕРАЦИОННЫЕ СИСТЕМЫ И СЕТИ

Современные прикладные задачи часто требуют от машины выполнения действий, которые могут конкурировать друг с другом за ресурсы машины. Например, компьютер может быть соединен с несколькими терминалами или рабочими станциями, с которых разные пользователи одновременно запрашивают выполнение каких-либо действий. Даже в вычислительных системах индивидуального пользования, таких, как персональный компьютер, пользователь может потребовать выполнения не связанных друг с другом действий – воспроизведения музыки с компакт-диска во время редактирования документа или даже распечатывание другого документа. Для осуществления всех этих действий необходима высокая степень координации, чтобы быть уверенным, что несвязанные действия не мешают друг другу и что взаимодействие между связанными действиями эффективно и надежно. Эта координация выполняется пакетом программ, который называется *операционной системой* (operation system).

Похожие проблемы координации и взаимодействия возникают, когда разные компьютеры объединены в *сеть* (network или net). Решение этих проблем является естественным продолжением предмета операционных систем. В этой главе мы обсудим основные понятия, относящиеся к операционным системам и организации сетей.

3.1. ЭВОЛЮЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

Изучение операционных систем и сетей мы проведем в порядке их возникновения и развития: от ранних однопроцессорных к более поздним многопроцессорным системам.

Однопроцессорные системы. В 1940 – 1950 гг. однопроцессорные машины были недостаточно гибкими и эффективными. Выполнение программ требовало определенной предварительной подготовки оборудования: установки лент, загрузки перфокарт в устройство чтения перфокарт, установки переключателей и т.д. Запуск каждой программы, называемый *заданием* (job), производился по отдельности. Если нескольким пользователям требовалось 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. Первой целью создания таких систем было отделение пользователя от оборудования, что позволяло избавиться от постоянного потока людей, входящих и выходящих из помещения, в котором находилась машина. Дополнительно была введена должность оператора компьютера, задача которого заключалась 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* (bath processing) – метода выполнения заданий посредством предварительного объединения их в единый пакет, который затем выполнялся без дальнейшего взаимодействия с пользователем. Задания, ожидающие своего выполнения в массовой памяти, образуют *очередь заданий* (job queue) (рис. 3.1).

Очередь – это способ организации памяти, когда сохраняемые в ней объекты (в нашем случае – задания) упорядочены по принципу "*Первым вошел – первым вышел*" (FIFO, First-in, First-out), т.е. объекты покидают очередь в том же порядке, в котором они 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* (Job Control Language – JCL) и помещались вместе с заданием в очередь задач.

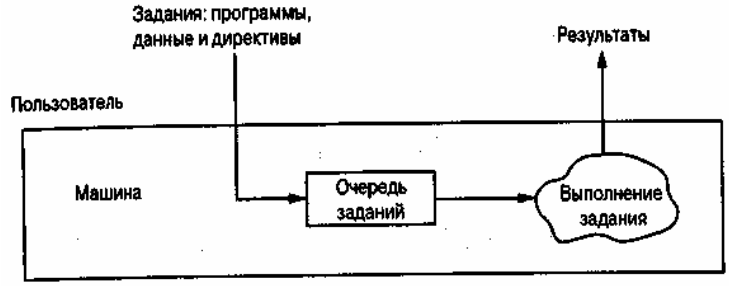


Рис. 3.1. Пакетная обработка

Когда задание выбиралось для выполнения, операционная система распечатывала эти инструкции на принтере, чтобы оператор мог их прочитать и выполнить. Инструкции, требовавшие вмешательства оператора, касались главным образом управления периферийным оборудованием. Поскольку в настоящее время эти действия сведены к минимуму, язык управления заданиями превратился, скорее, в язык общения с операционной системой, а не с оператором компьютера. И действительно, должность оператора компьютера стала практически ненужной.

Теперь организации нанимают для управления вычислительной системой системных администраторов. В их задачу, вместо управления вычислительными системами в прежнем понимании этого слова, входит получение нового оборудования и программного обеспечения, наблюдение за его установкой, осуществление локального руководства, включающего ведение

учетных записей пользователей, определение лимитов дискового пространства для отдельных пользователей и координацию усилий по решению возникающих в системе проблем.

Главным недостатком традиционной пакетной обработки является то, что пользователь лишен возможности взаимодействовать с программой с того момента, как она поставлена в очередь. 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 отказах 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 позволяют выполнять программы, ведущие диалог с пользователем, работающим за удаленным терминалом или рабочей станцией. Такой режим функционирования называется *интерактивной* или *диалоговой обработкой* (interactive processing) (рис. 3.2). Для работы интерактивных систем требуется, чтобы выполняемые машиной действия координировались с происходящими в ее среде событиями. Эта координация действий машины и среды именуется *обработкой в реальном времени* (real-time processing).

Если бы от интерактивных систем требовалось обслуживать в каждый момент времени только одного пользователя, то обработка в реальном времени не вызвала бы никаких проблем. Однако вычислительные машины стоили дорого, поэтому каждая машина должна была обслуживать несколько пользователей. Вполне типичной являлась ситуация, когда нескольким пользователям одновременно требовалось получить доступ к машине в интерактивном режиме, поэтому выполнение работы в реальном времени оказывалось затруднительным.

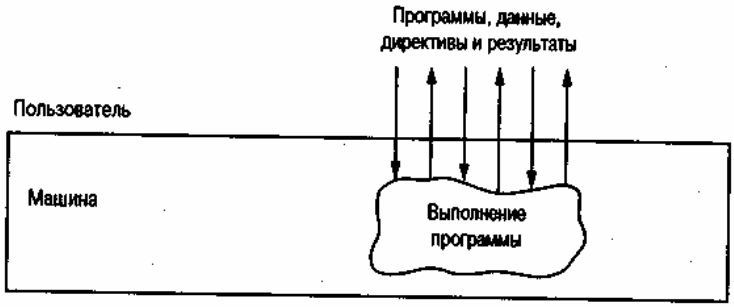


Рис. 3.2. Интерактивная обработка

Если в такой многопользовательской среде операционная система будет придерживать строго поочередного выполнения заданий, то только один из пользователей сможет получить удовлетворительное обслуживание в реальном времени.

Решение этой проблемы состоит в разработке операционной системы, способной организовать постоянное чередование выполнения частей различных заданий с помощью процесса *разделения времени* (time sharing). 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 текущее задание выгружается, за 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.

Сегодня разделение времени активно используется как в многопользовательских системах, так и в системах с одним пользователем, хотя раньше этот режим назывался *многозадачным* (multitasking), поскольку создавалась полная иллюзия односменного выполнения нескольких задач. Независимо от того, одно- или многопользовательской является данная вычислительная среда, было установлено, то применение режима разделения времени повышает эффективность работы машины. Конечно, вас это может удивить, особенно если принять во внимание, что процесс переключения заданий, обязательный при разделении времени, требует немалых дополнительных затрат времени. Действительно, время, потраченное на переключения между задачами, непродуктивно. Однако без использования такого режима разделения времени компьютер большую часть своего рабочего времени будет проводить в ожидании того, когда завершат работу периферийные устройства или пользователь введет свой следующий запрос. Режим разделения времени позволяет использовать это бесполезно теряемое время на решение других задач. В результате пока одно задание ожидает какое-либо событие, выполняется другое задание. В конечном итоге при использовании режима разделения времени вся группа одновременно запущенных заданий будет выполнена быстрее, чем в случае их последовательного выполнения.

Многопроцессорные системы. В последние годы необходимость в совместном использовании информации и ресурсов различными машинами породила потребность соединить компьютеры для обмена информацией. Для этого были созданы ставшие популярными объединенные компьютерные системы, называемые *сетями* (network или net). Сегодня концепция большой центральной машины, обслуживающей многих пользователей, в основном уступила место концепции множества маленьких машин, объединенных в сеть, в которой пользователи совместно используют ресурсы, рассредоточенные по всей системе, – устройства печати, программные пакеты, устройства памяти и информацию. Основной пример – это *Интернет* (Internet), глобальная сеть сетей, которая сегодня объединяет миллионы компьютеров во всем мире. Мы познакомимся с Internet подробнее в разделах 3.5 и 3.6.

Большинство проблем координации действий, возникающих при создании сетей, очень похожи на проблемы, с которыми пришлось столкнуться при разработке операционных систем. Фактически программное обеспечение для управления сетью может рассматриваться как сетевая операционная система. В этом свете разработка сетевого программного обеспечения является естественным расширением концепции операционной системы. Хотя первые сети создавались как свободно соединяемые между собой отдельные машины, каждая из которых являлась собственной операционной системой, дальнейшие исследования в области сетевой технологии привели к появлению сетевых систем, обеспечивающих совместное использование имеющихся ресурсов всеми выполняемыми в сети задачами. Различные ресурсы поочередно выделяются выполняемым

в сети задачам согласно их потребностям, невзирая на физическое местонахождение этих ресурсов. Примером может служить система серверов имен, существующая Internet, которую мы подробнее рассмотрим в разделе 3.5. Эта система позволяет множеству машин, разбросанных по всему миру, работать совместно, решая задачу перевода Internet-адресов из мнемонической формы, понятной человеку, в цифровую, понятную установленному в сети оборудованию.

Сети представляют собой только один из типов многопроцессорных проектных решений, используемых при разработке современных операционных систем. В то время как в сетях многопроцессорная система создается посредством объединения отдельных машин, каждая из которых имеет только один центральный процессор, другие многопроцессорные системы разрабатываются как одна машина с несколькими процессорами. Операционная система в такой машине должна не только координировать взаимодействие между различными видами деятельности, которые действительно выполняются одновременно, но и контролировать процесс распределения действий по отдельным процессорам в машине. В связи с этим возникают проблемы *баланса загрузки* (load balancing) (получение гарантий, что все процессоры в системе используются одинаково эффективно), а также *масштабирования* (scaling) (разбиение задач на количество подзадач, совместимое с числом процессоров в машине).

Мы видим, что появление многопроцессорных систем добавило много новых направлений исследований в области операционных систем. Эти исследования, несомненно, будут продолжаться и в будущем.

Вопросы для самопроверки

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 букв на экране монитора по мере их набора на клавиатуре.

г) Выполнение программы, предсказывающей состояние экономики в будущем году.

3. Каковы различия между обработкой в реальном времени и интерактивной обработкой?
4. Каковы отличия между режимом с разделением времени и многозадачностью?

3.2. АРХИТЕКТУРА ОПЕРАЦИОННЫХ СИСТЕМ

Для понимания архитектуры типичной операционной системы полезно представлять себе полный спектр программного обеспечения, используемого в стандартной вычислительной системе. Мы начнем обсуждение с обзора программного обеспечения, включающего общую схему 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 фирмы Microsoft имеется группа так называемых "Программ", содержащая 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.

Обзор программного обеспечения. Первым делом разделим программное обеспечение на две общие категории: *прикладное программное обеспечение* (application software) и *системное программное обеспечение* (system software) (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 создает фундамент, на котором ее граждане основывают свой индивидуальный стиль жизни.

Внутри класса системного программного обеспечения также есть две категории: одна – собственно операционная система, вторая – элементы программного обеспечения, объединяемые понятием *обслуживающие программы*, или *утилиты* (utility software).

Рис. 3.3. Классификация программного обеспечения

Большую часть установленных в системе обслуживающих программ составляют программы, предназначенные для выполне-

ния действий, необходимых для успешного функционирования компьютера, но еще не включенные в операционную систему. В некотором смысле обслуживающие программы объединяют элементы программного обеспечения, расширяющие возможности операционной системы. Например, обычно операционная система сама по себе не предоставляет средств форматирования диска или копирования файлов, поэтому данные функции обеспечиваются обслуживающими программами. К другим видам обслуживающих программ относятся программы для установки соединений по телефонным линиям с использованием модема, программы сжатия и распаковки данных, программное обеспечение для осуществления сетевых соединений.

Предоставление определенных функциональных возможностей с помощью обслуживающих программ существенно упрощает разработку операционной системы. Более того, выполнение рутинных операций, реализуемых с помощью утилит, легче настроить в соответствии с требованиями конкретной установки. Действительно, вовсе не являются исключением компании или независимые пользователи, модифицирующие или расширяющие возможности утилит, поставляемых вместе с операционной системой.

Различие между прикладным и обслуживающим программным обеспечением весьма условно. С нашей точки зрения, различие заключается в том, является ли пакет частью инфраструктуры программного обеспечения. Таким образом, новое приложение может превратиться в утилиту, если оно становится одной из основных сервисных программ. Различие между обслуживающими программами и операционной системой также условно. В некоторых системах такая основная функция, как ведение списка файлов в массовой памяти, представлена в виде обслуживающих программ, в то время как другие системы встраивают ее в операционную систему.

Компоненты операционной системы. Часть операционной системы, которая обеспечивает интерфейс операционной системы с пользователями, часто называют *оболочкой* (shell). Назначение оболочки – организация взаимодействия с пользователем (или пользователями) системы. Современные оболочки выполняют эту задачу с помощью *графического интерфейса пользователя* (graphical user interface – GUI), в котором объекты манипуляции, подобные файлам и программам, представлены на экране монитора в виде небольших рисунков – пиктограмм. Подобные системы позволяют пользователям вводить команды, указывая на эти пиктограммы и щелкая на них с помощью управляемого рукой приспособления, называемого мышью. Прежние оболочки поддерживали общение с пользователями посредством текстовых сообщений, вводимых с клавиатуры и отображаемых на экране монитора.

Хотя оболочка операционной системы играет важную роль в определении доступной на данной машине функциональности, она, тем не менее, является всего лишь интерфейсом между пользователем и сердцем самой операционной системы (рис. 3.4). Различие между оболочкой и внутренними частями операционной системы подчеркивается тем фактом, что некоторые операционные системы разрешают пользователю выбрать наиболее удобный для него тип оболочки. Например, пользователи операционной системы UNIX могут выбрать одну из оболочек, включая Borne, C или Korn. Ранние версии Windows также представляли собой всего лишь оболочки для операционной системы MS-DOS. Во всех этих случаях сама операционная система остается прежней – меняется лишь способ ее общения с пользователями.

Главным компонентом современных графических оболочек является *система управления окнами* или *оконный менеджер* (windows manager), который 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. В результате программа размещает предоставленный трафарет 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.

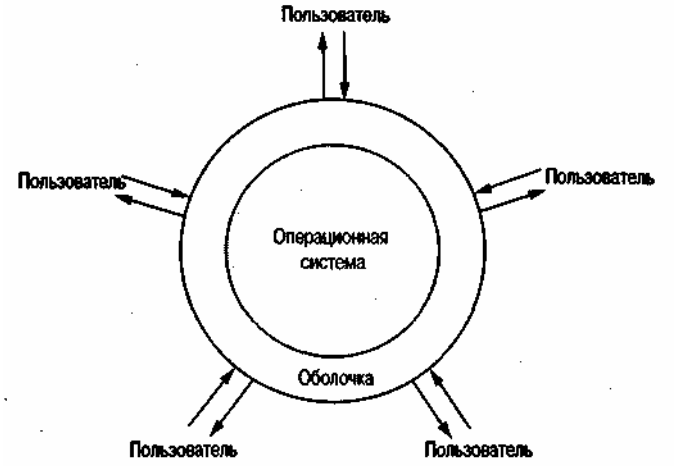


Рис. 3.4. Оболочка как интерфейс между пользователем и операционной системой

В отличие от оболочки операционной системы, ее внутренняя часть обычно называется *ядром* (kernel), которое включает компоненты программного обеспечения, выполняющие основные функции в процессе приведения компьютера в рабочее состояние. Одним из этих компонентов является *система управления файлами*, или просто *файловая система* (file manager), в задачу которой входит координация использования запоминающих устройств. Точнее говоря, эта подсистема 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.

Для удобства пользователей большинство подсистем управления файлами разрешает объединять файлы в группы, называемые *каталогами* (directory), или *папками* (folder). 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. Цепочка каталогов, ведущая к файлу, называется *путем* (path).

Любой доступ к файлу со стороны других компонентов программного обеспечения предоставляется и контролируется системой управления файлами. Процедура получения доступа к файлу начинается с запроса к файловой системе, который называется процедурой открытия файла. Если система управления файлами разрешает доступ, то она предоставляет информацию, необходимую для поиска файла и работы с ним. Эта информация записывается в область основной памяти, называемую *дескриптором файла* (file descriptor). Любые действия с файлом осуществляются посредством обращения к информации, содержащейся в дескрипторе файла.

Другой компонент ядра операционной системы представляет собой набор *драйверов устройств* (devices drivers), т.е. элементов программного обеспечения, взаимодействующих с контроллерами устройств (или же непосредственно с устройствами) в целях выполнения различных операций в периферийных устройствах машины. Каждый драйвер устройства специально разрабатывается для конкретного типа устройства (например, принтера, дисковода, накопителя на магнитных лентах или монитора). Он преобразует поступающие запросы в последовательность команд выполнения отдельных физических операций, которые требуется выполнить устройству, связанному с этим драйвером. В результате разработка других элементов программного обеспечения может вестись независимо от специфических особенностей конкретных устройств. Все это позволяет создать обобщенную операционную систему, которая будет настраиваться на использование определенных периферийных устройств с помощью простой установки соответствующих драйверов.

Еще один компонент ядра операционной системы – *система управления памятью* (memory manager), которая решает задачу координации использования машиной ее основной памяти. 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 многопользовательской среде или в среде со многими задачами, когда машина должна обрабатывать множество запросов, поступающих в одно и то же время, у подсистемы управления памятью обширные обязанности. В этой ситуации в основной памяти одновременно должно находиться множество программ и блоков данных, причем каждая из программ занимает собственную область памяти, отведенную ей программой управления памятью. По мере того как возникает необходимость в выполнении различных действий и после их окончания, подсистема управления памятью должна находить области памяти для удовлетворения возникающих потребностей в памяти, а также отслеживать информацию о тех участках памяти, которые уже освободились.

Задача подсистемы управления памятью еще больше усложняется, когда требуемый объем основной памяти превышает реально существующий 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 (virtual memory). Предположим, что выполняемым программам требуется 256 Мбайт основной памяти, а в наличии имеется только 128. Чтобы создать иллюзию большего объема памяти, программа управления памятью делит требуемый объем на элементы, называемые *страницами* (pages), и хранит содержимое этих страниц в массовой памяти. Типичный объем страницы – не больше 4 кбайт. Подсистема управления памятью помещает в основную память те страницы, которые в данный момент должны там находиться, замещая ими те, в которых больше нет потребности. Таким образом, остальные компоненты программного обеспечения могут работать так, как если бы объем основной памяти машины действительно составлял 256 Мбайт.

Кроме того, в состав ядра операционной системы входят *планировщик* (scheduler) и *диспетчер* (dispatcher), о которых речь 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 обсудили, как операционная система взаимодействует с пользователями и как компоненты операционной системы совместно осуществляют координацию действий внутри машины. Однако еще не было сказано ни слова о том, как же запускается сама операционная система. Запуск операционной системы осуществляется с помощью процесса, называемого *самозагрузкой* (booting), который выполняется при каждом включении машины. Первым шагом к пониманию этого процесса есть осознание того, почему его необходимо выполнять на первом этапе.

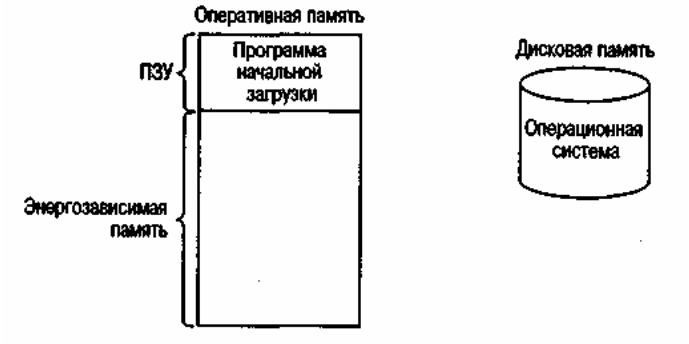
Центральный процессор машины (ЦП) разработан таким образом, что при его включении выполняемая им программа каждый раз стартует с определенного, наперед заданного адреса. Следовательно, именно в этом месте основной памяти ЦП ожидает найти первую команду, которую требуется выполнить. Чтобы гарантировать, что требуемая программа всегда будет присутствовать на указанном месте, этот участок памяти обычно конструируется так, чтобы его содержание было неизменным. Такая память носит название *постоянной памяти* или *постоянного запоминающего устройства* – ПЗУ (read-only memory – ROM). Когда код помещается в ПЗУ, он находится там постоянно, независимо от того, включена машина или выключена.

В маленьких компьютерах, используемых в качестве управляющих приборов в микроволновых печах, автомобильных системах зажигания и стереоприемниках, представляется удобным выделить значительный объем основной памяти под ПЗУ, так как гибкость в таких системах не нужна. При каждом включении выполняется одна и та же программа. Но в случае с универсальными компьютерами ситуация другая и в них не практикуется отведение большого объема основной памяти под постоянные программы. Содержимое памяти таких машин должно быть изменяемым. Фактически большая часть памяти универсальных компьютеров в настоящее время сконструирована так, что ее содержимое может не только изменяться, но и теряться при выключении машины. Такая память называется энергозависимой.

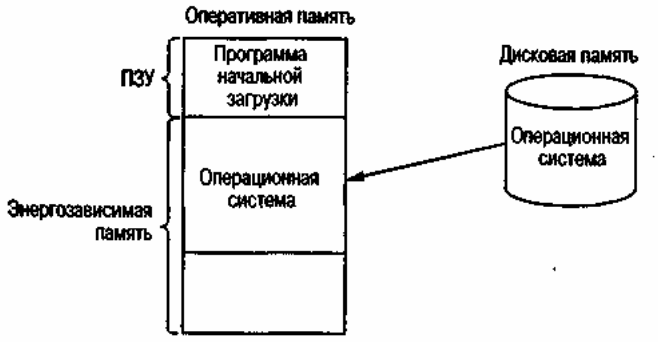
Поэтому для начальной загрузки в компьютерах общего назначения лишь малая часть основной памяти строится из микросхем ПЗУ. Эта область содержит ячейки памяти, в которых ЦП ожидает найти команды, выполняемые при включении машины. Небольшая программа, которая постоянно находится в этой области памяти, называется *программой первоначаль-*

ной загрузки (bootstrap). Эта программа выполняется автоматически при каждом включении компьютера. Она предписывает ЦП считать данные из заранее определенного участка массовой памяти в энергозависимую основную память (рис. 3.5). В большинстве случаев этими данными является программный код операционной системы. Как только программы операционной системы будут помещены в основную память, программа первоначальной загрузки потребует от ЦП выполнить команду перехода в данную область памяти. В результате стартуют программы ядра, и операционная система начинает контролировать дальнейшую деятельность машины.

В большинстве современных персональных компьютеров программа первоначальной загрузки разработана так, что прежде всего она пытается отыскать операционную систему на гибком диске (дискете).



а)



б)

Рис. 3.5. Процесс первоначальной загрузки:
а – Этап 1: Машина начинает выполнять программу начальной загрузки, находящуюся в памяти. Операционная система находится в массовой (внешней) памяти; *б* – Этап 2: Программа начальной загрузки выдает указание поместить операционную систему в оперативную память, а затем передает ей управление

Если дискета в машину не вставлена, программа загрузки автоматически приступает к считыванию операционной системы с жесткого диска. Однако если дискета вставлена в устройство, но не содержит копии операционной системы, программа загрузки приостановится и выдаст сообщение об ошибке оператору. Вы, вероятно, сталкивались с этим, когда включали персональный компьютер, забыв предварительно вынуть несистемную дискету из дисковода.

Вопросы для самопроверки

1. Перечислите компоненты типичной операционной системы и охарактеризуйте роль каждого из них одной фразой.
2. В чем заключается различие между прикладными программным обеспечением и обслуживающими программами?
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 понятия *процесса*.

Понятие процесса. Одной из наиболее фундаментальных концепций в современных операционных системах является разграничение между самой программой и деятельностью, связанной с ее выполнением. Первое представляет собой статический набор инструкций, в то время как второе – это динамическая деятельность, свойства которой меняются во времени. Эта деятельность и получила название *процесса* (process). Процесс охватывает текущее состояние работы, называемое *состоянием процесса* (process state). 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).

Чтобы подчеркнуть различие между программой и процессом, заметим, что одна программа может быть связана одновременно с несколькими процессами. Например, в многопользовательской системе с разделением времени двум пользователям может одновременно потребоваться редактировать различные документы. Оба могут использовать одну и ту же программу редактирования, но в каждом случае это будет самостоятельный процесс, со своим набором данных и относительной

скоростью выполнения. В такой ситуации операционная система может хранить в основной памяти только одну копию программы редактирования и разрешить каждому из процессов пользоваться ею на время выделенного ему кванта времени.

В типичной компьютерной установке с разделением времени в состязании за кванты времени обычно принимает участие множество процессов. Эти процессы включают выполнение прикладных программ, утилит и программных элементов операционной системы. Задача операционной системы состоит в координации выполнения всех этих процессов. Координация подразумевает получение гарантий в том, что каждый процесс получит все необходимые ему ресурсы (доступ к периферийным устройствам, место в основной памяти, доступ к данным и центральному процессору); что независимые процессы не влияют друг на друга, а процессы, которым необходимо обмениваться информацией, имеют возможность делать это. Взаимодействие между процессами называется *межпроцессным взаимодействием*.

Управление процессами. Задачи, связанные с координацией 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. Для отслеживания состояния всех процессов планировщик организует в основной памяти блок информации, называемый *таблицей процессов* (process table). 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, находится процесс в состоянии готовности или ожидания. Процесс находится в состоянии *готовности* (ready), если его развитие может продолжаться, и переводится в состояние *ожидания* (waiting), когда его развитие приостанавливается до тех пор, пока не произойдут некоторые внешние события, например, завершится процедура доступа к диску или поступит сообщение от другого процесса.

Диспетчер – это компонент ядра, отвечающий за то, чтобы запланированные процессы действительно выполнялись. В системе с разделением времени эта задача решается посредством разбиения времени процессора на короткие интервалы, называемые *квантами* (обычно продолжительностью около 50 миллисекунд). 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* (process switch).

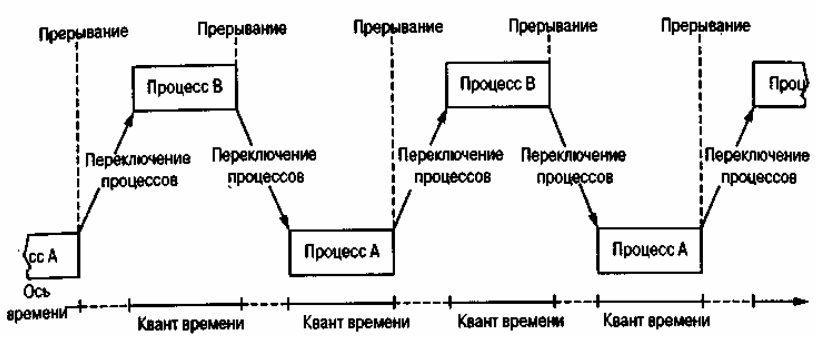


Рис. 3.6. Разделение времени между процессами А и В

Каждый раз, когда процессу предоставляется очередной квант времени, диспетчер инициирует цепь таймера, подготавливая его к измерению продолжительности следующего кванта. По окончании установленного кванта цепь таймера генерирует сигнал, называемый *прерыванием* (interrupt). 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 и обращает внимание на то, что его отвлекло. При получении сигнала прерывания центральный процессор завершает текущий машинный цикл, сохраняет свое положение в текущем процессе (подробнее мы обсудим это чуть ниже) и начинает выполнять программу, называемую *обработчиком прерываний* (interrupt handler), помещенную в заранее определенное место в основной памяти.

В нашем сценарии разделения времени обработчик прерываний – это часть диспетчера. Таким образом, результатом поступления сигнала прерывания является приостановка текущего процесса и передача управления диспетчеру. В этот момент диспетчер разрешает планировщику обновить состояние таблицы процессов (например, возможно, что приоритет только что отработавшего свой квант времени процесса следует понизить, а приоритеты других процессов – повысить). Затем из таблицы процессов диспетчер выбирает процесс с наивысшим приоритетом из числа процессов, находящихся в состоянии готовности, заново инициализирует цепь таймера, после чего разрешает выбранному процессу использовать новый квант времени.

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 эта ситуация есть 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 сохранять эту информацию 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.

Иногда использование процессом предоставленного ему кванта времени заканчивается раньше, чем истекает интервал,

установленный на таймере. Например, если процесс выдает запрос на выполнение операции ввода/вывода, скажем запрос на получение данных с диска, выделенный процессу квант времени будет принудительно завершен, так как в противном случае процесс бесполезно потратит оставшееся время на ожидание, когда контроллер выполнит запрос. В этом случае планировщик обновит таблицу процессов, отразив в ней переход данного процесса в состояние ожидания, а диспетчер предоставит очередной квант времени процессу, находящемуся в состоянии готовности. 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.

Модель "клиент/сервер". Различные составляющие операционной системы обычно выполняются как отдельные процессы, конкурирующие в системе с разделением времени за получение от диспетчера квантов времени ЦП. Для координации своих действий этим процессам необходимо взаимодействовать друг с другом. Например, чтобы запланировать новый процесс, планировщик должен получить для него место памяти от программы управления памятью, а чтобы получить доступ к файлу, массовой памяти, любой процесс должен сначала получить информацию об этом файле от программы управления файлами.

Обмен сообщениями между процессами называется *межпроцессным взаимодействием* (interprocess communication) и является объектом постоянных исследований. Межпроцессное взаимодействие может иметь самые разные формы. Одна из них (рис. 3.7) – *модель "клиент/сервер"* (client/server model) – широко применяется как для взаимодействия компонентов операционной систем, так и в организации компьютерных сетей.

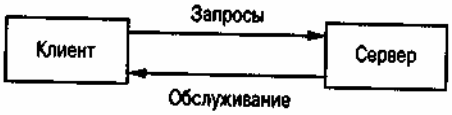


Рис. 3.7. Модель "клиент/сервер"

Согласно этой модели, каждый компонент выступает в роли клиента, посылающего запросы другим компонентам, или же в роли сервера, отвечающего на запросы, поступившие от клиентов. Например, система управления файлами функционирует как сервер, предоставляющий доступ к файлам в соответствии с запросами, поступающими от различных клиентов. Построенное в соответствии с этой моделью взаимодействие между процессами внутри операционной системы предусматривает поступление запросов от процессов, выполняющих роль клиентов, и предоставление ответов на них другими процессами, играющими роль серверов.

При разработке программного обеспечения соблюдение принципов модели "клиент/сервер" позволяет четко определить роли отдельных его элементов. Клиент просто посылает запросы серверам и ожидает поступления ответов, а сервер выполняет обслуживание поступивших запросов и посылает ответы клиентам. Роль сервера не зависит от того, функционирует ли клиент на этой же машине или на удаленной, соединенной с данной машиной через сеть. Различия существуют в программном обеспечении, используемом для их взаимодействия, но не в клиенте или сервере. В результате, если компоненты некоторой программной системы будут организованы как клиенты и серверы, то они смогут выполнять свои функции независимо от того, функционируют ли они на одной машине или на различных машинах, разделенных огромным расстоянием (рис. 3.8). Поэтому, если программное обеспечение, образующее нижний уровень системы, предоставляет средства обмена запросами и ответами на них, серверы и клиенты могут быть распределены по машинам в любой конфигурации, как это будет удобнее в данной сети.

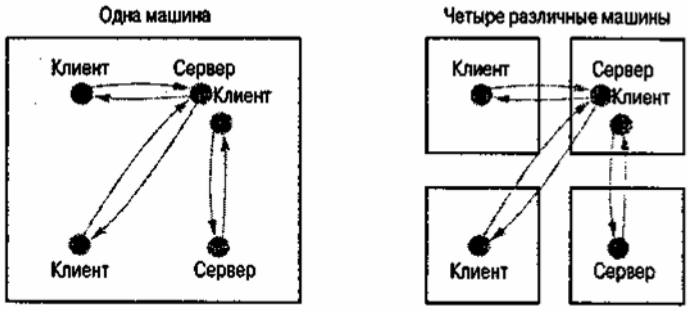


Рис. 3.8. Идентичность схемы взаимодействия клиентов и серверов, функционирующих на одной и той же машине и на разных машинах

Желание установить единообразную систему отправки сообщений, которая сможет поддерживать такую распределенную систему в компьютерной сети, является основополагающей целью ряда стандартов и спецификаций, известных как *CORBA* (Common Object Request Broker Architecture – архитектура брокеров запросов общих объектов). Спецификация CORBA включает систему стандартов, регламентирующих сетевые взаимодействия элементов программного обеспечения, называемых объектами (такими, как клиенты или серверы). Она была разработана группой OMG (Object Management Group – группа по управлению объектами), представляющей собой консорциум фирм-производителей аппаратного и программного обеспечения, а также пользователей, заинтересованных в расширении сферы применения объектно-ориентированной технологии, с которой мы познакомимся в главе 5 и к которой будем неоднократно возвращаться в последующих главах.

Вопросы для самопроверки

1. Кратко опишите различия между программой и процессом.
2. Кратко 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. В системе с разделением времени каждый квант времени равен 50 миллисекундам, а на каждое переключение между процессами затрачивается 5 миллисекунд. 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 5 миллисекунд от начала каждого выделенного ему кванта времени?

6. Определите, какие взаимосвязи в обществе отвечают модели "клиент/сервер"?

3.4. ОРГАНИЗАЦИЯ КОНКУРЕНЦИИ МЕЖДУ ПРОЦЕССАМИ*

Общей задачей всех компонентов ядра операционной системы является распределение машинных ресурсов между процессами в системе. Здесь понятие *ресурсы* 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 на диске при создании новых файлов. Программа управления памятью распределяет свободные области памяти. Планировщик распределяет место в таблице процессов, а диспетчер – кванты времени. На первый взгляд задача распределения 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 не 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 процессе в состояние ожидания, которое будет продолжаться до тех пор, пока принтер вновь не будет доступен. 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 или нет. 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 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 шагов. Следовательно, вполне возможно, что выполнение этой задачи будет прервано после того, как будет установлено, что флажок 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.

Проблема 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, выполнение ее 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* (semaphore). 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* (critical region). Требование о том, что в любой заданный момент только один процесс может выполнять команды критической области, называют *взаимным исключением* (mutual exclusion). 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.

Взаимная блокировка. Другой проблемой, которая может иметь место при распределении ресурсов, является *взаимная блокировка* (deadlock). Это состояние, при котором дальнейшая работа двух или нескольких процессов взаимно заблокирована, так как 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, 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.

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.

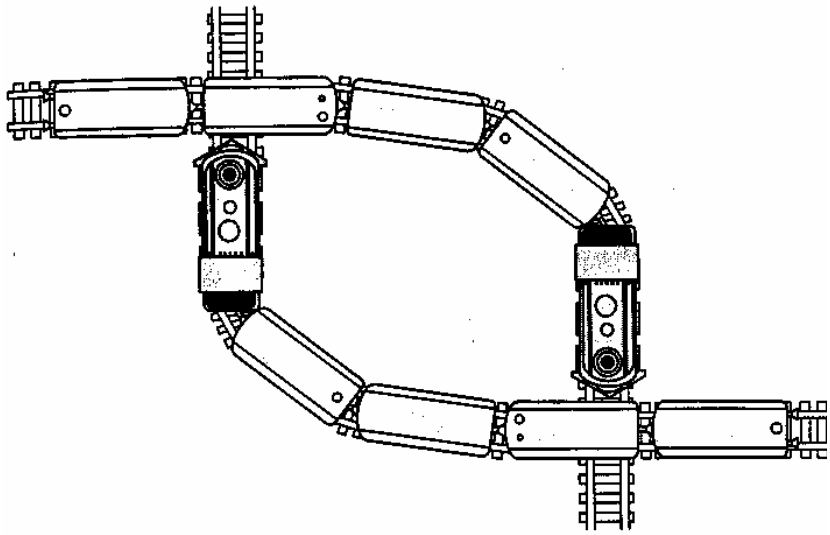


Рис. 3.9. Взаимная блокировка, возникшая в результате конкуренции за использование неразделяемых пересечений железнодорожных путей

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. Суть метода состоит в обнаружении ситуации взаимной блокировки, когда она уже возникла, и устранении ее посредством отбора некоторых предоставленных ресурсов. Приведенный выше пример с заполненной до отказа таблицей процессов можно отнести к этому классу. Как правило, системный администратор устанавливает размер таблицы процессов достаточно большим для данной конкретной системы. Если взаимная блокировка по причине переполнения таблицы процессов все же возникнет, то администратор просто использует свое право "суперпользователя", чтобы удалить (технический термин – *убить*) некоторые из процессов. В результате освобождается место в таблице процессов, после чего оставшиеся процессы смогут продолжить выполнение своих заданий.

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 на диск, 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 *спулинга* (spooling). В настоящее время она весьма распространена в операционных системах для любых платформ.

Безусловно, при конкуренции процессов за машинные ресурсы возникают и другие проблемы. 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. Кроме того, могут возникать и другие проблемы, требующие своего решения. Например, как проинформировать процессы, получившие доступ для чтения, о том, что процесс с доступом для записи изменяет файл?

Вопросы для самопроверки

1. Предположим, что процессы А и В 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 А запрашивает ресурс и значение флажка равно 0, следует удовлетворить этот запрос. В противном случае необходимо перевести процесс А в состояние ожидания. Если процесс В запрашивает ресурс и флажок имеет значение 1, следует удовлетворить этот запрос. В противном случае необходимо перевести процесс В в состояние ожидания. Каждый раз, когда процесс А 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 при подъезде машины красный свет включен, она ожидает, пока он не будет выключен, и только после этого въезжает в туннель. Какой недостаток 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, упомянутых выше в данном разделе, снимается каждым из следующих решений.

Автомобиль не заезжает на мост до тех пор, пока он не освободится.

Если на мосту встречаются две машины, одна из них едет назад.

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 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. СЕТИ

Ранние компьютерные сети состояли из машин, соединяемых для передачи файлов. Программное обеспечение, необходимое для управления такими коммуникациями между машинами, добавлялось к операционной системе в форме обслуживающего программного обеспечения. Сегодня взаимодействие между компьютерами с помощью сетей широко распространено. Многие современные системы программного обеспечения, такие, как всемирные информационно-поисковые системы, системы бухгалтерского учета компаний и даже некоторые компью-терные игры, являются *распределенными системами* (distributed system), т.е. они состоят из элементов, функционирующих на разных компьютерах сети. Поэтому программное обеспечение, необходимое для поддержки таких систем, трансформировалось из простых пакетов обслуживающих программ в систему сетевого программного обеспечения, обеспечивающую сложную инфраструктуру сети. В каком-то смысле сетевое программное обеспечение развивается в операционную систему сети. В оставшейся части этой главы мы рассмотрим некоторые вопросы, связанные с этой областью системного программного обеспечения.

Классификация сетей. Каждая компьютерная сеть принадлежит к одной из следующих обширных категорий: *локальные вычислительные сети – ЛВС* (local area networks – LAN) и *глобальные вычислительные сети – ГВС* (wide area networks – WAN). 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. Глобальная сеть соединяет машины, которые могут находиться в противоположных концах города или света. Основное различие между локальными и глобальными сетями заключается в технологиях, используемых для установления путей соединения. Например, использование спутниковых линий связи характерно для глобальных сетей, но не для локальных. В связи с этим сегодня программное обеспечение обычно заключается в небольшой изолированной части всего пакета сетевого программного обеспечения, а это означает, что с точки зрения перспектив развития программного обеспечения различие между локальными и глобальными сетями становится все менее и менее важным.

По типу физической среды передачи данных различают *беспроводные* (спутниковые, радиорелейные, оптические) и *кабельные сети*, в которых может использоваться телефонный, оптоволоконный или коаксиальный кабель, а также кабель с витой парой (скрученные между собой для уменьшения влияний внешних помех несколько пар проводов).

По скорости передачи данных сети делятся на *низкоскоростные* (до 10 Мбит/с), *среднескоростные* (до 100 Мбит/с) и *высокоскоростные* (свыше 100 Мбит/с). С увеличением скорости передачи данных возрастает стоимость линий связи, такая же связь наблюдается между дальностью передачи информации и стоимостью сетей.

При классификации по принадлежности сети делят на ведомственные (которые принадлежат одной организации и располагаются на ее территории), государственные или национальные (используемые в государственных структурах), а также международные.

Другой принцип классификации сетей базируется на том, является ли право собственности на проект внутреннего устройства сети общественным достоянием или же принадлежит отдельной корпорации. Сеть первого типа называется *открытой сетью*, а второго типа – *закрытой*, или *частной сетью*. Сеть Internet является открытой системой. Связь через Internet регулируется открытой системой стандартов, известной как семейство протоколов TCP/IP, которое мы рассмотрим в следующем разделе. В противоположность этому, компания Novell Inc. является главным поставщиком сетевого программного обеспечения, которое было разработано ею и является ее частной собственностью. Таким образом, сетевые системы, устанавливаемые и поддерживаемые с помощью программного обеспечения компании Novell, являются закрытыми.

Еще один способ классификации основывается на топологии сетей, т.е. на геометрической схеме соединения входящих в них машин. На рис. 3.10 представлены четыре наиболее распространенные конфигурации: а) "кольцо", в этом случае машины соединены по замкнутому кругу; б) "шина", здесь все машины соединены общей линией связи, называемой шиной; в) "звезда", в этой конфигурации одна машина

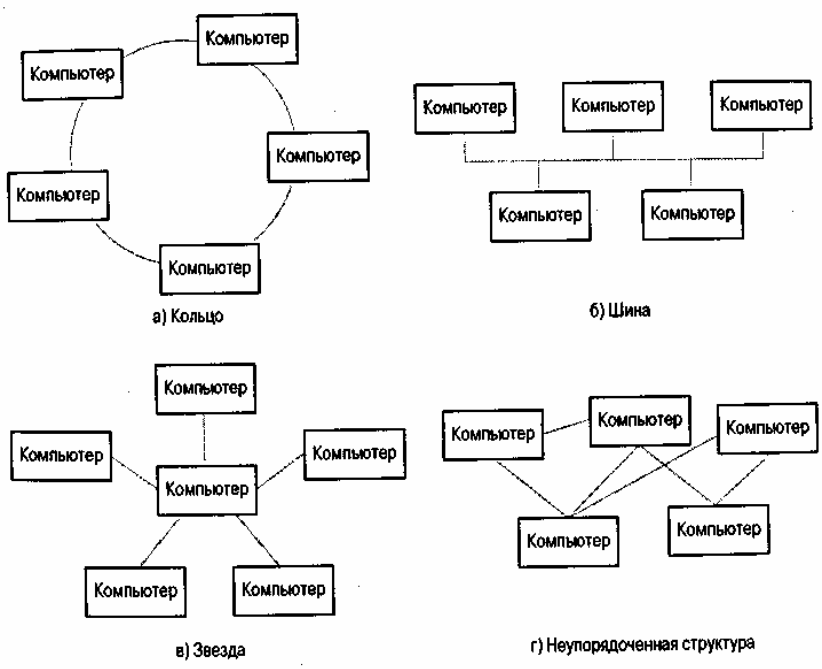


Рис. 3.10. Типы конфигурации сетей:
а – кольцо; *б* – шина; *в* – звезда; *г* – неупорядоченная структура

служит концентратором (hub), к которому подключаются все остальные машины; г) неупорядоченная, когда машины соединяются случайным образом. Неупорядоченная конфигурация характерна для глобальных сетей, тогда как при создании локальных обычно используется конфигурация "кольцо" или "шина", поскольку в этом случае работа по созданию сети чаще всего ведется под единым руководством.

Internet. Если мы соединим несколько уже существующих независимых сетей, то получим сеть из сетей. Наиболее известный пример такой структуры – глобальная суперсеть *Internet* (пишется всегда с прописной буквы), которая возникла в 1973 г. в ходе программы, начатой американским агентством DARPA, для проведения различных исследований в интересах министерства обороны США. Целью этой программы была разработка средств соединения разнообразных компьютерных сетей, позволяющих им функционировать как единая надежная сеть. В настоящее время Internet является глобальным объединением множества локальных и глобальных сетей, включающим миллионы машин. Сети в Internet соединены с помощью специальной машины, называемой маршрутизатором (рис. 3.11). *Маршрутизатор* (router) – это машина, принадлежащая к обеим сетям и передающая сообщения из одной сети в другую.[1]

Концептуально Internet на логическом уровне может рассматриваться как объединение сетевых кластеров, называемых *доменами*; каждый из доменов обычно состоит из сетей, принадлежащих к одной организации, например университету, компании или государственному учреждению. Каждый домен является автономной системой, конфигурация которой может быть выбрана местным руководством произвольно, вплоть до объединения из нескольких глобальных сетей.

Адрес каждой машины в Internet представляет собой строку из 32 бит (рис. 3.12), состоящую из двух частей: первая задает домен, в котором находится машина, а вторая определяет конкретную машину внутри домена. Часть адреса, определяющая домен, называется сетевым идентификатором и присваивается домену организацией InterNIC (Internet Network Information Center – Центр сетевой информации Internet) в процессе создания домена, при регистрации его в InterNIC. Именно эта процедура регистрации гарантирует, что каждый домен в Internet будет иметь уникальный сетевой идентификатор.

[1] Иногда вместо термина "маршрутизатор" используется термин "шлюз" (gateway). Однако шлюзами обычно называют более сложные соединения, чем типичные маршрутизаторы. Например, "шлюз" чаще всего используется по отношению к машине, соединяющей две надсети из сетей, причем в каждой надсети используется разный межсетевой протокол. В частности, шлюзом будет называться машина, соединяющая локальную корпоративную сеть с Internet.

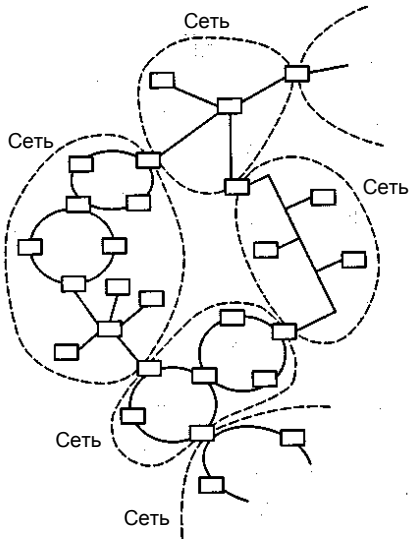


Рис. 3.11. Объединение локальных сетей

Рис. 3.12. Структура Internet-адреса машины

Часть адреса, определяющая конкретную машину в домене, называется *адресом узла*. (Термин *узел* (host) часто используется по отношению к машине в сети с целью подчеркнуть ее роль как пункта назначения для запросов, поступающих от других машин.) Адрес узла устанавливается локальной администрацией домена; обычно это администратор сети или системный администратор. Например, сетевой идентификатор Тамбовского государственного технического университета – 195.19.104. Сетевые идентификаторы традиционно записывают в десятичной нотации с точками (см. упр. 8 в конце раздела 1.4). Машина в этом домене будет иметь адрес, подобный следующему: 192.207.177.14. Последний байт здесь является адресом узла.

Представление адресов в битовой форме неудобно для нашего восприятия. Поэтому организация InterNIC присваивает каждому домену также уникальный мнемонический адрес, называемый именем домена. Каждый локальный администратор имеет право расширить это имя домена, чтобы получить мнемоническое имя для машины внутри домена. Например, имя домена Тамбовского государственного технического университета – tstu.ru. Отдельная машина в этом домене может иметь мнемоническое имя www.tstu.ru.

Нотация с точками, используемая в мнемоническом адресе, никак не связана с десятичной нотацией с точками, используемой для представления адресов в битовой форме. Напротив, элементы мнемонического адреса определяют местонахождение машины внутри иерархической системы классификации. В частности, адрес www.tstu.ru определяет машину, которой присвоено имя www в пределах организации tstu, входящий в российский домен ru. Если домен очень большой, локальная администрация может разделить его на поддомены, и тогда мнемонические адреса машин внутри домена станут длиннее. Например, пусть домену университета присвоено имя tstu.ru и пусть принято решение о разделении его на поддомены. Тогда адрес машины в университете может иметь вид ftp.is.tstu.ru; это означает, что машина с сетевым именем ftp находится в поддомене is домена tstu, входящего в домен России ru.

Локальная администрация каждого домена ответственна за ведение каталога, содержащего мнемонические адреса и соответствующие цифровые Internet-адреса машин данного домена. Каталог поддерживается на специально выделенной для этих целей машине данного домена, которая играет роль сервера, именуемого *сервером доменных имен* (domain name server – DNS). Назначение этого сервера состоит в предоставлении ответов на запросы, касающиеся адресов. Все серверы имен образуют распределенную в Internet систему каталогов, предназначенную для перевода адресов из мнемонической в эквивалентную цифровую форму. В частности, если пользователь хочет послать сообщение, причем адрес места назначения указан им в мнемонической форме, система серверов имен используется для перевода этого адреса в эквивалентную последовательность битов, совместимую с программным обеспечением Internet. Обычно эта задача выполняется за доли секунды.

Если организация принимает решение подключиться к Internet, она может стать частью уже существующего домена или найти в Internet точку, к которой можно будет подсоединить маршрутизатор и создать собственный домен. Преимущество создания нового домена состоит в том, что организация сама контролирует свое поведение, а не подчиняется администрации другой организации. Чтобы создать новый домен, организация должна зарегистрироваться в InterNIC и получить собственный сетевой идентификатор и имя домена.

Сетевые ресурсы и службы Internet. Соединенные в сеть компьютеры получают возможность совместно использовать периферийное оборудование (принтеры, плоттеры, дисководы, стримеры, сканеры, факс-модемы) и обмениваться информацией, предоставляя информационные ресурсы (каталоги, файлы, базы данных) и т.д.

Совместное использование ресурсов обеспечивает существенную экономию средств и времени. Однако в отличие от отдельного компьютера, где все ресурсы контролируются и предоставляются операционной системой, в Internet используются сетевые сервисы (службы), которые обеспечивают доступ пользователя и/или процесса к определенному ресурсу сети Интернет. В простейшем понимании сетевая служба – это пара программ (клиент и сервер), взаимодействующих между собой согласно определенным правилам (протоколам) (рис. 3.13). Соответственно, когда речь идет о работе служб Интернет, то имеется в виду взаимодействие серверного оборудования и его программного обеспечения, клиентского оборудования и его программного обеспечения.

Таким образом, чтобы воспользоваться какой-либо из служб, необходимо установить на компьютере клиентскую программу, и подключить ее к серверной программе.

Например, суперсеть Internet в настоящее время является средством публикации мультимедиа-документов, содержащих *гипертекст* (hypertext), т.е. текст со словами, фразами и графическими изображениями, связанными с другими документами.

Читатель такого текста может при необходимости получить доступ к связанным с ним документам. Для этого ему достаточно просто указать на соответствующую ссылку и щелкнуть мышью или выбрать эту ссылку с помощью клавиш со стрелками. Предположим, что в гипертекстовом документе есть следующее предложение: "Исполнение симфоническим оркестром «Болеро» Мориса Равеля было просто выдающимся", причем имя Морис Равель связано с другим документом, возможно, содержащим информацию об этом композиторе. Пользователь может просмотреть этот материал, поместив на имя Морис Равель указатель мыши и нажав кнопку мыши. Более того, если установлены соответствующие ссылки, читатель сможет прослушать аудиозапись указанного произведения, выбрав слово Болеро.

Таким способом читатель гипертекстового документа может изучать связанные с ним документы или следовать развитию мысли, переходя от документа к документу. По мере того как возрастает количество

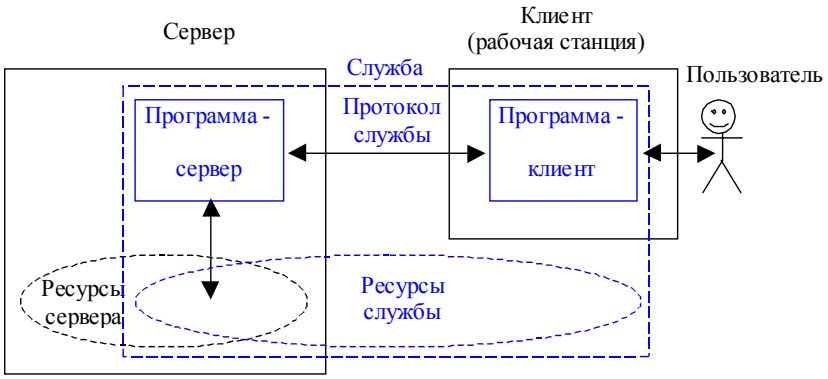


Рис. 3.13. Взаимодействие серверного оборудования и его программного обеспечения с клиентским оборудованием и его программным обеспечением

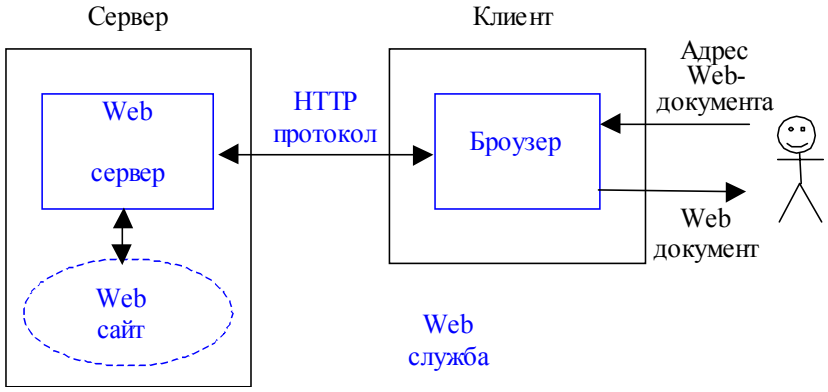


Рис. 3.14. Получение доступа к Web-документам

документов, связанных ссылками с другими документами, образуется некое подобие паутины из взаимосвязанной информации. В компьютерной сети документы, образующие эту информационную паутину, могут находиться на различных машинах, охватывая ссылками всю сеть. Подобная информационная паутина, развернутая в Internet, охватывает весь земной шар, и поэтому получила название *всемирной паутины* (World Wide Web, WWW, или просто Web).

Пакеты программ, помогающие пользователям работать с гипертекстовыми документами, относятся к одной из двух категорий: программы, исполняющие роль клиента, и программы-серверы (рис. 3.14). Программа-клиент выполняется на машине пользователя и имеет своей целью получение запрошенных пользователем материалов и представление их в подобающем виде. Именно программа-клиент предоставляет пользователю интерфейс, позволяющий этому пользователю получать доступ и просматривать информацию, размещенную в Web. Поэтому такую программу клиент часто называют *броузером* (browser), или Web-броузером. Сервер гипертекста функционирует на машине, содержащей те документы, к которым запрошен доступ. Задача сервера – предоставить доступ к размещенным на его машине документам (Web-сайту) в соответствии с запросами, поступающими от клиентов. Короче говоря, пользователь получает доступ к гипертекстовым документам, пользуясь услугами функционирующего на его машине броузера, а этот броузер выполняет требования пользователя, обращаясь с запросами к службам гипертекстовых серверов, размещенных в Internet.

Набор доступных на сегодняшний день броузеров включает разнообразные продукты конкурирующих фирм. Эти программы способны работать с документами, включающими звукозаписи, фотографии и видеозаписи. Такие документы иногда называют *гипермедиа*, чтобы подчеркнуть их отличие от традиционного гипертекста.

Для создания гипертекста необходимо иметь средства установки ссылок между документами. Для этих целей каждый документ идентифицируется уникальным адресом, который называется *URL-адресом* (Uniform Resource Locator – унифицированный указатель ресурса). Содержащаяся в URL информация позволяет броузеру связаться с соответствующим сервером и запросить требуемый документ. Типичный URL-адрес имеет вид: http://www.is.tstu.ru/is/inph/index.html. Здесь http – протокол для доступа к документу; www.is.tstu.ru – мнемоническое имя узла, на котором находится документ; /is/inph/ – путь к каталогу, определяющий местонахождение документа внутри файловой системы узла; index.html – имя документа. Иногда URL-адрес не указывает явным образом на конкретный документ, а состоит лишь из имени используемого протокола и мнемонического имени машины. В подобных случаях сервер этой машины возвращает определенный документ (обычно называемый основной страницей (home page)),

кратко описывающий информацию, доступную на этой машине. Такие сокращенные URL-адреса являются средством установления контакта с организациями. Например, URL-адрес http://www.tstu.ru указывает на основную страницу Тамбовского государственного технического университета, содержащую ссылки на многие документы, связанные с этим университетом и образовательными учреждениями Тамбова.

Гипертекстовый документ похож на традиционный текстовый тем, что его содержание также символ за символом закодировано с использованием таблицы символов стандарта ASCII или Unicode. Различие же состоит в том, что гипертекстовый документ дополнительно содержит специальные маркеры, подробно описывающие, как этот документ должен выглядеть на экране компьютера и какие элементы этого документа должны быть связаны с другими документами. Данная система маркеров получила название *языка разметки гипертекстов HTML* (Hyper Text Markup Language). Таким образом, при создании Web-страницы автор помещает в нее информацию, необходимую броузеру клиента для выполнения его задач, записывая ее на языке HTML.

В целом работа других служб аналогична, однако для них применяются соответствующие протоколы и сетевое программное обеспечение (см. табл. 3.1). Например, сервис *электронной почты* обеспечивает направленную передачу сообщений от одного человека к другому. Чтобы воспользоваться электронной почтой, необходимо соблюсти протоколы отправки и принятия сообщений и иметь программу (*почтовый клиент*) и установить связь с *почтовым сервером*. Электронная почта использует протоколы *SMPT* и *POP3*.

Таблица 3.1

Служба	Протокол	Программа-сервер	Программа-клиент
Web-служба (World Wide Web)	HTTP	Web-сервер	Броузер
Электронная почта (E-mail)	POP3, SNTP	Почтовый сервер	Почтовый клиент
Служба новостей (телеконференции Usenet)	NNTP	Сервер новостей	Клиент службы новостей
Служба передачи файлов (FTP-служба)	FTP	Ftp-сервер	Ftp-клиент

В отличие от электронной почты, *служба новостей* предоставляет ненаправленную передачу сообщений от одного человека всем, кто пожелает с этим сообщением познакомиться. Обмен сообщениями между *клиентом службы новостей* и *сервером новостей* обеспечивается с использованием протокола *NNTP*.

Для передачи файлов через Интернет используется *FTP-служба*, получившая название по одноименному протоколу. Соответственно, чтобы получить файл, необходимо иметь на компьютере программу, являющуюся *ftp-клиентом* и установить на компьютере связь с *ftp-сервером*, предоставляющим доступ к своему *файловому архиву*. Для обмена файлами используется *FTP*-протокол.

Вопросы для самопроверки

1. Что такое открытая сеть?
2. Что представляет собой маршрутизатор?
3. Что является компонентами полного Internet-адреса машины?
4. Что такое броузер?
5. Пусть локальная сеть имеет кольцевую конфигурацию. В чем состоит основной недостаток ограничения, разрешающего передачу сообщений только в одном направлении?

3.6. СЕТЕВЫЕ ПРОТОКОЛЫ*

Правила, регулирующие взаимодействие различных компонентов вычислительной системы, называются *протоколами* (protocols). Этот термин возник по аналогии с протоколом, регулирующим взаимоотношения между людьми в обществе. В компьютерной сети протоколы определяют детали каждого действия, включая то, как адресуются сообщения, как между машинами передается право передачи сообщения, как должны решаться задачи упаковки сообщений для передачи и дальнейшей обработки уже распакованных поступивших сообщений.

Многоуровневый подход к организации сетевого программного обеспечения. Главной задачей сетевого программного обеспечения является предоставление абстрактных механизмов для передачи сообщений по сети. В случае с Internet эта задача также включает передачу сообщений между сетями. Процесс передачи сообщений аналогичен процедуре, к которой мы прибегаем, если необходимо отослать некоторую запчасть к автомобилю заказчику, проживающему в другом городе. Прежде всего, запчасть упаковывают и на пакете пишут адрес получателя. Затем пакет доставляют в компанию по перевозке грузов. Работники компании помещают его в большой контейнер вместе с другими пакетами и отправляют в агентство некоторой авиалинии. Служащие авиалинии грузят контейнер в самолет, который отправится в нужный город, возможно, с промежуточными остановками по пути. В месте назначения служащие авиалинии выгружают контейнер из самолета и отправляют в контору компании по перевозке грузов. Наконец, работники компании извлекают наш пакет из контейнера и доставляют его адресату.

Говоря коротко, транспортировка запасных частей производится с помощью трехуровневой иерархии служб доставки (рис. 3.15). Первый уровень – это уровень пользователя, он состоит из отправителя и получателя. Второй уровень представлен компанией по перевозке грузов, а третий – авиалинией. Каждый из уровней рассматривает следующий

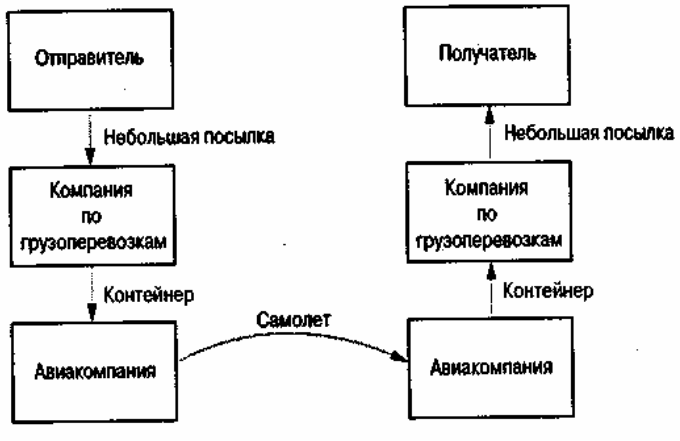


Рис. 3.15. Пример доставки посылки

более низкий уровень как некий абстрактный механизм доставки. (Отправитель не вникает в детали работы компании по перевозке грузов, а эта компания не интересуется внутренними делами авиалинии.) На каждом уровне имеются и отправители, и получатели, причем действия получателей противоположны действиям соответствующих отправителей. Сходным образом организовано и программное обеспечение, управляющее взаимодействиями через Internet; только в нем имеется четыре уровня вместо трех, и каждый уровень представляет собой набор стандартных программ, а не людей и организаций.

Четыре уровня программного обеспечения Internet (прикладной, транспортный, сетевой и канальный) представлены на рис. 3.16. Левый столбец представляет уровни программного обеспечения, используемые машиной для отправки исходного сообщения, а правый – уровни, используемые машиной при обработке поступившего сообщения. Как и в примере с доставкой груза, уровни программного обеспечения при обработке отправляемого и получаемого сообщения одни и те же.

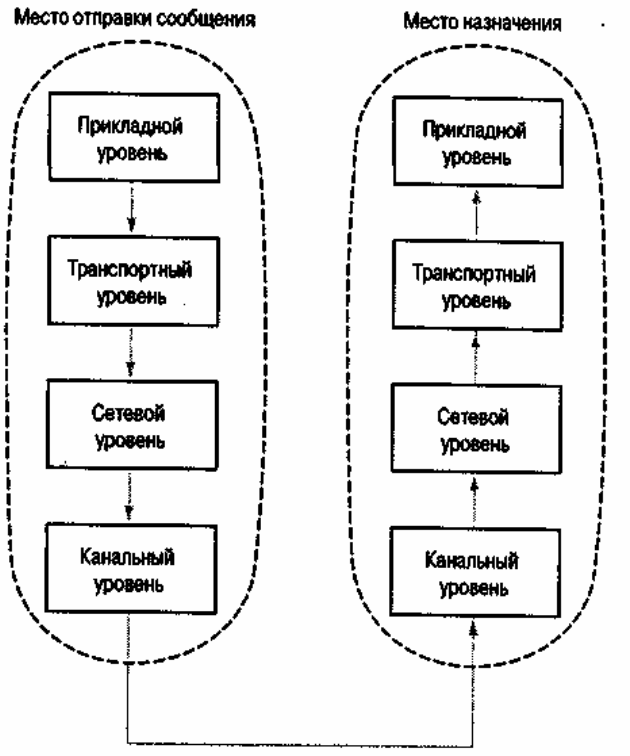


Рис. 3.16. Уровни сетевого программного обеспечения в Internet

На рис. 3.16 стрелки обозначают путь, который проходит сообщение. Обычно сообщение порождается на прикладном уровне. Оттуда оно последовательно передается от уровня к уровню по мере подготовки к отправке, спускаясь по левому столбцу, как показано на рисунке. Наконец, оно покидает канальный уровень породившей его машины и поступает на канальный уровень машины-получателя. Здесь сообщение продвигается вверх по той же иерархии уровней, пока не достигнет прикладного уровня машины-получателя.

Самым верхним уровнем в иерархии программного обеспечения Internet является прикладной уровень, который, несмотря на схожесть названий, не следует путать с прикладным программным обеспечением в классификации, приведенной в разделе 3.2. (В действительности мы скоро увидим, что большая часть программного обеспечения прикладного уровня в иерархии программного обеспечения Internet относится к категории обслуживающих программ.) Этот уровень иерархии состоит из элементов программного обеспечения, которые должны взаимодействовать друг с другом через Internet. Традиционный пример – набор стандартных утилит для передачи файлов по Internet с использованием протокола передачи файлов FTP (File Transfer Protocol). Этот набор стандартных утилит часто оформляется в виде единой прикладной программы с именем FTP, название которой отражает лежащий в ее основе протокол. Другой пример – пакет утилит под именем telnet, который был разработан в целях предоставления пользователям доступа к любой машине в Internet в таком режиме, как если бы они были

локальными пользователями этой машины. Как программа FTP, так и пакет telnet первоначально создавались как прикладное программное обеспечение в соответствии с классификацией, приведенной в разделе 3.2. Однако сегодня они стали частью инфраструктуры операционных систем большинства персональных компьютеров. Действительно, эти элементы системного программного обеспечения используются как абстрактные механизмы для конструирования более крупных приложений, подобных Web-броузерам. В этом смысле они превратились теперь в типичные программы класса утилит.

Транспортный уровень программного обеспечения Internet рассматривает прикладной уровень как источник отправляемых сообщений. Это означает, что прикладной уровень передает сообщения, которые нужно отправить, транспортному уровню аналогично тому, как посылку сдают в компанию грузоперевозок. И так же, как в обязанности отправителя посылки входит написание адреса получателя в форме, соответствующей требованиям компании грузоперевозок, обязанностью прикладного уровня является предоставление адреса доставки в формате, отвечающем требованиям транспортного уровня. Именно для решения этой задачи прикладной уровень нуждается в услугах серверов имен Internet, к которым он обращается в целях перевода мнемонических адресов, понятных людям, в двоичные адреса, совместимые с программным обеспечением сети.

Задача транспортного уровня машины-отправителя сообщения – проследить за тем, чтобы сообщение было успешно доставлено транспортному уровню компьютера-получателя, где оно будет вновь преобразовано в форму, соответствующую прикладному уровню. Таким образом, обязанности транспортного уровня состоят в выполнении тех действий, которые должны производиться в месте отправления сообщения и в месте его получения, без учета возможных промежуточных остановок по пути следования. В частности, в задачу транспортного уровня входит разбиение длинных сообщений на сегменты, размеры которых совместимы с требованиями лежащего ниже сетевого уровня. Отдельные сегменты последовательно нумеруются, что позволяет машине-получателю воссоздать первоначальное сообщение после пересылки. Затем транспортный уровень присоединяет к каждому сегменту адрес места назначения и передает сформированные блоки данных, называемые пакетами, на сетевой уровень.

Как видите, сообщения путешествуют по Internet в виде небольших пакетов, каждый из которых содержит фрагмент исходного сообщения и дополнительную "упаковку", содержащую информацию, необходимую для передачи сообщения. Нередко бывает так, что размеры упаковки пакета превышают размеры находящегося внутри элемента сообщения. Достаточно часто встречаются элементы сообщений, состоящие из одного байта, в то время как каждый пакет содержит более 50 байтов упаковки. Хотя это и кажется неэффективным, тем не менее, вся система работает достаточно хорошо.

В обязанности сетевого уровня входит наблюдение за тем, чтобы полученные им пакеты надлежащим образом передавались от одной сети в составе Internet к другой, пока они не достигнут места назначения. Таким образом, в отличие от транспортного уровня, имеющего дело только с исходным и конечным пунктами пересылки сообщения, сетевой уровень контролирует промежуточные этапы прохождения пакетов через Internet. Он решает эту задачу посредством прикрепления к каждому пакету адреса промежуточного пункта назначения. Адрес промежуточного пункта назначения определяется следующим образом. Если конечное место назначения находится в данной сети, то прикрепляемый адрес будет простым повторением адреса пункта конечного назначения. В противном случае это будет адрес маршрутизатора, установленного в данной сети. Таким образом, пакет, предназначенный машине внутри данной сети, будет послан непосредственно этой машине, а пакет, предназначенный машине за пределами сети, будет отправлен маршрутизатору, через который он попадет в смежную сеть. Для достижения указанной цели сетевой уровень снабжает пакеты, полученные от транспортного уровня, дополнительной упаковкой, содержащей, как правило, промежуточный адрес, а не конечный. Эти расширенные пакеты передаются на канальный уровень.

В обязанности канального уровня входит учет всех деталей установки соединений, присущих той сети, где находится данная машина. Каждая отдельная сеть в составе Internet имеет собственную систему адресации, независимую от общей системы адресации в Internet. Поэтому канальный уровень должен перевести Internet-адреса, указанные во внешней оболочке пакета, в адреса, соответствующие локальной адресной системе (6-битовые MAC-адреса (Media Access Control) сетевого адаптера, который назначается производителями оборудования и является уникальными адресами), после чего добавить их к пакетам в виде дополнительного слоя упаковки.

Каждый уровень в иерархии программного обеспечения Internet играет определенную роль и в процессе получения сообщений, которая в общих чертах противоположна задаче, выполняемой на данном уровне при отправке сообщения. Так, канальный уровень получает пакеты из сетевых линий связи, удаляет внешнюю упаковку (адрес данной машины в форме, совместимой с этой локальной сетью), в которую поместил данный пакет канальный уровень той машины, откуда этот пакет поступил, а затем передает обработанный пакет своему сетевому уровню.

Каждый раз, когда сетевой уровень получает пакет от своего канального уровня, он удаляет из него промежуточный Internet-адрес, прикрепленный сетевым уровнем последней машины-отправителя, и изучает адрес конечного места назначения пакета, находящийся под ним. Если это собственный адрес данной машины, сетевой уровень передает обработанный пакет своему транспортному уровню. В противном случае сетевой уровень решает, что пакет должен быть передан по Internet дальше. Для этого он должен снова "упаковать" пакет, снабдив его новым промежуточным адресом, и вернуть его на свой канальный уровень для дальнейшей передачи. Таким способом пакеты "перепрыгивают" с машины на машину, пока не достигнут конечного места назначения. На каждом промежуточном этапе именно сетевой уровень определяет место назначения следующего прыжка.

Чтобы упростить процесс переадресации, сетевой уровень поддерживает таблицу маршрутизации, содержащую конечные адреса, с которыми ему приходилось сталкиваться в недавнем прошлом, и промежуточные адреса, куда он направил каждый из этих пакетов. Сетевые уровни различных машин в Internet постоянно обмениваются данными из своих таблиц маршрутизации, в результате чего информация об адресах пересылки распространяется по Internet. Сетевой уровень каждой машины хранит только ту информацию, которая, по его мнению, может ему пригодиться, и постоянно удаляет устаревшие сведения из своей таблицы маршрутизации; в результате таблицы не разрастаются до недопустимо больших размеров. Следовательно, содержание таблиц маршрутизации в Internet является динамическим, и вполне возможно, что пакеты, представляющие разные части одного и того же сообщения, пройдут через Internet различными путями.

Заметим, что сетевой уровень передает транспортному уровню только те пакеты, которые адресованы данной локальной машине. Таким образом, в пересылке пакетов, предназначенных другим машинам, участвуют только канальный и сетевой уровни. Транспортный и прикладной избавлены от этой обязанности и обрабатывают только те пакеты, которые предназначены их собственной машине (рис. 3.17). Когда транспортный уровень получает пакеты от сетевого уровня, он выделяет из них сегменты сообщения и воссоздает первоначальное сообщение, используя последовательную нумерацию фрагментов, выполненную транспортным уровнем в месте отправления сообщения. Как только сообщение будет готово, транспортный уровень передает его прикладному уровню, завершая, таким образом, процесс передачи сообщения.

В заключение отметим, что передача данных через Internet предусматривает взаимодействие нескольких уровней программного обеспечения, расположенных на многих машинах этой глобальной системы. Учитывая все это, кажется совершенно невероятным, что среднее время ответа в Internet измеряется в миллисекундах. И действительно, большинство описанных действий происходит фактически мгновенно.

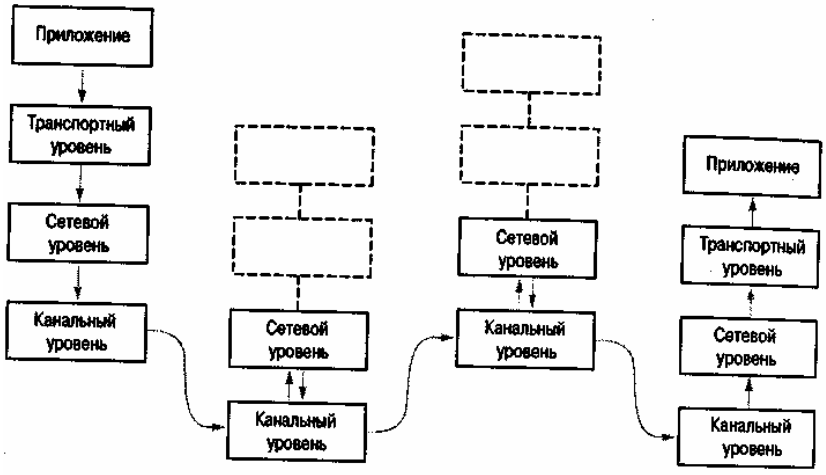


Рис. 3.17. Путь прохождения данных по Internet, включающий две промежуточные машины

Управление правом передачи данных. В обязанности канального уровня входит учет всех деталей установки соединений, присущих той сети, где находится данная машина, в том числе задача управления правом машины передавать по сети собственное сообщение. Одним из подходов к координации распределения прав на отправку сообщений является *протокол с передачей маркера по сети с кольцевой конфигурацией* (Token Ring protocol). Согласно этому протоколу, каждая машина передает сообщения только "направо", а получает их только "слева", как показано на рис. 3.18. Следовательно, сообщение от одной машины к другой передвигается по сети против часовой стрелки до тех пор, пока не достигнет своего места назначения. Когда сообщение поступает по назначению, машина-получатель сохраняет его копию и отправляет ее дальше по сети. Когда отправленная копия достигает машины-отправителя, эта машина определяет, что ее сообщение получено, и удаляет его из кольца. Конечно, такая система зависит от четкого взаимодействия всех машин. Если каждая машина постоянно будет отправлять только свои сообщения, а не пересылать сообщения других машин, то ничего не получится.

Чтобы разрешить эту проблему, по кольцу посылается определенная последовательность битов, называемая *маркером* (token). Обладание этим маркером дает машине право послать собственное сообщение; в противном случае ей разрешено передавать только чужие сообщения. В обычной ситуации каждая машина просто передает маркер слева направо, точно так же, как она передает сообщения. Если машина, получившая маркер, имеет собственные сообщения, которые нужно передать по сети, она посылает одно сообщение, удерживая маркер у себя. Если это сообщение завершит свой кольцевой цикл, машина передаст маркер следующей машине в кольце. Аналогично, когда следующая машина получает маркер, она может или сразу передать маркер дальше,

Рис. 3.18. Обмен информации в сети с кольцевой конфигурацией

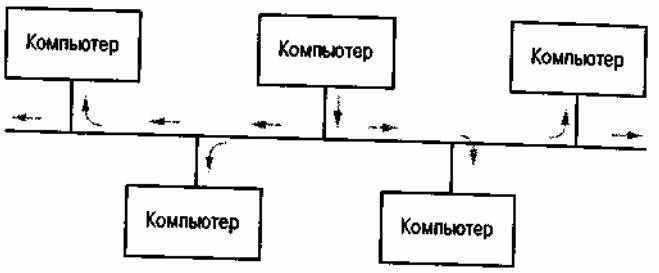


Рис. 3.19. Обмен информации в сети с шинной конфигурацией

или послать собственное сообщение, перед тем как передать маркер следующей машине. Таким образом, машины в сети имеют равные возможности вводить свои сообщения по мере циркуляции маркера в сети.

Другой протокол для координации распределения прав отправки сообщений применен в сетях Ethernet – популярной версии сети с шинной конфигурацией. В таких сетях право передачи сообщений регулируется протоколом *CSMA*/*CD* (Carrier Sence, 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. 3.19). 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.

Семейство протоколов TCP/IP. Спрос на открытые сети вызвал потребность в разработке открытых стандартов. Следуя этим стандартам, 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 четырехуровневой системы, принятой в Internet. На модель OSI часто ссылаются, так как ее поддерживает авторитет международной организации. Но она так и не была окончательно внедрена, главным образом из-за того, что появилась уже после того, как семейство протоколов TCP/IP получило широкое распространение в Internet.

Семейство протоколов TCP/IP представляет собой набор протоколов, определяющих четырехуровневую иерархию, используемую в Internet, основанную на представлении о том, что сетевая инфраструктура содержит четыре вида объектов: ресурсы, процессы, хосты, сети (см. табл. 3.2).

Таблица 3.2

Название уровня	Объекты	Адресация	Протоколы	Название блока информации
Уровень приложений	Ресурсы	Определяется конкретным сервисом	HTTP, FTP, POP3, SMTP	Сообщение
Транспортный уровень	Сервисы (процессы)	Номер порта	TCP, UDP	Сегмент (пакет)
Сетевой уровень	Сети	IP-адрес	IP	Датаграмма (пакет)
Канальный уровень	Хосты (узлы)	Локальный адрес узла (MAC-адрес в сетях Ethernet)	CSMA/CD (в сетях Ethernet)	Кадр (пакет)

В действительности TCP/IP – это название только двух протоколов: *TCP* (Transmission Control Protocol – протокол управления передачей) и *IP* (Internet Protocol); так что по отношению ко всему 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 существуют два способа; второй определяется протоколом *UDP* (User Datagram Protocol – протокол датаграмм пользователя). Это напоминает ситуацию, когда при отправке запчастей отправитель имеет возможность выбора среди различных компаний по доставке грузов, каждая из которых предлагает одинаковый набор основных услуг, но имеет собственные 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.

Протоколы транспортного уровня вводят новый уровень адресации, так называемый *номер порта* (port number), который определяет, какому процессу на машине передаются данные. Существует список соответствия номеров портов приложениям, определенных в RFC1700 (Request For Comments – запрос для пояснений – данные документы описывают стандарты протоколов Internet и их взаимодействие). Некоторые зарезервированные порты представлены в табл. 3.3.

Таблица 3.3

№ порта	Сервис	Описание
20	FTP-data	Передача данных
21	FTP	Управляющие команды
23	Telnet	Удаленный доступ в систему
25	SMTP	Протокол электронной почты
53	DNS	Сервер доменных имен
80	HTTP	Сервер WWW
110	POP3	Протокол электронной почты
119	NNTP	Телеконференции

Между протоколами TCP и UDP существуют два основных различия. Первое состоит в том, что, перед тем как передать данные на транспортный уровень, базирующийся на протоколе TCP, посылается 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 он приступает к передаче сегментов данных. Говорят, что транспортный уровень с протоколом TCP сначала устанавливает соединение, а затем посылает данные. Транспортный уровень, основанный на UDP, не устанавливает никакие соединения, прежде чем послать данные. Он просто посылает 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 отличие 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. Это не означает, что протокол UDP вообще плох. В действительности, транспортный уровень, основанный на UDP, работает существенно быстрее, и если прикладной уровень готов к устранению возможных последствий использования протокола UDP, последний может оказаться лучшим выбором.

Протокол IP – это стандарт Internet для сетевого уровня. Одна из его функций состоит в том, что каждый раз, когда использующий протокол IP сетевой уровень с протоколом IP подготавливает пакет к передаче на канальный уровень, он прикрепляет к пакету некоторое значение, называемое счетчиком переходов по сети, или временем жизни пакета. Это значение указывает, сколько раз пакет может пересылаться между машинами в попытках проложить свой путь через Internet. 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 Internet от бесконечной циркуляции пакетов в системе. Хотя размеры Internet 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 локальных и глобальных сетей и маршрутизаторов.

Вопросы для самопроверки

1. Какие уровни иерархии программного обеспечения Internet используются для пересылки исходящего сообщения в адрес другой машины?
2. Каковы различия между транспортным уровнем, основанным на протоколе TCP, и транспортным уровнем, основанным на UDP?
3. Как программное обеспечение Internet гарантирует, что сообщения не будут вечно блуждать по Internet?
4. Что удерживает подключенную к Internet машину от записи копий всех проходящих через нее сообщений?

3.7. БЕЗОПАСНОСТЬ

Когда машина подключена к сети, она доступна многим потенциальным пользователям. В связи с этим возникают проблемы, которые можно отнести к двум категориям: несанкционированный доступ к информации и вандализм. Одним из решений проблемы несанкционированного доступа является использование паролей, контролирующих или доступ к самой машине, или доступ к отдельным элементам данных. К сожалению, пароли можно узнать многими способами. Некоторые просто сообщают свои пароли друзьям, что является достаточно сомнительным в этическом отношении. В других случаях пароли похищаются. Один из способов состоит в использовании слабых мест в операционной системе для получения информации о паролях. Другой способ заключается в написании программы, имитирующей процесс регистрации в локальной системе; пользователи, полагающие, что они общаются с операционной системой, вводят свои пароли, которые затем запи-

сываются этой программой. Еще один способ получения паролей – поочередно вводить более очевидные пароли и наблюдать, что из этого получится. Например, пользователи, которые боятся забыть свой пароль, могут в качестве последнего использовать собственное имя. Кроме того, некоторые даты, например рождения, также весьма популярны в качестве паролей.

Чтобы помешать тем, кто пытается сыграть в игру с угадыванием пароля, в операционные системы можно включить средства, сообщающие о лавине неправильных паролей. Многие операционные системы также предусматривают возможность предоставления отчета о времени, когда учетная запись использовалась в последний раз, и о времени начала нового сеанса работы с ней. Это позволяет обнаружить любое несанкционированное использование их учетных записей. Более сложный способ борьбы с хакерами состоит в создании иллюзии успеха при введении неправильного пароля (так называемый "вход-ловушка"); при этом нарушителю предоставляется ложная информация и одновременно осуществляется попытка установить его местонахождение.

Другой подход к защите данных от несанкционированного доступа заключается в шифровании данных. Даже если похититель получит данные, заключенная в них информация будет ему недоступна. Для этого разработано множество методов шифрования. Одним из наиболее популярных методов шифрования сообщений, посылаемых через Internet, является шифрование с открытым ключом. Он позволяет посылать сообщения, не доступные посторонним. При шифровании с открытым ключом используются два значения, называемые ключами. Так называемый открытый ключ используется для кодирования сообщений и известен всем, кому разрешено создавать сообщения. Закрытый ключ требуется для декодирования сообщения и известен только тому, кто должен получать сообщения. Знание открытого ключа не позволяет декодировать сообщения. Так что большого вреда не будет, даже если он попадет в чужие руки; пользователь, получивший несанкционированный доступ, сможет посылать шифрованные сообщения, но не будет иметь возможности декодировать перехваченные сообщения. Знание закрытого ключа, безусловно, более существенно, но узнать его сложней, чем открытый, так как им владеет только один человек. Мы познакомимся с некоторыми системами кодирования с открытым ключом в главе 11.

Существует также множество правовых вопросов, касающихся несанкционированного доступа к информации. Некоторые из них имеют отношение к тому, что именно считать санкционированным и несанкционированным доступом. Например, может ли работодатель контролировать обмен информацией, осуществляемый его работником? В каких пределах провайдеру услуг Internet разрешается доступ к информации, которой обмениваются его клиенты? Какую ответственность он несет за содержание информации, которой обмениваются его клиенты? Эти и другие подобные вопросы беспокоят сегодня юридическое сообщество.

В Соединенных Штатах Америки многие из этих 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.

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, предоставляющим 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.

В целом закон ECPA подтверждает право индивидуума на тайну общения; службы, не уполномоченные на то специально, не имеют права прослушивать информацию, которой обмениваются третьи лица. Также незаконными считаются действия провайдера сетевых услуг, связанные с распространением сведений о содержании информации, которой обмениваются его клиенты. Однако законом также устанавливается следующее.

Не будет незаконным... если должностное лицо, наемный работник или агент Федеральной комиссии по связи по долгу службы и при исполнении своих обязанностей по наблюдению, осуществляемому федеральной комиссией согласно положениям главы 5 статьи 47 Кодекса Соединенных Штатов Америки, перехватывает информацию, распространяемую по проводам или электронным коммуникациям, устные переговоры по радио, а также раскрывает или использует информацию, полученную таким путем.

Таким образом, закон ECPA в явном виде предоставляет Федеральной комиссии по связи (FCC) США право контролировать электронные средства связи с некоторыми ограничениями. Это приводит к определенным и довольно сложным проблемам. Чтобы комиссия FCC могла воспользоваться предоставленными ей законом ECPA правами, системы связи должны быть разработаны и запрограммированы так, чтобы наблюдение за ними было возможно. Для обеспечения этой возможности 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 настолько сложным и дорогостоящим, что это привело к определенному смягчению требований. 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 правительство США пошло по пути создания системы регистрации, требующей обязательной регистрации ключей шифрования (или, возможно, ключей к ключам). Но мы живем в мире, где промышленный шпионаж получил столь же широкое распространение, как и военный. Отсюда понятно, что требование регистрации ключей шифрования ставит многих законопослушных граждан в затруднительное положение. Насколько надежной будет сама регистрирующая система? Этот вопрос важен не только для США. Возможность создания аналогичных систем регистрации рассматривалась также в Канаде и странах Европы.

Проблема вандализма иллюстрируется появлением таких неприятных явлений, как компьютерные вирусы и сетевые

черви. В общем случае *вирус* представляет собой сегмент программы, который сам прикрепляет себя к другим программам компьютерной системы. Например, вирус может внедриться в начало некоторой присутствующей в системе программы, так что каждый раз при выполнении программы-хозяина сначала будет выполняться программа-вирус. Она может осуществлять злонамеренные действия, которые будут заметны пользователю, или просто искать другие программы, к которым сможет прикрепить свою копию. Если пораженная вирусом программа переносится на новую машину (неважно, с помощью сети или гибкого диска), вирус начнет поражать программы на новой машине, как только перенесенная программа будет запущена на выполнение. Именно таким способом вирусы перемещаются с машины на машину. В некоторых случаях вирусы разрабатываются так, чтобы просто заражать другие программы до тех пор, пока не будет выполнено определенное условие, например до определенной даты, и лишь затем программа-вирус начнет свои разрушительные действия. В этом случае существенно повышается вероятность того, что, прежде чем его обнаружат, вирус сможет распространиться на множество машин.

Такое понятие, как *червь*, обычно применяют к автономной программе, которая сама распространяется по сети, резидентно загружаясь в машины и рассылая свои копии. Как и в случае с вирусами, такие программы создаются и для того, чтобы просто рассылать свои копии, и для нанесения определенного ущерба.

С ростом популярности сетей увеличивается и вероятность нанесения ущерба от несанкционированного доступа к информации и вандализма. В связи с этим возникает множество вопросов, касающихся благоразумного размещения важной информации на сетевой машине, ответственности за распространение неадекватно защищенной информации, а также ответственности за вандализм. В свою очередь, в ближайшем будущем можно ожидать проведения обширных дебатов по этическим и юридическим вопросам, связанным с этими проблемами.

Вопросы для самопроверки

1. С технической точки зрения термин "данные" означает некоторое представление информации, а "информация" – содержимое данных. Что защищает пароль – данные или информацию? Что из них защищает шифрование?
2. Каковы основные положения закона ECPA?
3. На примере закона CALEA покажите, что принятие требующего определенных действий закона может и не привести к выполнению этих требований на практике.

Упражнения

(Упражнения, 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. В чем состоит различие между интерактивной обработкой и обработкой в реальном масштабе времени?
4. Что такое многозадачная операционная система?
5. Какая информация содержится в таблице процессов, поддерживаемой операционной системой?
6. В чем различие между процессом, готовым к выполнению, и ожидающим процессом?
7. В чем заключается различие между виртуальной и основной памятью?
8. Какие сложности могут возникнуть в системе с разделением времени, если два процесса одновременно запрашивают доступ к одному и тому же файлу? Существуют ли ситуации, при которых программа управления файлами предоставит такой доступ? В каких случаях программа управления файлами ответит отказом?
9. Дайте определение понятиям балансировки загрузки и масштабирования в контексте многопроцессорной архитектуры.
10. Кратко опишите процесс первоначальной загрузки.
11. Предположим, что операционная система с разделением времени использует кванты времени длительностью 50 миллисекунд. Если обычно процедура позиционирования головки чтения/записи диска над нужной дорожкой занимает 8 миллисекунд и еще 17 миллисекунд 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 одной команде за каждую миллисекунду, 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.)
12. Назовите пять ресурсов, доступ к которым должна координировать многозадачная операционная система.
13. 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?
14. Какая система достигнет большей производительности – система, выполняющая два процесса, зависимых от ввода/вывода (см. вопрос 13), или система с одним процессом, зависимым от ввода/вывода, и вычислительно зависимым процессом? Почему?
15. Разработайте набор инструкций, который будет определять действия программы-диспетчера операционной системы по истечении кванта времени, выделенного выполняющемуся процессу.
16. Назовите компоненты информации о состоянии процесса.
17. Приведите пример ситуации в системе с разделением времени, при которой процесс не использует весь предоставленный ему квант времени.
18. Перечислите в хронологическом порядке основные события, которые происходят при прерывании процесса.
19. Опишите модель "клиент/сервер".
20. Что такое CORBA?
21. Опишите два способа классификации компьютерных сетей.
22. Укажите и опишите назначение составных частей следующего адреса электронной почты:

kermit@frog.animals.com

23. Дайте определение следующим понятиям:

а) Сервер имен.

б) Домен.

в) Маршрутизатор.

г) Узел.

24. Предположим, что адрес узла в Internet задан как 134.48.4.123. Каков будет его 32-битовый адрес в шестнадцатеричном представлении?

25. В чем состоит различие между открытой и закрытой сетью?

26. Дайте определение для каждого из следующих понятий:

а) Гипертекст.

б) HTML.

в) Броузер.

27. Что такое World Wide Web?

28. Укажите компоненты следующего URL-адреса и объясните их значение: http://frogs.animals.com/animals/moviestars/kermit.html.

29. В чем состоит разница между червем и вирусом в контексте компьютерных сетей?

30. Что вызывает беспокойство в отношении безопасности и секретности работы в Internet?

31. Что такое ECPA и CALEA?

32*. Объясните назначение команды "проверить и установить", присутствующей во многих машинных языках. Почему важно, чтобы весь процесс "проверки и установки" был реализован в виде одной команды?

33*. Банкир, который имеет всего $ 100 000, дает взаймы по $ 50 000 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 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?

34*. 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 отслеживает 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 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 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.

35*. Объясните, как может возникнуть ситуация взаимной блокировки при игре в шахматы, когда пешки вплотную подходят друг к другу. Что выступает здесь в роли неделимого ресурса? Каким образом эта ситуация обычно преодолевается?

36*. 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. Сначала он должен запросить все необходимые ему ресурсы первого уровня, прежде чем он 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 и почему?

37*. 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 нерегулярным интервалом. 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 будет завершено, компьютер вновь помещает в ячейку памяти значение 0. Какая последовательность действий может привести к конфликту между двумя руками?

38*. Предположим, что каждый компьютер в сети с кольцевой конфигурацией запрограммирован так, чтобы передавать сразу в обоих направлениях те сообщения, которые создаются на нем и предназначены всем остальным компьютерам в сети. Также предположим, что сначала устанавливается соединение с машиной слева, которое удерживается до тех пор, пока не будет установлено соединение с машиной справа, после чего осуществляется собственно передача сообщения. Поясните, как в этой сети может возникнуть ситуация взаимной блокировки, если всем ее машинам одновременно потребуется отправить подобные сообщения.

39*. Опишите, как можно использовать механизм очереди в процессе спулинга выводимой на принтер информации.

40*. 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 операцион-

ная 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). Что в данном случае означает это зависание? Что будет происходить в многопользовательской компьютерной системе, в которой всем выполняющимся процедурам присвоены приоритеты и ресурсы распределяются согласно этим приоритетам?

41*. 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?)

42*. В чем сходство между ситуацией взаимной блокировки и зависанием (см. вопрос 41)? В чем отличие между ними?

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*. Что представляет собой модель OSI, рекомендованная международным комитетом стандартов (OSI)?

45*. В сети с шинной конфигурацией сама шина является неделимым ресурсом, за доступ к которому машины должны соревноваться при необходимости передать сообщение. Как решается проблема тупиковых ситуаций в этом контексте?

46*. Протоколы, основанные на использовании маркеров, можно применять для управления правом передачи и в сетях, не имеющих кольцевой конфигурации. Разработайте протокол, построенный на использовании маркера и предназначенный для управления правом передачи в локальной сети с шинной конфигурацией.

47*. Опишите действия, выполняемые машиной при необходимости отправить сообщение по сети, работа которой регулируется протоколом CSMA/CD.

48*. Перечислите четыре уровня иерархии программного обеспечения Internet и опишите задачи, выполняемые каждым уровнем.

49*. С какой точки зрения протокол TCP представляется более удачным протоколом транспортного уровня, чем протокол UDP? В чем преимущества протокола UDP?

50*. Что означает утверждение о том, что UDP является протоколом, не устанавливающим соединения?

Ответы на вопросы для самопроверки

Раздел 3.1

1. Традиционным примером является очередь людей, желающих купить билет на представление. В этом случае может найтись некто, желающий обойти очередь, что может привести к разрушению структуры FIFO.

2. Пункты б и в.

3. Обработка в реальном времени означает согласование выполнения программы с процессами, происходящими в машине. Интерактивная обработка означает взаимодействие человека с программой во время ее выполнения. Для успешной интерактивной обработки требуются хорошие показатели обработки в реальном времени.

4. Разделение времени – это метод, с помощью которого многозадачность реализуется на машине с одним процессором.

Раздел 3.2

1. *Оболочка.* Осуществляет взаимодействие с внешней средой, окружающей компьютер.
Менеджер файлов. Координирует использование внешних запоминающих устройств.
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 выполняет задачи, специфические для данного пользователя.

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 – выделить больший промежуток времени для выполнения процессов, имеющих более высокий приоритет.

4. Каждую секунду машина сможет выделять по одному полному кванту времени 18 процессам.

5. В целом $^{10}/_{11}$ машинного времени может быть затрачено на собственно выполнение вычислительных процессов. Когда процесс запрашивает выполнение операции ввода/вывода данных, выделенный этому процессу квант времени процессо-

ра завершается и процессор приступает к обработке запроса. Таким образом, если каждый процесс будет выдавать запрос на ввод/вывод через 5 миллисекунд 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.

6. В качестве примеров можно предложить работу компании "Товары – почтой" и обслуживаемых ею клиентов, биржевого брокера и его клиентов или фармацевта и его клиентов.

Раздел 3.4

1. Эта система гарантирует, что в каждый момент времени ресурс будет использоваться не более чем одним процессом. Однако это 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 получает в свое распоряжение или весь мост или не получает 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 операционной системе распознать ситуацию взаимной блокировки и предпринять соответствующие действия по ее устранению.

Раздел 3.5

1. Открытая сеть – это сеть с открытыми спецификациями и протоколами, позволяющими различным поставщикам производить совместимые продукты.

2. Маршрутизатор – это компьютер, соединяющий между собой две сети. Точнее, маршрутизатор – это компьютер, соединяющий между собой две сети, использующие один и тот же протокол Internet. *Шлюз* – это компьютер, соединяющий между собой две сети, которые для работы в Internet используют разные протоколы.

3. Полный адрес узла в Internet состоит из идентификатора сети и адреса узла.

4. Адрес URL – это, в сущности, адрес документа в подсистеме World Wide Web. Броузер – это программа, которая помогает пользователю получить доступ к гипертекстовому документу.

5. Любой разрыв в кольце может нарушить связь. Если же сообщение можно передавать в обоих направлениях, то один разрыв в кольце может и не повредить связь.

Раздел 3.6

1. Канальный уровень получает сообщение и передает его на сетевой уровень. На сетевом уровне обнаруживается, что сообщение предназначено для другого компьютера, после чего к сообщению присоединяется адрес промежуточного компьютера и оно возвращается назад на канальный уровень.

2. В отличие от протокола TCP, UDP – это протокол без обратной связи, который не подтверждает получения сообщения адресатом.

3. Каждому сообщению присваивается счетчик переходов по сети, который определяет максимальное число выполнения передачи сообщения между компьютерами.

4. Фактически ничего. На любом компьютере в Internet программист может так модифицировать программное обеспечение, чтобы этот компьютер мог сохранять такие записи. Вот почему конфиденциальные данные следует шифровать.

Раздел 3.7

1. Использование паролей защищает данные (а значит, и информацию). Использование шифрования защищает только информацию.

2. В контексте нашего изложения ECPA определяет права собственности, имеющие отношение к электронным средствам связи.

3. Если соответствие закону технически невозможно, то этот закон не будет (не может) выполняться. Требования CALEA как технически, так и финансово трудновыполнимы. Это означает, что соответствие закону является проблематичным.

4. АЛГОРИТМЫ

Во второй главе уже отмечалось, что прежде чем машина сможет выполнить какую-либо задачу, ей нужно дать алгоритм (algorithm), точно описывающий, что делать. Поэтому наука об алгоритмах является краеугольным камнем вычислительной техники. В этой главе мы рассмотрим базовые понятия науки об алгоритмах, включая вопросы разработки и представления алгоритмов, а также понятия итерации и рекурсии. Мы также опишем известные алгоритмы поиска и сортировки.

4.1. ПОНЯТИЕ АЛГОРИТМА

Понятие алгоритма неформально можно определить как последовательность этапов, описывающую способ решения поставленной задачи. В данном разделе мы рассмотрим это основополагающее понятие более детально. 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, сам по себе сюжет останется прежним.

Точно так же 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.

Эту последовательность действий можно представить даже в виде электронной схемы. В каждом случае лежащий в основе алгоритм остается прежним, отличаются только методы его представления.

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. По сути, специалисты в области компьютерных наук 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 и выполнить

> *Алгоритм – это упорядоченный набор из недвусмысленных и выполнимых этапов, определяющий некоторый конечный процесс.*

Рис. 4.1. Определение алгоритма

алгоритм, представленный этой программой; поэтому процесс можно эквивалентно определить как деятельность по выполнению алгоритма. Из сказанного можно сделать заключение, что процессы, алгоритмы и программы – это различные, хотя и взаимосвязанные понятия.

Рассмотрим теперь формальное определение алгоритма, приведенное на рис. 4.1. Требование *упорядоченности* этапов алгоритма указывает, что отдельные этапы алгоритма должны составлять четко определенную структуру в смысле порядка их выполнения. Однако это не означает, что этапы должны выполняться в строгой последовательности: первый, второй и т.д. Например, 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". Он содержит множество последовательностей, которые разветвляются и вновь объединяются, по мере того как разные процессоры выполняют различные части задачи. Другими примерами могут служить алгоритмы, выполняемые такими электронными схемами, как триггеры (см. раздел 1.1), 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.

Теперь рассмотрим требование, по которому алгоритм должен состоять из *выполнимых* этапов. Чтобы оценить важность этого условия, рассмотрим последовательность инструкций.

Этап 1. Составить список всех целых положительных чисел.

Этап 2. Упорядочить этот список в убывающем порядке (от большего значения к меньшему).

Этап 3. Выбрать первое число из отсортированного списка.

Этот набор инструкций не является алгоритмом, так как этапы 1 и 2 выполнить невозможно. Никто не в состоянии составить список всех целых положительных чисел, и целые положительные числа невозможно упорядочить, начиная с "наибольшего". Для понятия "выполнимый" специалисты в области компьютерных наук используют термин "эффективный". Таким образом, если говорится, что данный этап эффективен, то, значит, он осуществим.

Еще одним требованием, налагаемым определением, приведенным на рис. 4.1, 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.

При обсуждении проблемы неоднозначности важно различать алгоритм и его представление. Неоднозначность конкретного представления алгоритма часто неправильно интерпретируется как неоднозначность, присущая самому алгоритму. Распространенным примером является степень детализации в описании алгоритма. Для метеорологов инструкция "Перевести показания в градусах Цельсия в эквивалентные им показания по шкале Фаренгейта" будет вполне удовлетворительной, но

непрофессионалы, нуждающиеся в 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 во избежание проблем неоднозначности в представлении алгоритма может использоваться концепция примитивов.

Требование, утверждающее, что алгоритм должен определять *конечный процесс,* 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 результату.

На практике требование конечности определяемого алгоритмом процесса полезно тем, что исключает бесконечные процессы, которые никогда не приведут к получению каких-либо содержательных результатов. Например, прямое следование инструкции "Выполнить этот этап еще раз" лишено смысла. 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.

Независимо от того, какую точку зрения считать правильной, на практике термин "алгоритм" часто неформально используется по отношению к последовательностям этапов, не обязательно определяющим конечные процессы. Примером может служить известный нам еще со школьной скамьи алгоритм деления в столбик, который не определяет конечный процесс в случае деления 1 на 3.

Вопросы для самопроверки

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 начале этого раздела.
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.

4.2. 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, русский, японский), и язык картинок, представленный на рис. 4.2. (На этом рисунке приведен алгоритм изготовления игрушечной птички из квадратного листа бумаги.) Однако зачастую использование этих естественных средств общения ведет к неправильному пониманию. Иногда причина 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. 4.2, 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* (primiteve). 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. Каждый примитив состоит из двух частей: синтаксической и семантической. Синтаксис относится к символьному представлению примитива, а семантика – к представляемой концепции, т.е. к значению примитива. Например, синтаксис слова "воздух" включает шесть соответствующих символов, тогда как семантически – это окружающая нас газовая субстанция.

В качестве примера на рис. 4.3 представлены некоторые примитивы, используемые в искусстве оригами.

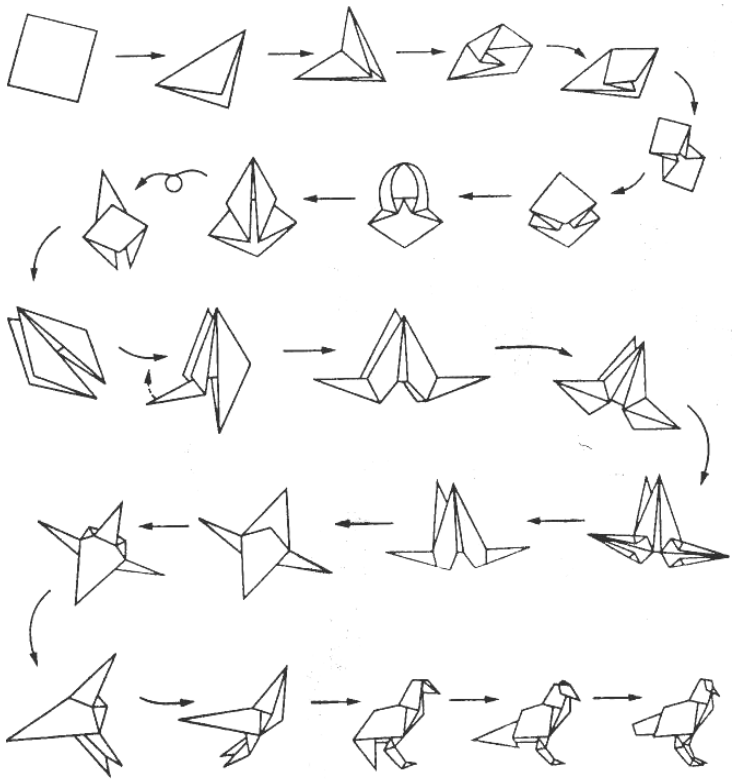


Рис. 4.2. Сложение птицы из квадратного листа бумаги

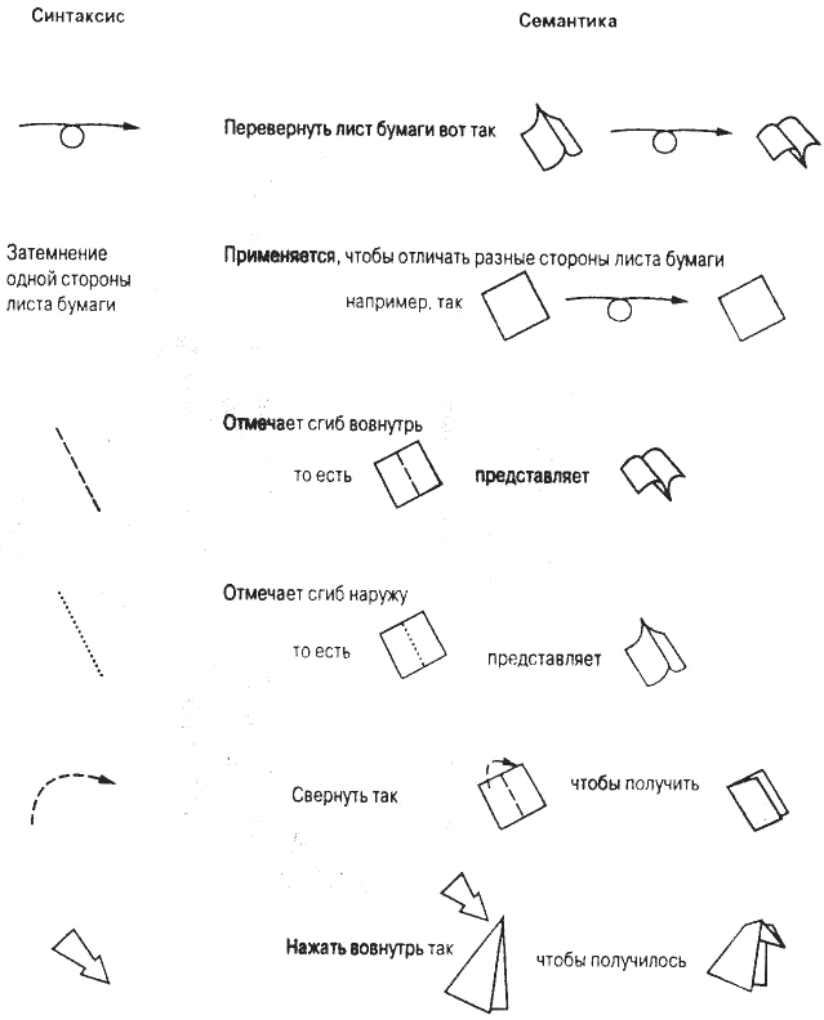


Рис. 4.3. Примитивы, используемые в оригами

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 которых 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.

Псевдокод. При построении алгоритма человек должен учитывать влияние большого количества связанных идей. Эта задача может выходить за рамки возможностей человеческого разума. Джордж А. Миллер (George A. Miller) в своей статье 1956 г. "Психологическое обозрение" (Psychological Review) изложил результаты исследования, которое показало, что человек может манипулировать одновременно только семью элементами. Поэтому разработчику сложных алгоритмов необходимо средство записи частей алгоритма для последующего возвращения к ним.

В 50-х и 60-х гг. XX в. для создания алгоритмов использовались блок-схемы, в которых алгоритм изображался с помощью геометрических фигур, соединенных стрелками. Однако блок-схемы часто становились запутанной паутиной переплетающихся стрелок, что значительно осложняло понимание лежащего в основе алгоритма. Поэтому блок-схемы уступили дорогу другим средствам проектирования алгоритмов. Примером может послужить *псевдокод* (pseudocode), в соответствии с которым алгоритмы записываются с помощью строго определенных текстовых структур (примитивов), предназначенных для неформального представления идей в процессе разработки алгоритмов.

Один из путей создания псевдокода состоит в простом ослаблении правил того формального языка программирования, на котором требуется записать окончательную версию алгоритма. Этот подход обычно используется, когда целевой язык программирования известен заранее. В подобной ситуации псевдокод, используемый на ранних стадиях разработки программы, может состоять из синтаксических и семантических структур, аналогичных структурам целевого языка программирования, но не столь формализованных.

В заключение стоит отметить, что блок-схемы могут быть полезны, когда целью является изображение алгоритма, а не его построение. Например, на рис. 4.8 и 4.9 используется представление в виде блок-схемы, чтобы наглядно продемонстрировать алгоритмические структуры, представленные операторами управления.

Поиск лучших средств записи все еще продолжается. В главе 6 мы увидим, что сохраняется тенденция использования графики в создании крупных систем программного обеспечения, а псевдокод остается распространенным средством разработки меньших процедурных компонентов системы.

Пример псевдокода. Начиная с этого момента, мы отказываемся от использования команд формального языка программирования в пользу менее формальной и более интуитивной системы обозначений, известной как псевдокод.

Однако наша задача – рассмотреть проблемы разработки и представления алгоритмов, не концентрируя внимание на каком-либо определенном языке программирования. Поэтому выбранный здесь подход к созданию псевдокода состоит в разработке непротиворечивой и краткой системы обозначений для представления повторяющихся семантических структур. В свою очередь, эти структуры станут примитивами, с помощью которых мы попытаемся выразить дальнейшие идеи.

Одна из таких повторяющихся семантических структур – присвоение значения описательному имени. Например, будем использовать для значений такие имена: Температура, СуммарноеКоличествоОсадков, БалансБанка и Положение. Для установления связи между именем и значением будем использовать форму

имя ← *выражение*

где *имя* – это описательное имя, а *выражение* описывает значение, которое присваивается этому имени. Такое утверждение читается как "присвоить имени значение выражения". Например, утверждение

```
Итог ← Цена + Налог
```

присваивает результат сложения значений `Цена` и `Налог` имени `Итог`.

Другая повторяющаяся семантическая структура – выбор одного из действий в зависимости от истинности или ложности какого-либо условия.

Примеры:

```
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.
```

Каждое из этих предложений можно переписать так, чтобы оно соответствовало следующей структуре:

if (*условие*) **then** {*действие*} **else** {*действие*}

В этой структуре ключевые слова **if** (если), **then** (то) и **else** (иначе) используются для того, чтобы объявить различные подструктуры внутри основной структуры, а скобки позволяют обозначить границы этих подструктур. Включив такую синтаксическую структуру в наш псевдокод, мы получаем универсальный способ описания указанной общей семантической структуры. Это и есть то, что требовалось сделать.

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) then {разделить итог на 366} else {разделить итог на 365}
```

Мы также примем сокращенный синтаксис этого конструкта:

if (*условие*) **then** {*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 ее позволяет сократить исходный вариант следующим образом:

```
if (продажи сократились) then {снизить цену на 5 %}
```

Другая универсальная алгоритмическая структура заключается в необходимости продолжать выполнение последовательности инструкций до тех пор, пока некоторое условие остается верным. Ниже приведено несколько неформальных примеров этой структуры.

```
До тех пор, пока есть билеты для продажи, продолжать продавать билеты.
Пока есть билеты для продажи, продолжать продажу билетов.
```

Для подобных случаев в нашем псевдокоде будет применяться следующий универсальный шаблон:

while (*условие*) **do** {*действие*}

Коротко говоря, 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) do {продавать билеты}
```

Использование отступов при размещении текста позволяет повысить читабельность программы:

```
if (товар налогооблагаемый)
    then {if (цена > минимум)
              then {платить x}
              else {платить y}
          }
 else {платить z}
```

Эта конструкция воспринимается легче, чем ее эквивалент, приведенный ниже:

```
if (товар налогооблагаемый) then {if (цена > минимум) then {платить x} else {платить y}} else {платить z}
```

Поэтому мы договоримся использовать отступы для отражения структуры текста в нашем псевдокоде. (Заметим, что мы даже будем выравнивать скобки, связанные с внешней конструкцией **then**, чтобы отметить начало и окончание этой структуры.)

Мы собираемся использовать наш псевдокод для описания действий, которые могут выступать в роли абстрактных вспомогательных программ в других приложениях. В компьютерных науках такие программные элементы имеют несколько различных названий, а именно: *подпрограммы, процедуры, модули* и *функции* (значение каждого из них имеет свой смысловой оттенок). Договоримся применять к нашему псевдокоду термин *процедура* (procedure), используя его для обозначения заголовка, по которому можно будет распознать данный блок псевдокода. Точнее говоря, мы будем начинать каждый блок псевдокода со следующей инструкции:

procedure *имя*

Здесь *имя* – это конкретное название, присвоенное данному блоку. Ниже будут следовать инструкции, определяющие выполняемые в этом блоке действия. Например, на рис. 4.4 представлен псевдокод процедуры с именем `Приветствие`, в которой трижды печатается сообщение "Привет".

```
procedure Приветствие
Счетчик ← 3
while (Счетчик > 0) do
   {напечатать сообщение "Привет";
    Счетчик  ← Счетчик - 1}
```

Рис. 4.4. Процедура "Приветствие", записанная на псевдокоде

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-then-else**, написав следующую инструкцию:

```
if (...)
    then {выполнить процедуру ПредоставлениеЗайма}
    else {выполнить процедуру ОтклонениеЗаявки}
```

В результате процедура `ПредоставлениеЗайма` будет выполняться, если проверяемое условие истинно, а процедура `Отклонение-Заявки` – если это условие ложно.

Процедура должна быть общей насколько это возможно. Например, процедура для сортировки списка имен должна сортировать любой список, а не какой-то один. То есть она должна быть написана так, чтобы параметры сортируемого списка задавались не в самой процедуре. Вместо этого в представлении процедуры список должен быть представлен формаль-

ным параметром.

В нашем псевдокоде мы будем записывать формальные параметры в скобках сразу после имени процедуры. Например, процедура с именем `Сортировка`, которая сортирует любой список имен, будет начинаться следующей инструкцией

```
Procedure Сортировка (список)
```

Если дальше в представлении упоминается сортируемый список, то для него используется имя `Список`. А когда мы вызываем процедуру `Сортировка`, мы задаем, какой именно список нужно отсортировать. Для этого можно использовать, например, следующий синтаксис:

```
Применить процедуру Сортировка к списку членов организации.
```

В зависимости от наших потребностей, мы можем применить другой вариант синтаксиса:

```
Применить процедуру Сортировка к списку гостей на свадьбе.
```

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 неформальных фраз, запрашивающих такие действия, детали которых 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).

В то же время, если позже мы обнаружим, что определенная идея повторяется в обсуждаемых нами схемах, то мы примем соответствующий синтаксис для ее представления и этим расширим наш псевдокод.

Вопросы для самопроверки

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 является наибольшим общим делителем.)

   Запишите этот алгоритм с помощью нашего псевдокода.
4. Опишите набор примитивов, используемых не в программировании, а в какой-либо другой области.

4.3. 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 – означает найти метод решения задачи, в решении которой и состоит назначение этого алгоритма. Поэтому, чтобы понять, как создаются алгоритмы, необходимо понять процесс решения задач.

Теория 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 11 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.

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 г., 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 алгоритмическая процедура позволяет 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 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.

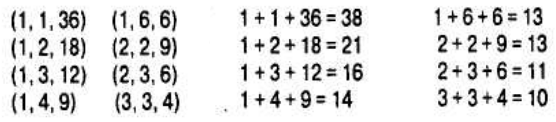
Кроме того, необходимо заметить, что эти четыре фазы не обязательно выполняются в указанном порядке. Как отмечают многие авторы, те, кто успешно решают задачи, часто начинают формулировать стратегию решения (фаза 2) еще до того, как полностью смогут понять существо задачи (фаза 1). Впоследствии, если 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 6), 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. 4.5, *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. 4.5, *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.



а) *б*)

Рис. 4.5. Иллюстрация к задаче о трех детях:
а – тройки чисел, произведение которых равно 36;
б – суммы троек чисел, приведенных в *а*)

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 г., 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.

Общие методы решения задач. 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 позиции 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. Для получения окончательного решения надо просто продолжить наши рассуждения. Поскольку прогноз участника A оказался неверным, значит, 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 занять оптимальную позицию, – 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 Полиа 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 уже были решены раньше. Позже предпринимается попытка применить способ их решения к данной задаче. Этот метод особенно важен в контексте разработки программ. Часто при разработке программы основная трудность заключается в создании общего алгоритма для решения всех разновидностей данной задачи. Другими словами, если нам необходимо разработать программу для упорядочения списка имен в алфавитном порядке, наша задача не сводится к сортировке отдельного списка, а заключается в нахождении 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 он 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 начинать" 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 метода поэтапного уточнения, свойственна естественная модульная структура. Именно в этом кроется основная причина популярности этого метода при разработке алгоритмов. Если алгоритм имеет естественную модульную структуру, то он легко реализуется в модульном представлении, способствующем созданию удобных в сопровождении программ. Более того, модульная структура, создаваемая в процессе поэтапного уточнения, легко совмещается с концепцией коллективного программирования, в соответствии с которой несколько человек объединяются в команду, имеющую своей задачей разработку определенного программного продукта. После того как исходная задача будет разбита на подзадачи (потенциальные модули), члены команды смогут работать над ними независимо, не мешая друг другу.

Метод поэтапного уточнения пользуется широкой популярностью при разработке программного обеспечения. Однако, несмотря на все его преимущества, этот метод отнюдь не является последним словом науки в отношении создания алгоритмов. В сущности, он представляет собой только средство организации работы, и все его возможности по решению задач являются лишь следствием этой организации. Вполне естественно использовать метод поэтапного уточнения при организации общенациональной политической кампании, написании курсовой работы или заключении торгового соглашения. Аналогичным образом для большинства проектов разработки программного обеспечения из области обработки данных характерна большая роль организационного компонента. Задача этих проектов состоит не столько в создании нового сенсационного алгоритма, сколько в представлении решаемых задач таким образом, чтобы их можно было реализовать в виде согласованно функционирующего пакета программ. Поэтому метод поэтапного уточнения по праву стал основой всей методологии проектирования в области обработки данных.

Однако поэтапное уточнение остается всего лишь одной из многих методологий проектирования, представляющих интерес для специалистов в области компьютерных наук. Поэтому не следует впадать в заблуждение, полагая, что все алгоритмы могут быть созданы с помощью этого метода. Фактически применение при решении задач предвзятых представлений и заранее выбран-

ных схем в некоторых случаях может даже усложнить изначально простое задание. Рассмотрим следующую задачу.

Когда вы садились в лодку, ваша шляпа упала в воду, но вы этого не заметили. Скорость течения реки 2,5 мили в час, и ваша шляпа поплыла вниз по течению. Тем временем вы поплыли в лодке вверх против течения со скоростью 4,75 мили в час (относительно воды). Спустя 10 минут вы заметили пропажу шляпы, развернули лодку и поплыли вниз по реке догонять вашу шляпу. Через какое время вы поймаете свою шляпу?

Большинство студентов, изучающих алгебру в высшей школе, 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 проплыла шляпа 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 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 мин, то вернуться к шляпе вы сможете за те же 10 мин. 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 мин, чтобы вернуться к оставленной шляпе.

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 четко определенным методологиям, значит "загубить" творческие способности учащихся, которые, напротив, нужно всемерно развивать.

Вопросы для самопроверки

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*1*1*1, 2*1*1 и 3*1. Для n, равного 5, искомая комбинация будет (2, 3).

б) Какова искомая комбинация для $n = 2001$?

в) Объясните, как вам удалось продвинуться в решении этой задачи.

2. Найдите алгоритм решения следующей задачи и ответьте на дополнительные вопросы.

а) Пусть дана квадратная шахматная доска размером $2n \times 2n$, где n – целое положительное число, и коробка с L-образными фишками, каждая из которых может закрыть в точности три квадрата на доске. 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 решение этой задачи может быть использовано, чтобы показать, что $2^n - 1$ 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 и объясните, как вам удалось сделать первый шаг в поисках решения:

```
Pdeo eo pda yknnayp wjosan.
```

4.4. 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. Будем считать, что 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 возрастания.

Давайте 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 нужное нам имя не будет найдено.

С помощью нашего псевдокода этот процесс можно представить следующим образом:

```
Выбрать в качестве ПроверяемоеЗначение первый элемент Списка;
while (ИскомоеЗначение > ПроверяемоеЗначение
и есть непроверенные элементы) do
   {выбрать в качестве ПроверяемоеЗначение следующий элемент Списка}
```

По окончании выполнения структуры **while** искомое значение либо будет найдено, либо выяснится, что его нет в списке. В любом случае успешность поиска можно установить, сравнивая искомое значение с проверяемым. Если они эквивалентны, поиск успешен. Таким образом, в конец приведенной выше программы необходимо добавить следующую инструкцию:

```
if (ИскомоеЗначение = ПроверяемоеЗначение)
    then {сообщить об успехе}
    else {сообщить о неудаче}
```

Наконец, в нашей программе предполагается, что первая инструкция, где в качестве проверяемого значения явно указан первый элемент списка, содержит, как минимум, один элемент. Конечно же, это условие могло бы выполняться всегда, однако для полной уверенности в правильности программы следует поместить составленную выше программу в предложение **else** следующей инструкции **if**:

```
if (Список пуст)
    then {сообщить о неудаче}
    else {...}
```

В результате получается процедура, текст которой приведен на рис. 4.6. Заметим, что для поиска некоторых значений в списке другие процедуры вполне могут использовать ее с помощью следующей инструкции:

```
Применить процедуру Поиск к списку пассажиров, чтобы найти имя "Даррел Бейкер"
```

Эта инструкция позволяет установить, является ли Даррел Бейкер пассажиром некоторого рейса. Вот еще один пример:

```
procedure Поиск (Список, ИскомоеЗначение)
if (Список пуст)
   then {сообщить о неудаче}
   else {выбрать в качестве ПроверяемоеЗначение
        первый элемент Списка;
        while (ИскомоеЗначение > ПроверяемоеЗначение
         и есть непроверенные элементы) do
             {выбрать в качестве ПроверяемоеЗначение
              следующий элемент списка}
         if (ИскомоеЗначение = ПроверяемоеЗначение)
              then {сообщить об успехе}
```

Рис. 4.6. Алгоритм последовательного поиска, записанный с помощью псевдокода

Применить процедуру Поиск к списку ингредиентов, используя в качестве искомого значения "мускатный орех"

Данная инструкция позволит установить, входит ли мускатный орех в перечень ингредиентов некоторого блюда.

Итак, можно сказать, что представленный на рис. 4.6 алгоритм последовательно рассматривает все элементы списка. По этой причине данный алгоритм называется алгоритмом *последовательного поиска* (sequential search). 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* (loop)**;** 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. 4.6. Здесь инструкция **while** используется в целях управления повторным выполнением единственной инструкции "выбрать следующий элемент списка как Проверяемое_значение". Общий синтаксис инструкции **while** имеет такой вид:

while (*условие*) **do** {*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*

Эти действия будут продолжаться 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:

Выполнить инструкцию "Добавить каплю серной кислоты" три раза.

Эта циклическая структура эквивалентна такой последовательности инструкций:

Добавить каплю серной кислоты.
Добавить каплю серной кислоты.
Добавить каплю серной кислоты.

Однако невозможно написать аналогичную последовательность, эквивалентную следующему циклу:

while (уровень pH больше, чем 4) **do** {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. Ведь 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. 4.7), 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 завершения цикла*. Именно для выполнения операции проверки некоторое условие обязательно указывается при записи каждой инструкции **while** нашего

Инициализация: Установить начальное состояние, которое будет модифицироваться до тех пор, пока не будет выполнено условие окончания.

Проверка: Сравнить текущее состояние с условием завершения и прекратить повторение тела цикла, если состояние соответствует условию окончанию.

Модификация: Изменить состояние таким образом, чтобы приблизиться к ситуации, в которой будет выполнено условие окончания.

Рис. 4.7. Операции управления циклом

псевдокода. Однако условие, задаваемое в инструкции **while**, – это *условие продолжения цикла,* а условие завершения – это отрицание условия, заданного в инструкции **while**. Поэтому в инструкции **while** показанной на рис. 4.6, условие завершения выглядит следующим образом:

```
(Искомое_Значение ≤ Проверяемое_Значение) или (больше нет не рассмотренных элементов)
```

Остальные две операции управления циклом гарантируют, что условие завершения обязательно возникнет. Операция *инициализации* устанавливает начальное состояние, а операция *модификации* изменяет его в направлении достижения условия завершения. Например, на рис. 4.6 операция инициализации выполняется инструкцией, предшествующей инструкции **while**. В этой операции 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 завершения обязательно будет достигнуто. (Или обнаружится проверяемый элемент, больший либо равный искомому, или будет достигнут конец списка.)

Следует подчеркнуть, что операции инициализации и модификации обязательно должны приводить к заданному условию завершения. 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) do {Число ← Число + 2}
```

В данном случае условием завершения 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 будут присваиваться значения 1, 3, 5, 7, 9, ... и ее значение никогда не будет равно 6. В результате выполнение данного цикла никогда не закончится.

Существуют два варианта широко распространенных циклических структур, которые отличаются только порядком выполнения операций управления циклом. Первая представлена инструкцией нашего псевдокода:

```
while (условие) do {действие}
```

Семантика этой циклической структуры представлена на рис. 4.8 в виде *блок-схемы*. На подобных схемах для представления отдельных этапов выполнения используются различные геометрические фигуры, а стрелки указывают порядок выполнения этих этапов. Различные фигуры отражают отдельные 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 завершения производится до того, как выполняется тело цикла.

В противоположность этому, в структуре, представленной на рис. 4.9, указывается, что тело цикла должно выполняться до проверки условия завершения. В результате тело цикла всегда выполняется хотя бы один раз, в то время как в структуре типа **while** тело цикла может не выполниться ни разу (если условие завершения будет выполнено при первой же его проверке).

В нашем псевдокоде общий синтаксис цикла этого типа имеет следующий вид:

repeat {*действие*} **until** (*условие*)

Рис. 4.8. Блок-схема цикла while-do

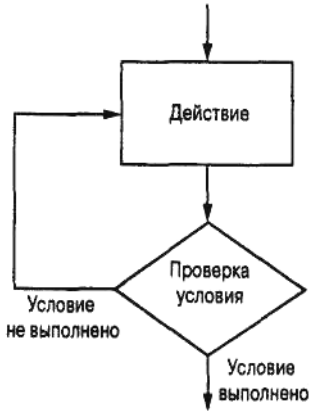


Рис. 4.9. Блок-схема цикла repeat-until

Рассмотрим конкретный пример:

repeat {взять монету из кармана} **until** (в кармане нет монет)

Когда выполнение алгоритма доходит до этой инструкции, подразумевается, что в кармане есть хотя бы одна монета. Это предположение не является обязательным при использовании следующей инструкции:

while (в кармане есть монета) **do** {взять монету из кармана}

Придерживаясь терминологии нашего псевдокода, будем называть эти структуры оператором цикла с условием продолжения и оператором цикла с условием завершения. Иногда оператор цикла **while** называют *априорным циклом* (pretest loop), или циклом с предусловием, поскольку условие завершения проверяется до выполнения тела цикла, а оператор цикла **repeat** называется *апостериорным циклом* (posttest loop), или циклом с постусловием, поскольку условие завершения проверяется после выполнения цикла.

Алгоритм сортировки методом вставки. В качестве дополнительного примера итеративной структуры рассмотрим задачу сортировки списка имен в алфавитном порядке. Прежде чем приступить к обсуждению, следует установить некоторые ограничения, которые необходимо будет учитывать. Попросту говоря, наша задача – отсортировать список "внутри его 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 задачи. Рассмотрим, как можно было бы отсортировать карточки с именами, расположенные на рабочем столе. Пусть исходный список имен выглядит следующим образом:

Fred
Alice
David
Bill
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 Alice, – еще нет. Поэтому подымем карточку с именем Alice, поместим карточку с именем Fred вниз, туда, где раньше была карточка с именем Alice, а затем положим карточку с именем Alice в самое верхнее положение, как показано в первой строке на рис. 4.10. Теперь список будет иметь такой вид:

Alice
Fred
David
Bill
Carol

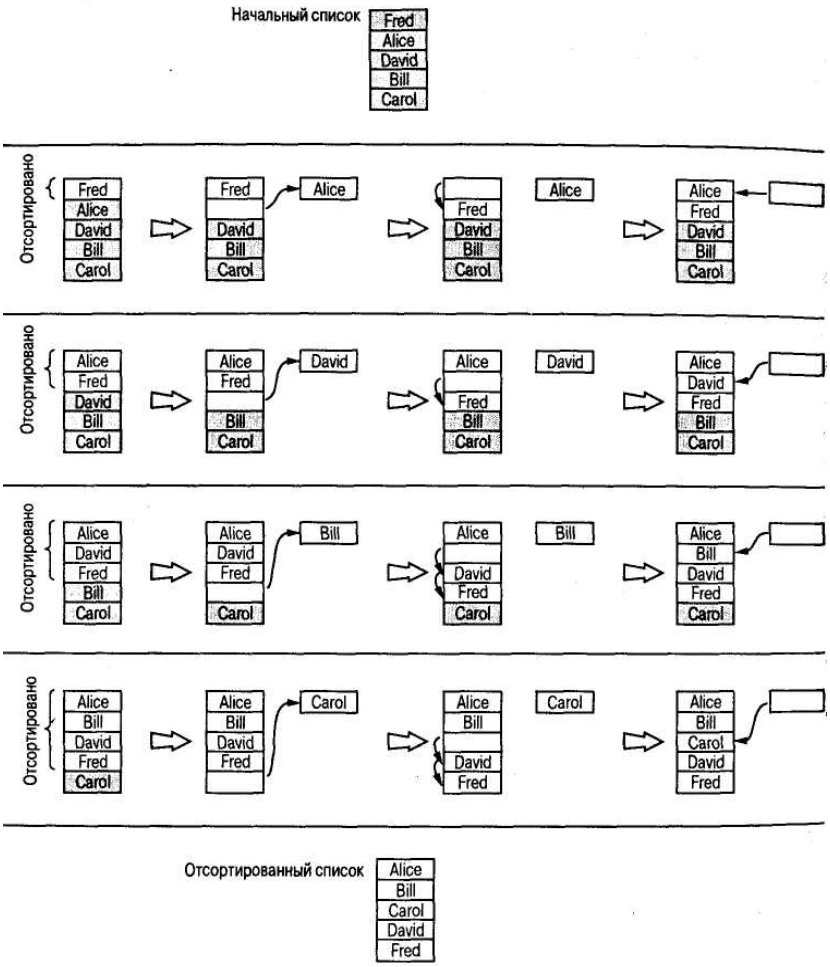


Рис. 4.10. Сортировка списка имен Fred, Alice, David, Bill и Carol

В этом варианте два верхних имени образуют отсортированный подсписок, а три верхних – нет. Подымем третью карточку с именем David, опустим карточку с именем Fred вниз, туда, где только что была карточка с именем David, а затем поместим карточку с именем David в ту позицию, которую раньше занимала карточка с именем Fred, как показано во второй строке на рис. 4.10. 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 Bill, опустить вниз карточки с именами Fred и David, а затем поместим карточку с именем Bill в освободившуюся позицию (третья строка на рис. 4.10). И наконец, чтобы завершить процесс сортировки, необходимо поднять карточку с именем Carol, опустить вниз карточки с именами Fred и David, а затем поместить карточку с именем Carol в освободившуюся позицию – как показано в четвертой строке на рис. 4.10.

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. 4.10 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 ОпорныйЭлемент во временное хранилище, оставив в списке пустое место;
while (над пустым местом есть имя, и оно > ОпорныйЭлемент) do
   {переместить имя, находящееся над пустым местом, вниз,
    оставив в его прежней позиции пустое место}
Поместить ОпорныйЭлемент 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 (N ≤ ДлинаСписка) do
   {выбрать N-й элемент как ОпорныйЭлемент;
     .
     .
     .
    N ← N + 1}
```

Здесь N – счетчик, параметр ДлинаСписка – количество элементов в списке, а точки указывают место, где должна располагаться составленная нами выше процедура.

Полный текст программы сортировки на языке псевдокода приведен на рис. 4.11. 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* (insertion sort).

Обратите внимание, что представленная на рис. 4.11 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.

```
    procedure Сортировка (Список)
    N ← 2;
    while (N ≤ ДлинаСписка) do
       {выбрать N-й элемент как ОпорныйЭлемент;
        переместить ОпорныйЭлемент во временное
хранилище,
        оставив в списке пустое место;
        while (над пустым местом есть имя, и оно >
ОпорныйЭлемент) do
           {переместить имя, находящееся над пустым
местом, вниз,
            оставив в его прежней позиции пустое
место}
        поместить ОпорныйЭлемент на пустое место в
списке;
        N ← N + 1}
```

Рис. 4.11. Алгоритм сортировки методом вставки, написанный на псевдокоде

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 превысит 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.

Вопросы для самопроверки

1. Преобразуйте показанную на рис. 4.6 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 **repeat-until**.

```
Z ← 0;
X ← 1;
while (X < 6) do
   {Z ← Z + X;
    X ← X + 1}
```

3. Предположим, что процедура сортировки методом вставки, представленная на рис 4.11, была применена к списку George, Cheryl, Alice и Bob. Опишите, как будет выглядеть список каждый раз по окончании выполнения тела внешней структуры **while**.
4. Почему не следует заменять фразу "`по алфавиту размещается ниже`" в инструкции **while** на рис. 4.11 на фразу "`по алфавиту размещается ниже или эквивалентно`"?
5. Вариантом алгоритма сортировки методом вставки является *выборочная сортировка* (selection sort). Каждый раз 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 вначале, 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. 4.11.
6. Другой известный алгоритм сортировки – *сортировка методом пузырька* (bubble sort). 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. 4.11.

4.5. РЕКУРСИВНЫЕ СТРУКТУРЫ

Рекурсивные структуры являются другим видом повторяющихся структур. Цикл означает повторное выполнение набора команд, при этом весь набор команд полностью выполняется, а затем повторяется. Рекурсия же предполагает повторное выполнение набора команд как подзадачу самой себя. Чтобы ввести понятие рекурсии, рассмотрим алгоритм *двоичного поиска* (binary search), в котором в процессе поиска применяется метод разбиения.

Поиск и сортировка. Алгоритмы последовательного и двоичного поиска – это всего лишь два представителя из большого семейства алгоритмов, осуществляющих поисковый процесс. (Использование для этой цели индексов и механизм перемешивания будет рассматриваться в главе 8.) Аналогично, сортировка методом вставки – это лишь один из многих существующих алгоритмов сортировки. Другими классическими алгоритмами являются сортировка слиянием (обсуждается в главе 11), выборочная сортировка (ее описание можно найти в подразделе "Вопросы для самопроверки", п. 5 раздела 4.4), сортировка методом пузырька (см. подраздел "Вопросы для самопроверки", п. 6, раздела 4.4), 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. Третий том книги Дональда Кнута "Искусство программирования", хотя и сложен для восприятия начинающими, в целом может считаться последним словом в области методик поиска и сортировки. В своем многотомном труде (который со временем может составить семь томов) Кнут собрал невероятное количество информации, относящейся к фундаментальным алгоритмам вычислений, и тем самым внес значительный вклад в библиотеки специалистов в области компьютерных наук и обработки данных.

Алгоритм двоичного поиска. 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 имени в телефонном справочнике. Человек никогда не просматривает справочник последовательно, элемент за элементом или даже страница за страницей. Мы просто открываем его примерно в том месте, где, как мы думаем, может находиться нужное имя. Если повезет, оно окажется именно там, в противном случае поиск придется продолжить. Однако в этой точке мы уже сузим область поиска либо до начальной части справочника, предшествующей нашей текущей позиции, либо до остальной части справочника, следующей за ней.

На рис. 4.12 этот подход, применяемый к произвольному отсортированному списку, описан с помощью псевдокода. В данном случае мы не знаем примерного места расположения элементов, поэтому инструкции на рисунке предписывают начинать работу с открытия списка на "среднем" элементе. Слово "средний" заключено в кавычки, поскольку вполне возможно, что число элементов в списке будет четным, и тогда среднего элемента в строгом смысле этого слова просто не существует. В этом случае условимся, что средним считается первый элемент второй половины списка.

```
    выбрать "средний" элемент как ПроверяемоеЗначение;
    выполнить набор команд, соответствующий одному из
случаев:
    Случай 1: ИскомоеЗначение = ПроверяемоеЗначение
       {сообщить об успехе}
    Случай 2: ИскомоеЗначение < ПроверяемоеЗначение
       {применить процедуру Поиск для обнаружения
        ИскомоеЗначение в верхней части Списка, и
        сообщить о результате этого поиска}
    Случай 3: ИскомоеЗначение > ПроверяемоеЗначение
       {применить процедуру Поиск для обнаружения
        ИскомоеЗначение в нижней части Списка, и
        сообщить о результате этого поиска}
```

Рис. 4.12. Принцип двоичного поиска

Если выбранный элемент не является искомым, то программа, приведенная на рис. 4.12, предлагает два варианта дальнейших действий (поиск или в начальной или конечной половине списка). В каждом из них предусматривается выполнения вторичного поиска, осуществляемого процедурой с именем `Search`. Следовательно, чтобы завершить нашу программу, необходимо создать подобную процедуру, описывающую, как осуществляется этот вторичный поиск. Заметим, что эта процедура должна быть в состоянии удовлетворить запрос на поиск в пустом списке. Например, если показанной на рис. 4.12 программе будет передан список, состоящий из одного элемента, который не является искомым, то процедура `Search` будет вызвана для поиска либо в подсписке, находящемся выше этого единственного значения, либо в подсписке, находящемся ниже его, однако оба эти списка пусты.

В качестве искомой процедуры `Search` можно было бы использовать процедуру последовательного поиска, созданную нами в предыдущем разделе, но это совсем не тот метод, который выбрал бы человек при поиске в телефонном справочнике. Вероятно, он просто применил бы к оставшейся части справочника ту же процедуру, которой только что воспользовался для всего справочника в целом. Другими словами, выбрал бы какой-то элемент, достаточно близкий к середине оставшейся части спи-

ска, и использовал его для дальнейшего сужения области поиска.

Подобный подход к процедуре поиска представлен на рис. 4.13. Здесь демонстрируется способ решения задачи, заключающейся в определении присутствия имени John в списке, приведенном в левой части рисунка. Прежде всего, анализируется средний элемент Harry. Так как имя, которое мы ищем, по алфавиту располагается после данного имени, поиск продолжается в нижней части исходного списка. Средним элементом этой части является имя Larry. Поскольку по алфавиту искомое имя предшествует данному, поиск следует продолжать в верхней части текущего подсписка. Обратившись к среднему элементу этого вторичного подсписка, обнаруживаем искомое имя John и объявляем поиск успешным. Короче говоря, наша стратегия состоит в последовательном делении анализируемого списка на меньшие по размеру сегменты до тех пор, пока либо будет найдено искомое значение, либо поиск сузится до пустого сегмента.

Можно реализовать эту стратегию с помощью нашего псевдокода, модифицировав приведенную на рис. 4.12 программу так, чтобы учесть возможность получения пустого списка. Модифицированная соответствующим образом программа на псевдокоде, которой присвоено имя `Search`, показана на рис. 4.14. При выполнении этой процедуры и при достижении инструкции "`Применить процедуру Search, чтобы ...`", мы будем просто применять этот же метод поиска к меньшему списку, который является частью исходного списка. Если этот вторичный поиск завершится успешно, мы вернемся в исходную процедуру, чтобы объявить выполняемый в ней поиск успешным. Если же вторичный поиск окончится неудачей, мы объявим неудачным и исходный поиск.

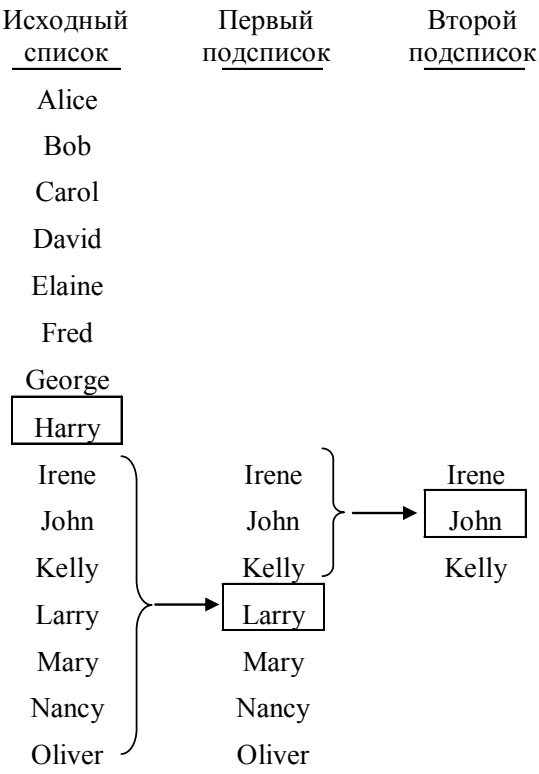


Рис. 4.13. Применение стратегии двоичного поиска для обнаружения имени John в упорядоченном списке

Чтобы увидеть, как представленная на рис. 4.14 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 `Search` к списку элементов, предшествующих имени David, т.е. к списку Alice, Bill, Carol. Для этого нам потребуется создать

```
    Procedure Поиск (Список, ИскомоеЗначение)
    if (Список пустой)
        then {сообщить о неудаче}
        else {выбрать "средний" элемент как Про-
веряемоеЗначение;выполнить набор команд, соответ-
ствующий одному из случаев:
            Случай 1: ИскомоеЗначение = Проверяе-
моеЗначение {сообщить об успехе}
            Случай 2: ИскомоеЗначение < Проверяе-
моеЗначение {применить процедуру Поиск для обна-
ружения ИскомоеЗначение в верхней части Списка, и
сообщить о результате этого поиска}
            Случай 3: ИскомоеЗначение > Проверяе-
моеЗначение {применить процедуру Поиск для обна-
ружения ИскомоеЗначение в нижней части Списка, и
сообщить о результате этого поиска}
          }
```

Рис. 4.14. Алгоритм двоичного поиска, записанный на псевдокоде

вторую копию процедуры `Search`, предназначенную для решения данной промежуточной задачи.

Теперь мы имеем две выполняющиеся копии нашей процедуры поиска, как показано на рис. 4.15. Дальнейшее выполнение исходной копии процедуры временно приостановлено на следующей инструкции:

`Применить процедуру Search, чтобы определить, есть ли в части списка, предшествующей элементу <проверяемый_элемент>, элемент <искомое_значение>`

Вторая копия процедуры используется для поиска имени Bill в списке Alice, Bill, Carol. Завершив вторую процедуру двоичного поиска, мы аннулируем ее копию и сообщим 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.

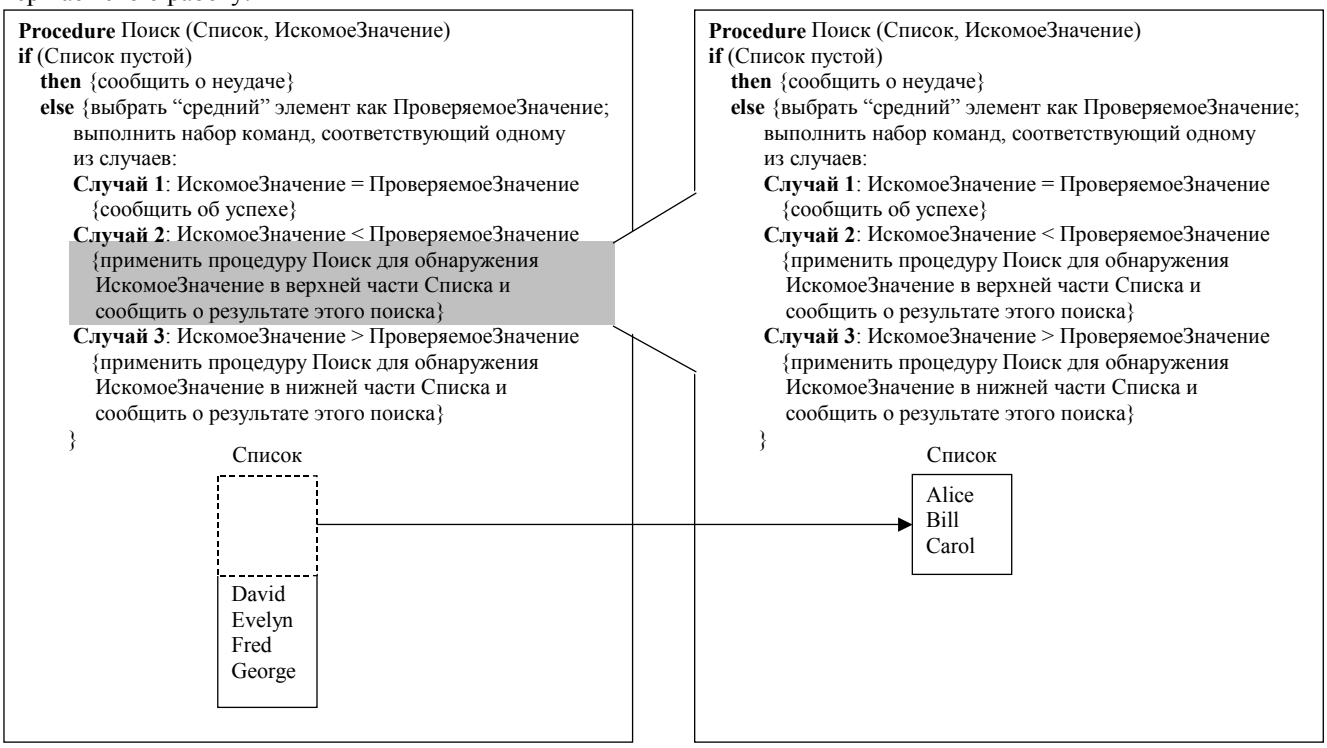


Рис. 4.15. Вызов второй копии процедуры из ее исходной копии при поиске записи Bill

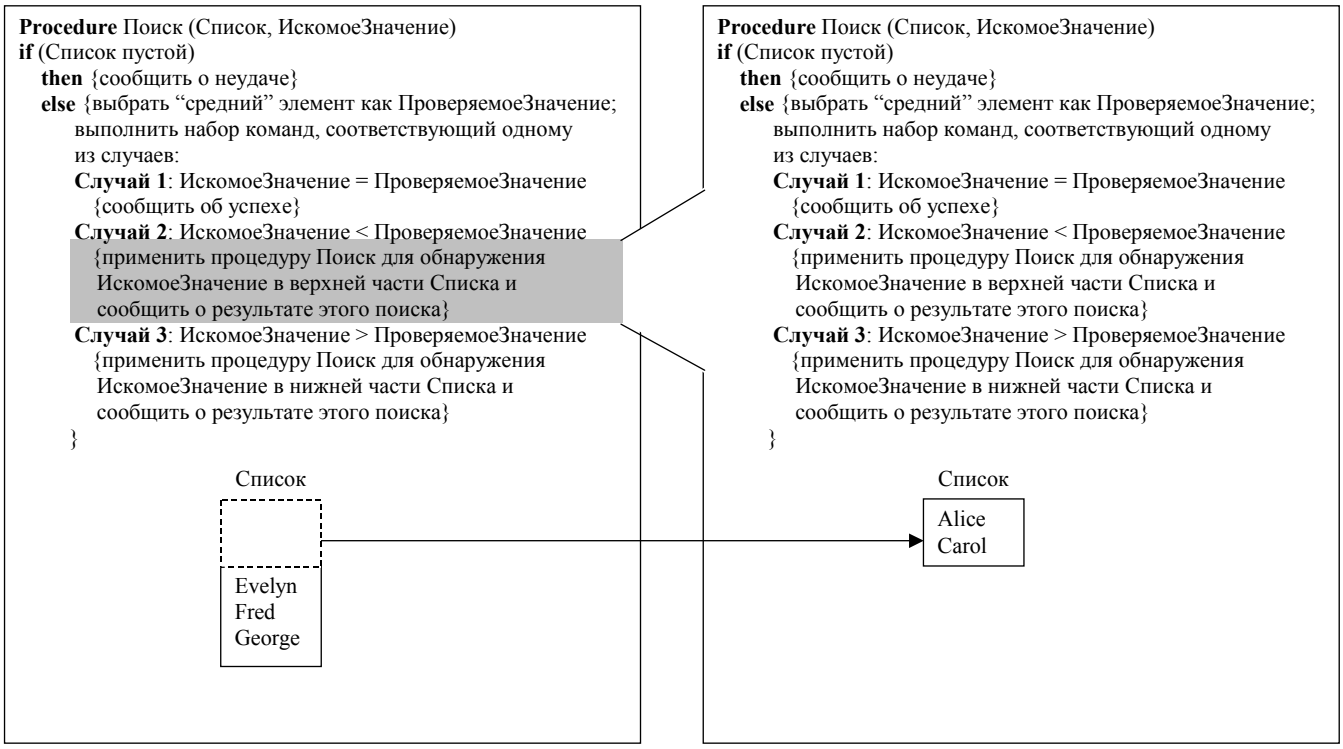


Рис. 4.16. Вызов второй копии процедуры из ее исходной копии при поиске записи David

На этом этапе мы завершили дополнительный поиск, как предписывалось исходной процедурой, поэтому можно продолжить выполнение этой исходной копии, т.е. объявить результат дополнительного поиска результатом первоначального поиска. В итоге 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. 4.14 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. 4.16.

Вторая копия процедуры выберет в качестве проверяемого элемента имя Carol и определит, что искомое значение должно находиться после него. Процедура вызовет третью копию процедуры "Поиск" 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. 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 качеству проверяемого элемент Carol; а третья начинает поиск в пустом списке.

Конечно же, 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 неуспешным, 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.

Если еще раз просмотреть приведенные выше примеры, можно увидеть, что в процессе, осуществляемом представленным на рис. 4.14 алгоритмом, необходимо 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 отличается. 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* (recursion).

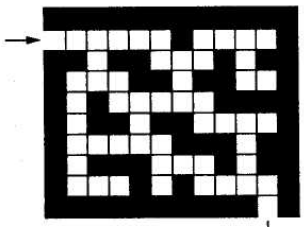
При выполнении рекурсивного алгоритма создается иллюзия существования множества его копий, называемых *активациями* (activation), которые появляются и исчезают по мере выполнения алгоритма. Из всех активаций, существующих в заданный момент времени, активно функционирует только одна. Все остальные фактически остановлены, поскольку каждая из них ожидает, пока завершится следующая, запущенная ею активация, и только после этого она сможет продолжить 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.

Разработка рекурсивных процедур. При создании рекурсивных процедур основным условием является выбор способа, с помощью которого исходную задачу можно разделить на меньшие задачи того же типа, а также определить, как результаты решения этих меньших задач могут быть использованы в качестве абстрактных инструментов нахождения решения исходной задачи. Давайте применим этот подход к поиску выхода из лабиринта, подобного приведенному на этом рисунке. Будем продвигаться вперед до тех пор, пока не подойдем к первому разветвлению. В этой точке мы будем считать, что каждый вариант дальнейшего движения представляет собой вход в новый лабиринт меньшего размера. Если в любом из этих меньших лабиринтов выход будет найден, наша задача решена. Такой способ рассуждений приводит к построению следующей процедуры:

```
procedure FindExit (<лабиринт>)
Продвигаться по лабиринту до развилки, тупика или выхода; В каждом из указанных случаев выполнить соответствующие инструкции из числа приведенных ниже:
case 1: достигнут выход: (сообщить "выход найден")
case 2: достигнут тупик: (сообщить "неудача")
case 3: достигнута развилка:
   (while (есть ветвь, ведущая к неисследованной территории,
        и выход еще не найден) do
          (применить процедуру FindExit к лабиринту, представленному
           одной из неисследованных ветвей;
          if (эта процедура сообщила, что выход найден)
           then (сообщить "выход найден".)
          )
    if (все варианты этой развилки неудачны)
    then (сообщить "неудача")
   )
```



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. Это означает, что одной из остановленных активаций разрешается продолжить свое выполнение, завершить ее задачу и, в свою очередь, позволить следующей остановленной активации возобновить свои действия. Таким образом, все порожденные активации в конечном счете будут завершены, что приведет к завершению исходной задачи.

Теперь посмотрим, как операции инициализации и модификации механизма управления повторением реализованы в рекурсивной программе двоичного поиска (рис. 4.14). 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.

Рассматривая структуры управления рекурсией и итерациями, можно попробовать установить, одинаковы ли их возможности. То есть, если некоторый алгоритм разработан с использованием циклической структуры, то 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 11, где будут рассматриваться некоторые теоретические аспекты компьютерных наук и их математических основ. Опираясь на сделанные в этой главе выводы, в приложении Д будет доказана эквивалентность итеративных и рекурсивных структур.

Вопросы для самопроверки

1. Какие имена будут проверены программой двоичного поиска (рис. 4.14) 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? В списке из 100 000 элементов?

4.6. ЭФФЕКТИВНОСТЬ И ПРАВИЛЬНОСТЬ

В этом разделе мы обсудим понятия, которые являются частью данного формального введения в теорию алгоритмов. О

них всегда нужно помнить при самостоятельной разработке алгоритмов. Первое из них – эффективность, а второе – правильность.

Эффективность алгоритма. Хотя современные машины способны выполнять миллионы операций в секунду, эффективность по-прежнему остается важнейшим аспектом разработки алгоритмов. Зачастую выбор между эффективным и неэффективным решением задачи может на самом деле означать выбор между реализуемым и нереализуемым способом ее решения.

Рассмотрим задачу, с которой сталкивается секретарь университета при поиске личных дел студентов и их заполнении. Хотя в университете на протяжении любого семестра фактически числится около 10 000 студентов, секретарю в действительности приходится иметь дело с более чем 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 отдельных случаях поиск потребует меньшего числа операций, а в других – большего. 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 выполняется за десять миллисекунд (десять тысячных долей секунды), то среднее время поиска будет составлять 150 с 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 с – 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, самое большее, 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 с, 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[2].

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. В нашем случае мы проанализировали время, требующееся алгоритмам последовательного и двоичного поиска, что позволило нам определить, какой из них больше подходит в данном конкретном случае. В общем случае такой анализ осуществляется в более широком контексте. Это означает, что при рассмотрении алгоритмов, выполняющих поиск в списке, мы не ограничимся списком фиксированной длины, но пытаемся вывести формулу эффективности алгоритма для списков произвольной длины. Такой анализ включает изучение ситуаций, в которых алгоритм демонстрирует свои наилучшие свойства, ситуаций, когда его эффективность минимальна, а также оценку его средней производительности.

В предыдущем случае выполненный нами анализ заключался в оценке средней производительности алгоритма последовательного поиска, а также определении той производительности, которую алгоритм двоичного поиска продемонстрирует в наихудшем случае. Хотя мы рассматривали список определенной длины, несложно обобщить наши 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.)

Давайте попробуем проанализировать аналогичным образом алгоритм сортировки методом вставки. Поскольку основным действием в реализации данного алгоритма является сравнение двух имен, наш подход будет состоять в подсчете количества таких сравнений, которые потребуется выполнить при сортировке списка длиной n элементов.

Вспомним, что при сортировке методом вставки осуществляется выбор некоторого элемента, называемого опорным; после этого он сравнивается с предшествующими ему элементами, пока для него не будет найдено надлежащее место, после чего опорный элемент вставляется в соответствующую позицию. Алгоритм начинается с выбора второго элемента списка в качестве опорного. По мере его выполнения в качестве опорных выбираются следующие элементы – пока не будет достиг-

---

[2] Чтобы воспользоваться преимуществами алгоритма двоичного поиска, личные дела студентов должны располагаться в памяти машины таким образом, чтобы можно было извлекать средние записи последовательно уменьшающихся подсписков без чрезмерных усилий. Это можно осуществить, запоминая личные дела в индексированных файлах-структурах, которые будут обсуждаться в главе 8. Кроме того, тех же результатов можно достичь и с использованием хешированных файловых структур, которые также описываются в главе 8.

нут 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 потребуется сравнивать со всеми впереди стоящими 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. 4.17). Следовательно, общее количество сравнений при сортировке списка из n элементов составит $1+2+3+...+(n-1)$, что эквивалентно $n(n-1)/2$ или $1/2(n^2-n)$. В частности, для списка из 10 элементов алгоритму сортировки методом вставки в наихудшем случае потребуется выполнить 45 сравнений.

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 длины 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. 4.18. Он показывает, как будет возрастатьвремя, необходимое для выполнения сортировки методом вставки, при увеличении длины сортируемого списка.

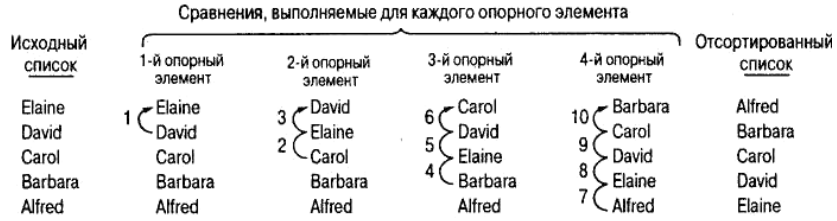


Рис. 4.17. Работа алгоритма сортировки методом вставки в наихудшем случае

Данный график построен по оценкам работы алгоритма в наихудшем случае, когда, исходя из результатов наших исследований, для списка длиной n требуется выполнить не менее $1/2(n^2-n)$ сравнений элементов. 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. 4.18. График зависимости времени сортировки от длины списка при сортировке методом вставки

Выполним аналогичный анализ работы алгоритма двоичного поиска в наихудшем случае. 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. 4.19 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 отличием между графиками, представленными на рис. 4.18 и 4.19, безусловно, является их общая форма. Именно форма графика, а не его индивидуальные особенности, демонстрирует, насколько хорошо данный алгоритм будет справляться с все возрастающими объемами данных. Заметим, что общая форма графика определяется типом отображаемого 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. 4.19. Общую форму кривой принято определять простейшим выражением, порождающим кривую данной формы. В частности, параболическая форма обычно определяется выражением n^2, а логарифмическая – выражением lg$n$.

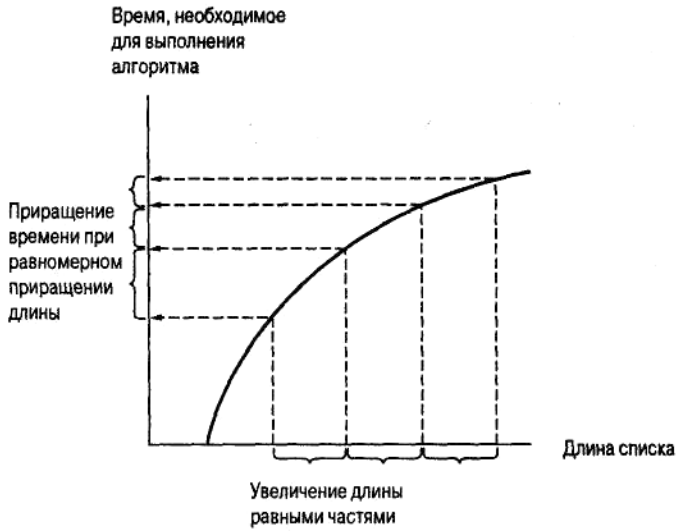


Рис. 4.19. График зависимости времени поиска от длины списка для алгоритма двоичного поиска

Выше было показано, что форма графика, представляющего зависимость времени выполнения алгоритма от объема входных данных, отражает общие характеристики эффективности алгоритма. Поэтому принято классифицировать алгоритмы согласно форме их графиков, построенных для самого неблагоприятного случая. Способ обозначения, используемый для определения этих классов, иногда называют тета-классами. Алгоритмы, графики которых имеют параболическую форму (например, сортировка методом вставки), относятся к классу $\Theta(n^2)$, алгоритмы, графики которых имеют логарифмическую форму (например, двоичный поиск), – к классу $\Theta(\lg n)$. Любой алгоритм из тета-класса $\Theta(\lg n)$ по самой свой сути всегда более эффективен, чем алгоритм из тета-класса $\Theta(n^2)$.

Верификация программ. Вспомним, что четвертая фаза процедуры решения задачи, согласно схеме, предложенной математиком Полиа (см. раздел 4.3), заключается в *оценке точности работы программы* или *верификации* (verification) 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, должен остановиться в уединенном отеле 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?

Первым делом уясним себе, 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. 4.20). Более того, любое меньшее число разрезов оставит два звена соединенными, поэтому мы заключаем, что правильный ответ для этой задачи – три звена.

Однако, рассмотрев задачу более внимательно, можно заметить, что если разрезать только третье звено, то получится три фрагмента цепочки, состоящие из одного, двух и четырех звеньев (рис. 4.21). 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.

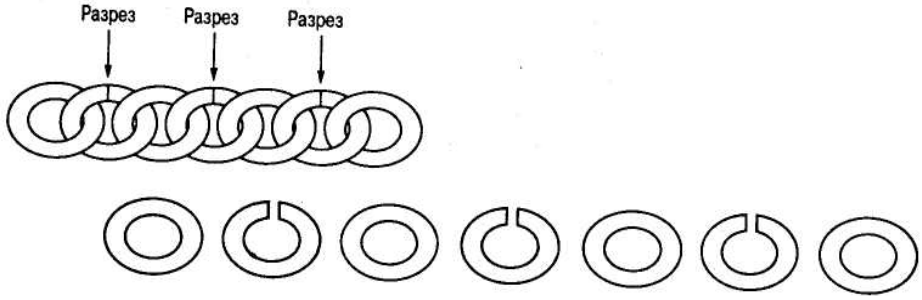


Рис. 4.20. Разделение всех звеньев цепочки с помощью лишь трех разрезов

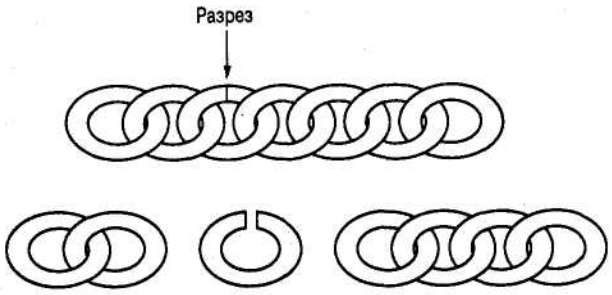


Рис. 4.21. Решение задачи с помощью лишь одного разреза

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.

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, то 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* (precondition), 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. Например, пусть определенное утверждение о значении переменной Y перед выполнением приведенной ниже инструкции было истинно:

```
X ← Y
```

После выполнения этой инструкции то же утверждение 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.

Несколько более сложным случаем является выполнение оператора **if-then-else**, например, следующего вида:

```
if (условие) then {инструкция 1} else {инструкция 2}.
```

Если в этом примере известно, что некоторое утверждение было истинно перед выполнением данной структуры, то непосредственно перед выполнением инструкции 1 мы знаем, что истинны как это утверждение, так и проверяемое условие. В то же время, если должна выполняться инструкция 2, мы знаем, что должны быть истинны исходное утверждение и отрицание проверяемого условия.

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 1, созданной в Гарвардском университете в 1940 г., монтажные ошибки в которой оставались необнаруженными в течение многих лет. Более "свежий" пример – ошибки в блоке выполнения операций с плавающей точкой, имевшие место в первых микропроцессорах типа Pentium. В обоих случаях существующие ошибки были выявлены до каких-либо серьезных последствий.

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, где 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-do`**, представленную на рис. 4.22. Предположим, как следствие предусловий, заданных в точке А, мы можем установить, что определенное утверждение истинно при каждой проверке условия окончания цикла (точка В) на протяжении всего процесса повторения. Такое утверждение внутри цикла называется *инвариантом цикла* (loop invariant). Как только повторение завершается, выполнение переходит к точке С, где 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 бы просто не завершился.) 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.

Этот метод анализа можно применить к приведенному выше алгоритму сортировки методом вставки (см. рис. 4.11). Внешний цикл в этой программе основан на следующем инварианте цикла:

Каждый раз при выполнении проверки условия окончания цикла имена, предшествующие N-му элементу, образуют отсортированный список.

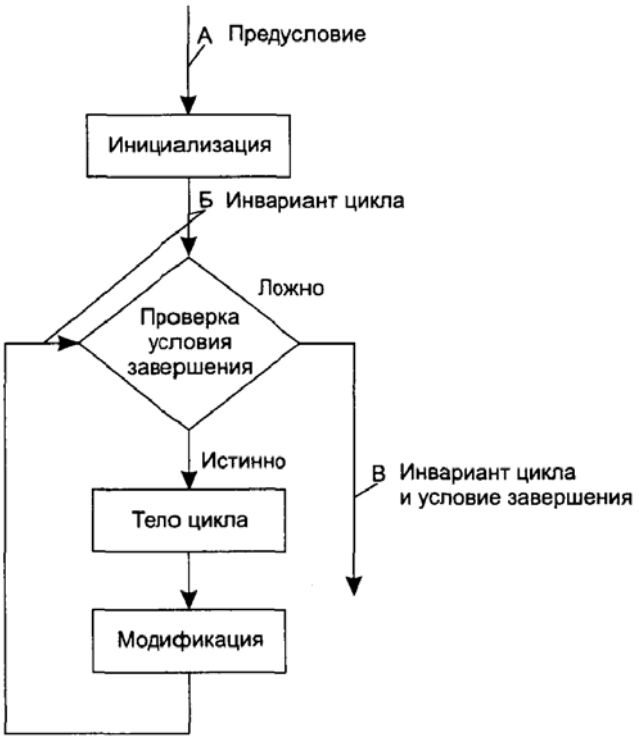


Рис. 4.22. Использование формального утверждения для проверки правильности оператора цикла `while-do`

Условие окончания этого цикла формулируется следующим образом:

```
Значение N больше, чем длина списка.
```

Таким образом, как только цикл завершится, мы будем знать, что оба условия должны быть выполнены, а это означает, что весь список будет отсортирован.

К сожалению, методы формальной верификации программ не настолько хорошо разработаны, чтобы их можно было использовать в приложениях общего типа. Сегодня в большинстве случаев "верификация" программного обеспечения производится посредством его тестирования при различных исходных условиях, что весьма ненадежно. 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 случай произошел в компании AT&T. Ошибка в программном обеспечении, управляющем 114 телефонными станциями, оставалась незамеченной с момента его установки в декабре 1989 г. и вплоть до 15 января 1990 г., когда исключительное стечение обстоятельств привело к тому, что за девять часов было беспричинно заблокировано примерно 5 миллионов телефонных звонков.

Вопросы для самопроверки

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 списков из 1000 и 10 000 элементов?

2. Приведите примеры алгоритмов, относящихся к каждому из следующих классов: $\Theta(\lg n)$, $\Theta(n)$, $\Theta(n^2)$.

3. Перечислите следующие классы в порядке убывания их эффективности: $\Theta(n^2)$, $\Theta(\lg n)$, $\Theta(n)$, $\Theta(n^3)$.

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 это.

```
Произведение ← Y;
Счетчик ← 1;
while (Счетчик < X) do
   {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 + ... + N.

```
Сумма ← 0;
I ← 0;
while (I < N) do
   {I ← I + 1;
    Сумма ← Сумма + I}
```

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?

Упражнения

1. Приведите примеры последовательности этапов, удовлетворяющей неформальному определению алгоритма, приведенному в начале раздела 4.1, но не удовлетворяющей определению, приведенному на рис. 4.1.

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. Представляет ли следующая программа алгоритм в строгом смысле этого слова? Поясните свой ответ.

```
Счетчик ← 0;
while (Счетчик ≠ 5) do
   {Счетчик ← Счетчик + 2}
```

5. По какой причине приведенная ниже последовательность этапов не образует алгоритм?

Этап 1. Провести отрезок прямой линии, соединяющий точки с координатами (2, 5) и (6, 11).

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.

6. Перепишите следующий сегмент программы, используя структуру **`repeat-until`** вместо **`while-do`**. Убедитесь, что новая версия программы печатает те же значения, что и исходная.

```
Счетчик ← 2;
while (Счетчик < 7) do
   {напечатать значение переменной Счетчик;
    Счетчик ← Счетчик +1}
```

7. Перепишите следующий фрагмент программы, используя структуру **while-do** вместо **repeat-until**. Убедитесь, что новая версия печатает те же значения, что и исходная.

```
Счетчик ← 1;
repeat {напечатать значение переменной Счетчик;
        Счетчик ← Счетчик + 1}
until (Счетчик = 5)
```

8. 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.

9. В чем отличие формального языка программирования от псевдокода?

10. В чем отличие между синтаксисом и семантикой?

11. Четыре шахтера, которые имеют один фонарь, должны пройти через шахту. Одновременно по шахте могут двигаться не больше двух человек, и каждый шахтер, двигаясь в шахте, должен иметь фонарь. Шахтеры, имена которых Эндрю, Блэйк, Джонсон и Келли, могут пройти шахту за одну, две, три и четыре минуты, 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.

12. Допустим, у нас есть большой и маленький стаканчики для вина. Сначала наполним вином маленький стаканчик и перельем его в большой стакан. Затем наполним водой маленький стакан, 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 в маленьком стакане – воды в вине или вина в воде? После того как вы решите задачу, объясните ход ваших рассуждений.

13. 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.

14. Разработайте алгоритм, который получает на входе две строки символов и проверяет, является ли первая строка частью второй.

15. 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?

```
Последнее ← 0;
Текущее ← 1;
while (Текущее < 100) do
   {напечатать значение переменной Текущее;
    Временное ← Последнее;
    Последнее ← Текущее;
    Текущее ← Последнее + Временное}
```

16. Какую последовательность чисел напечатает следующий алгоритм, если на входе задать значения 0 и 1?

```
procedure MysteryWrite (Последний, Текущий)
if (Текущий < 100) then
   {напечатать значение переменной Текущий;
    Временный ← Текущий + Последний;
    применить процедуру MysteryWrite к значениям Текущий и Временный}
```

17. Преобразуйте процедуру `MysteryWrite` из предыдущего задания так, чтобы она печатала числа в обратном порядке.

18. Какие буквы будут проверяться, если применить двоичный поиск (см. рис. 4.13) для поиска значения J в списке A, B, C, D, E, F, G, H, I, J, K, L, M, N, O? A в случае поиска значения Z?

19. 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?

20. Определите тело цикла в следующей структуре и подсчитайте, сколько раз оно будет выполнено. Что произойдет, если проверяемое условие заменить на выражение "**while** `(Счетчик ≠ 6)`"?

```
Счетчик ← 1;
while (Счетчик ≠ 7) do
      {напечатать значение переменной Счетчик;
       Счетчик ← Счетчик + 3}
```

21. Какие проблемы могут возникнуть при реализации на компьютере следующей программы? (Подсказка. Вспомните об ошибках округления при выполнении арифметических операций с плавающей точкой.)

```
Счетчик ← 0.1;
repeat {напечатать значение переменной Счетчик;
        Счетчик ← Счетчик + 0.1}
```

```
until (Счетчик = 1)
```

22. Разработайте рекурсивную версию алгоритма Евклида (вопрос 3 к разделу 4.2).

23. Предположим, что на вход процедур `Test1` и `Test2`, приведенных ниже, передано значение 1. Чем будут отличаться напечатанные этими процедурами результаты?

```
procedure Test1 (Счетчик)
if (Счетчик ≠ 5)
    then {напечатать значение параметра Счетчик;
          выполнить процедуру Test1(Счетчик + 1)}
procedure Test2 (Счетчик)
if (Счетчик ≠ 5)
   then {выполнить процедуру Test2(Счетчик + 1);
         напечатать значение параметра Счетчик}
```

24. Определите основные составляющие механизма управления в предыдущей задаче. В частности, какое условие вызывает окончание процесса? Где происходит модификация состояния процесса, приближающая его к условию завершения? Где инициализируется состояние управляющего процесса?

25. Разработайте алгоритм генерации последовательности целых положительных чисел (в возрастающем порядке), 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?

26. 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?

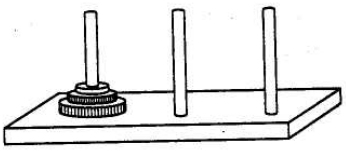
д) Сколько элементов будет рассмотрено при поиске имени Elaine с использованием метода последовательного поиска? Сколько элементов будет рассмотрено при поиске этого имени с использованием метода двоичного поиска?

27. 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 запись *n*!. Таким образом, факториал числа 3 (обозначается как 3!) – это 3!= 3*(2!)= 3*(2*(1!))= 3*(2*(1*(0!))) = 3*(2*(1*(1))) = 6. Разработайте рекурсивный алгоритм вычисления факториала заданного числа.

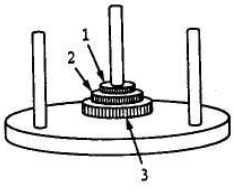
28. а) Предположим, что вам необходимо отсортировать список из пяти имен и что вы уже разработали раньше алгоритм для сортировки списка из четырех имен. Разработайте алгоритм сортировки списка из пяти имен, использующий ранее разработанный алгоритм.

б) Разработайте рекурсивный алгоритм сортировки списка произвольной длины, основанный на методе, используемом для решения предыдущей задачи.

29. Головоломка "Ханойская башня" состоит из трех вертикальных стержней, на один из которых надето несколько колец последовательно уменьшающихся (в направлении снизу вверх) размеров. 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.



30. Еще один подход к решению головоломки "Ханойская башня" (упр. 29) 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 т.д., начиная с верхнего. Кольца с нечетными номерами, находящиеся сверху 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".



31. 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?

32. 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?

33. Разработайте алгоритм, который печатает все возможные варианты перестановки символов в строке из пяти различных символов.

34. Разработайте алгоритм, который в заданном списке имен находит самое длинное имя. 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?

35. Разработайте алгоритм, который в заданном списке из пяти или более чисел находит пять наименьших и пять наибольших чисел, не сортируя полностью весь список.

36. 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. 4.11).

37. Какое максимальное количество элементов может быть проверено при применении алгоритма двоичного поиска (рис. 4.13) к списку, содержащему 4000 строк? Как оно соотносится с аналогичным значением для метода последовательного поиска (рис. 4.6)?

38. 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* и сколько отдельных операций умножения потребуется для их перемножения.

39. 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.

40. 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, 1673

41. 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 ← 0;
Y ← 1/2;
while (X ≠ 1) do
      {X ← X+Y;
       Y ← Y-2}
```

42. Следующий фрагмент программы разработан для вычисления произведения двух неотрицательных целых чисел X и Y путем вычисления суммы X копий числа Y. 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)
```

43. Следующий фрагмент программы составлен для определения, какое из двух целых чисел X и Y является большим. Является ли этот фрагмент правильным? Поясните свой ответ.

```
Разность ← X—Y;
if (Разность положительна)
    then {напечатать "X больше Y"}
    else {напечатать "Y больше X"}
```

44. Следующий фрагмент программы должен находить наибольший элемент в непустом списке целых чисел. Правильно ли он составлен? Поясните свой ответ.

```
Значение ← значение первого элемента списка;
Текущий ← значение первого элемента списка;
while (Текущий ≠ последнему элементу списка) do
   {if (Текущий > Значение)
        then {Значение ← Текущий}
   Текущий ← значение следующего элемента списка}
```

45. а) Определите предусловия для алгоритма последовательного поиска. 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** действительно завершается.

46. 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) do
```

```
      {Z ← Z-1;
       J ← J+1}
```

Ответы на вопросы для самопроверки

Раздел 4.1

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, исполнения фокусов, а также алгоритм Евклида. Многие из "алгоритмов", с которыми мы сталкиваемся в обыденной жизни, не соответствуют его формальному определению. В тексте был приведен пример алгоритма деления столбиком. 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. Однако в действительности процесс рано или поздно достигнет ситуации, когда в кармане больше не останется ни одной монеты. Второй момент состоит в том, что фактически в этой ситуации алгоритм может даже начаться. С данной точки зрения задача является неоднозначной. Представленный алгоритм не "говорит" нам, как поступить в этой ситуации.

Раздел 4.2

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 процедура 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 не нуль) do
   {Remainder ← остаток от деления X на Y;
    X ← Y;
    Y ← Remainder}
GCD ← X
```

4. Все цвета можно получить в результате сочетания красного, синего и зеленого цветов, поэтому телевизионные электронно-лучевые трубки разрабатываются так, чтобы генерировать именно эти три основных цвета.

Раздел 4.3

1. а)

```
if (n = 1 или n = 2)
    then {Ответ – это список из одно числа n}
    else {Разделить n на 3, получив частное q и остаток r
          if (r = 0)
            then {Ответ – это список из q троек}
          if (r = 1)
              then {Ответ – это список из q-1 троек и 2 двойки}
          if (r = 2)
              then {Ответ – это список из q троек и 1 двойки}
          }
```

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 $2n - 1$ квадратов, и каждая фишка покрывает точно три квадрата.

в) Части а и б этого вопроса представляют собой замечательный пример того, как знание решения одной проблемы позволяет решить другую. См. четвертую фазу Полиа.

3. Он говорит: "Это правильный ответ".

Раздел 4.4

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.

Cheryl	Alice	Alice
George	Cheryl	Bob
Alice	George	Cheryl
Bob	Bob	George

4. Настаивать на том, чтобы предшествующий элемент помещался на то же место в списке, бессмысленно. Например, сделайте предложенные изменения, а затем примените новую программу к списку, все элементы которого одинаковы.

5.

```
procedure sort(Список)
N ← 1;
while (N < длины Списка) do
   {J ← N+1;
    while (J ≤ длины Списка) do
       {if (элемент в позиции J < элемент в позиции N)
        then {поменять местами эти два элемента)
             J ← J+1}
    N ← N+1}
```

6. Приведенное ниже решение является неэффективным. Можете ли вы сделать его более эффективным?

```
procedure sort(Список)
N ← длина Списка;
while (N > 1) do
   {J ← длина Списка;
    while (J > 1) do
       {if (элемент в позиции J < элемента в позиции J-1)
         then {поменять местами эти два элемента}
        J ← J-1}
    N ← N-1}
```

Раздел 4.5

1. Первый подсписок состоит из имен, следующих за именем Henry, т.е. Irene, Joe, Karl, Larry, Mary, Nancy и Oliver. Далее идут имена из этого списка, предшествующие имени Larry, т.е. Irene, Joe и Karl. Теперь в очередном цикле поиска искомое имя Joe будет найдено в середине рассматриваемого подсписка.

2. 8, 17

3.

Alice	Alice
Carol	Bob
Bob	Carol
Larry	Larry
John	John

В итоге будет выполнено четыре вызова процедуры.

4. В результате выполнения процесса список будет отсортирован. Однако его выполнение будет сопровождаться излишней потерей времени, поскольку при первом вызове процедуры первый элемент списка сначала удаляется, а потом возвращается на прежнее место.

5. Будет выполнено несколько вызовов процедуры. При каждом из них элемент будет просто удаляться из списка, а затем возвращаться на прежнее место.

Раздел 4.6

1. Если машина может отсортировать список из 100 имен за секунду, то она способна выполнять $^1/_4$ (10 000 – 100) сравнений в секунду. Это означает, что каждое сравнение выполняется приблизительно за 0,0004 с. Следовательно, сортировка списка из 1000 имен (которая в среднем потребует выполнения $^1/_4$ (10 000 000 – 1000) сравнений) займет около 100 с, или $1^2/_3$ мин.

2. Алгоритм бинарного поиска принадлежит к классу Θ (lg$n$), алгоритм последовательного поиска – к классу Θ (n), а алгоритм сортировки вставками – к классу Θ (n^2).

3. Класс Θ (lg$n$) содержит наиболее эффективные алгоритмы, за которыми следуют алгоритмы классов Θ (n), Θ (n^2) и Θ (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 + ... + 1 и I меньше или равно N" является истинным. Объединяя его с условием прекращения суммирования "I больше или равно N", мы получим желаемый вывод "Sum = 1 + 2 + ... +N". Поскольку переменная I 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.

5. ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Разработка сложных систем программного обеспечения, таких, как операционные системы, программное обеспечение для сетей и огромное количество прикладных программ, доступных сегодня, была бы невозможна, если бы люди были вынуждены представлять алгоритмы непосредственно на машинном языке. Сказать, что работать с такими языками при создании сложной системы было бы чрезвычайно трудно, значит ничего не сказать. Следовательно, языки программирования, подобные нашему псевдокоду, были разработаны так, чтобы они позволяли записать алгоритм в форме, понятной человеку и легко преобразуемой в команды машинного языка. Языки программирования позволяют избежать лабиринта регистров, адресов памяти и машинных циклов в процессе разработки программы и сосредоточиться на решаемой задаче. В этой главе мы и познакомимся с языками программирования.

5.1. ЭВОЛЮЦИЯ И КЛАССИФИКАЦИЯ

Начнем обсуждение языков программирования с рассмотрения их истории и существующих в настоящий момент основных подходов к программированию.

Ранние поколения. Первоначально процесс программирования предусматривал запись программистом всех алгоритмов непосредственно на машинном языке. Такой подход усугублял и без того трудную задачу разработки алгоритмов и слишком часто приводил к ошибкам, которые необходимо было обнаружить и исправить (процесс, известный как отладка) до того, как работу можно было считать законченной.

Первым шагом на пути к облегчению 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 при разработке программ стали широко применять мнемоническую запись различных команд вместо их шестнадцатеричного представления. Например, вместо цифрового кода команды загрузки регистра программист мог теперь написать `LD (от Load)`, а вместо кода команды копирования содержимого регистра в память мог использовать мнемоническое обозначение `ST (от Story)`. Для записи операндов были разработаны правила, в соответствии с которыми программист мог присваивать некоторым областям памяти описательные имена (идентификаторы) и использовать их при записи команд программы вместо адресов соответствующих ячеек памяти. Одним из специфических вариантов является присвоение мнемонических имен регистрам центрального процессора, например `R0, R1, R2, ...`

Используя идентификаторы для ячеек памяти и мнемонические обозначения для команд, программисты смогли значительно повысить читабельность написанных ими последовательностей машинных команд. Давайте вернемся, например, к программе на машинном языке, приведенной в конце раздела 2.2. Эта программа суммировала содержимое ячеек с адресами '6C' и '6D', после чего помещала результат в ячейку с адресом '6E'. Напомним, что в шестнадцатеричном виде соответствующая последовательность команд имеет следующий вид:

```
156C
166D
5056
306E
C000
```

Если мы присвоим имя `PRICE` ячейке с адресом '`6C`', имя `TAX` – ячейке адресом '`6D`' и имя `TOTAL` – ячейке с адресом '`6E`', то сможем переписать ту же самую программу с использованием мнемонических записей команд так, как показано ниже:

```
LD R5,PRICE
LD R6,TAX
ADDI R0,R5,R6
ST R0,TOTAL
HLT
```

Большинство читателей, вероятно, согласятся, что второй способ записи текста программы намного лучше отражает ее смысл, чем первый (оставаясь, впрочем, также не вполне понятным). Заметим, что мнемоническая запись `ADDI` здесь использована для команды сложения двух целых чисел, в то время как команду сложения двух чисел с плавающей точкой можно мнемонически обозначить как `ADDF`.

Вначале программисты использовали такие обозначения при разработке программ на бумаге, а затем переводили их на машинный язык. Однако вскоре стало понятно, что такой перевод может выполнить и сама машина. В результате были разработаны программы, названные ассемблерами и предназначенные для перевода записанных в мнемоническом виде программ на машинный язык. Название *ассемблер* (assembler – сборщик) эти программы получили потому, что их назначение заключалось в сборке машинных команд из кодов команд и операндов, полученных в результате перевода мнемонических обозначений и идентификаторов. Мнемонические системы записи программ стали, в свою очередь, рассматриваться как особые языки программирования, именуемые *языками ассемблера.*

В свое время разработка языков ассемблера считалась гигантским шагом вперед в поисках более совершенных технологий программирования. Многие считали, что они представляют собой совершенно новое поколение языков программирования. Со временем языки ассемблера стали называть языками программирования второго поколения, а 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 выполнить на другой машине, поскольку для этого ее нужно переписать с учетом новой конфигурации регистров и набора команд.

Кроссплатформенное программное обеспечение. При решении многих задач типичная прикладная программа вынуждена полагаться на операционную систему. Например, она может обратиться к диспетчеру окна для организации взаимодействия с пользователем или диспетчеру файлов для считывания 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. Чтобы отметить этот уровень 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.

Кроме того, 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 и других материалов. Конечно, реальная конструкция дома состоит именно из этих элементарных вещей, но проектировать его все же легче, имея дело с комнатами, окнами, дверьми и прочими подобными понятиями.

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 и были машинно-независимыми. 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 военного морского флота США для решения экономических задач.

В общем случае язык программирования третьего поколения представляет собой определенный набор языковых конструкций достаточно высокого уровня, предназначенный для разработки программного обеспечения. По существу, точно так же был разработан и наш псевдокод, описанный в главе 4. Каждая из языковых конструкций была разработана так, чтобы ее можно было реализовать в виде последовательности низкоуровневых примитивов, существующих в машинных языках. Рассмотрим следующий оператор:

```
Total ← Price + Tax
```

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. Таким образом, показанная ниже структура потенциально 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*. Разработку первого компилятора приписывают Грейс Хоппер (Grace Hopper), которая играла ведущую роль в продвижении концепции языков программирования высокого уровня. Действительно, идея писать программы в форме, близкой к естественному языку, была настолько революционной, что многие руководители поначалу отвергали ее.

Популярной альтернативой трансляторам являются *интерпретаторы* (interpreters), предложенные как еще один способ 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 немедленно выполняет все переведенные им инструкции.

Машинная независимость. 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 за счет использования соответствующего компилятора.

В действительности не все так просто. При разработке самого компилятора приходится учитывать определенные ограничения, накладываемые той машиной, для которой он предназначен. В результате эти ограничения отражаются на языке программирования, который подлежит переводу на машинный язык. Например, размер машинных регистров и ячеек памяти влияет на максимальный размер значений целых переменных, которыми может непосредственно оперировать программа. Такие ограничения приводят к тому, что один и тот же язык программирования на разных машинах имеет свои особенности, или *диалекты*. Вследствие этого программистам часто приходится выполнять, как минимум, легкую модификацию программы при переносе ее с одной машины на другую.

Проблема переноса программ с одной машины на другую заключается в отсутствии общей точки 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.

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 – это лишь промежуточная ступень на пути к достижению более важных целей. Со временем машинная независимость стала вполне достижимой, однако она стала менее важной по сравнению с другими велениями времени. Действительно, понимание того, что машина могла бы выполнять такие операторы высокого уровня, как

```
Total ← Price + Tax,
```

породило среди ученых в области компьютерных наук мечту 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 выбирать 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.

Парадигмы программирования. Классификация языков программирования по поколениям требует распределения их по линейной шкале (рис. 5.1) в соответствии с той степенью свободы от компьютерной тарабарщины, которую данный язык предоставляет программисту. Это позволяет ему мыслить понятиями, связанными непосредственно с решаемой задачей. В действительности развитие языков программирования происходило несколько иначе. Оно протекало по разным направлениям, связанным с альтернативными подходами к процессу программирования (называемыми парадигмами программирования). Таким образом, историческую схему развития языков программирования наиболее точно можно изобразить составной диаграммой, показанной на рис. 5.2, на которой отдельные линии, символизирующие различные парадигмы программирования, появляются и развиваются независимо друг от друга.

Рис. 5.1. Схематическое представление поколений языков программирования

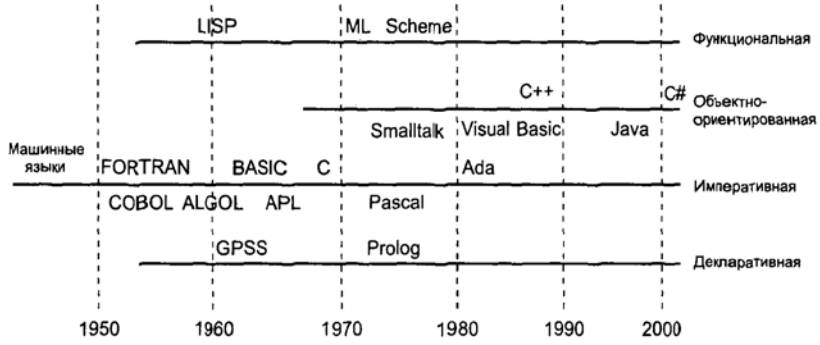


Рис. 5.2. Эволюция парадигм языков программирования

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.

Императивная (imperative paradigm), или *процедурная парадигма* (procedural paradigm), представляет традиционный подход к процессу программирования. Действительно, именно в соответствии с этой парадигмой построен цикл обработки команды центрального процессора: "извлечь-декодировать-выполнить". Как следует из названия, императивная парадигма определяет процесс программирования как запись последовательности команд, которая при выполнении выполнит обработку данных, необходимую для получения желаемого результата. Таким образом, для решения задачи императивная парадигма предлагает попытаться найти алгоритм ее решения.

В противоположность этому, *декларативная парадигма* (declarative paradigm) во главу угла ставит вопрос "Что представляет собой задача?", а не "Какой алгоритм нужен для решения задачи?". Основная проблема здесь состоит в том, чтобы создать и реализовать общий алгоритм решения задач. После этого задачи можно формулировать в виде, совместимом с этим алгоритмом, а затем применять его. В этом случае роль программиста заключается в точной формулировке задачи, а не в поисках и реализации алгоритма ее решения.

Основной трудностью в разработке декларативных языков программирования является выбор базового алгоритма решения задач. По этой причине ранние декларативные языки были узкоспециализированными по своей природе и ориентированными на специфические приложения. Например, декларативный подход уже многие годы применяется для моделирования систем (экономических, физических, политических и т.п.) в целях проверки выдвинутых гипотез. 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. Использование декларативного языка для выполнения такого моделирования сводится, прежде всего, к реализации алгоритма, выполняющего указанную повторяющуюся процедуру. В результате программисту остается лишь описать взаимоотношения моделируемых параметров. Далее базовый алгоритм моделирования просто имитирует течение времени, используя указанные соотношения для выполнения требуемых вычислений.

Не так давно декларативная парадигма нашла свое новое применение – благодаря осознанию 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 5.7.

Функциональная парадигма (functional paradigm) рассматривает процесс разработки программ как конструирование ее из неких "черных ящиков", каждый из которых получает некоторые исходные данные (на входе) и вырабатывает соответствующий результат (на выходе). Математики называют такие "ящики" функциями, поэтому этот подход называется функциональной парадигмой. Языковые конструкции функциональных языков программирования состоят из элементарных функций, на основе которых программист должен создавать более сложные функции, необходимые для решения поставленной задачи. Таким образом, согласно функциональной парадигме, программист решает задачу, рассматривая исходные данные, требуемые результаты и преобразование, которое необходимо выполнить, чтобы получить результаты из исходных данных. Решение требуемой задачи, вероятнее всего, можно получить, разбивая исходное преобразование на более простые преобразования, порождающие промежуточные результаты, служащие, в свою очередь, исходными данными для других простых преобразований. Короче говоря, в соответствии с функциональной парадигмой процесс программирования заключается в конструировании требуемых функций в виде вложенных друг в друга совокупностей более простых функций.

Например, на рис. 5.3 показано, как можно построить функцию вычисления среднеарифметического нескольких чисел из трех более простых функций. Первая из них – `Sum` – получает на вход список чисел и вычисляет их сумму; вторая – `Count` – получает список чисел и подсчитывает их количество; третья – `Divide` – получает на вход два числа и вычисляет их частное. На языке LISP (популярном функциональном языке программирования) эта конструкция может быть записана в виде следующего выражения:

```
(Divide (Sum Numbers) (Count Numbers))
```

Использование в этом выражении вложенных структур отражает, что исходные данные для функции `Divide` являются результатами выполнения функций `Sum` и `Count`. В качестве другого примера предположим, что у нас есть функция Sort, которая сортирует список чисел, и функция `First`, которая находит первое число в этом списке. В этом случае приведенное ниже выражение позволяет извлечь из списка `List` наименьшее из чисел:

```
(First (Sort List))
```

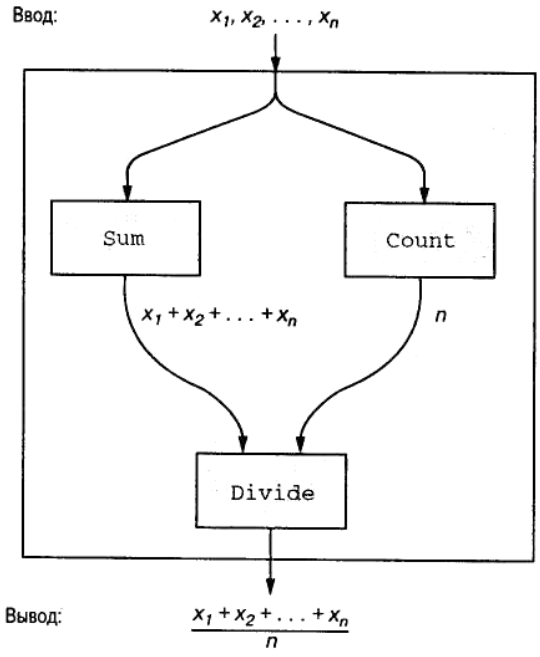


Рис. 5.3. Функция вычисления среднеарифметического для списка чисел, построенная из более простых функций `Sum`, `Count` и `Divide`

В данном случае использование вложенных структур означает, что результат функции `Sort` является исходной информацией для функции `First`. Таким образом, список сначала сортируется, а затем из отсортированного списка извлекается

первое число.

Превосходство функциональной парадигмы программирования над императивной проявляется в том, что она стимулирует модульный подход к конструированию программ. Действительно, то, что программы рассматриваются как функции, которые, в свою очередь, должны состоять из других функций, вынуждает программиста думать в терминах модулей. По этой причине сторонники функционального программирования утверждают, что этот подход приводит к созданию более высокоорганизованных программ, чем в случае применения императивной парадигмы. Более того, многие утверждают, что функциональная парадигма является естественной средой для метода, предусматривающего построение программ из "строительных блоков". Данный подход напоминает скорее конструирование программ из заранее подготовленных блоков, нежели выполнение всей работы с нуля. Такому способу программирования отдают предпочтение в основном специалисты по разработке больших пакетов программ. Эти же аргументы приводятся и в защиту объектно-ориентированной парадигмы.

Объектно-ориентированная парадигма (object-oriented paradigm), которая предполагает применение методов объектно-ориентированного программирования (ООП), – это еще один подход к процессу разработки программного обеспечения. В рамках этого подхода элемент данных рассматривается как активный "объект", а не как пассивный элемент, как это принято в традиционной императивной парадигме. Поясним это на примере списка 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. Таким образом, список является пассивным объектом, поскольку он обрабатывается управляющей программой, а не обрабатывает себя сам. Однако при объектно-ориентированном подходе список рассматривается как объект, содержащий некоторую совокупность данных вместе с набором процедур для их обработки. Этот набор может включать процедуры для вставки в список нового элемента, удаления элемента из списка или сортировки списка. Поэтому программа, получающая доступ к списку для его обработки, не обязана содержать алгоритм для выполнения указанных действий. При необходимости она просто выполняет процедуры, предоставляемые самим объектом. В этом смысле объектно-ориентированная программа вместо сортировки списка (как при императивной парадигме) скорее просит список отсортировать самого себя.

Язык Visual Basic. 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 графические интерфейсы пользователя (GUI). В действительности Visual Basic – это 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, программист должен описать, что должно случиться, если пользователь щелкнет на ней. В главе 6 мы увидим, что эта стратегия создания программ из готовых компонентов представляет собой важнейшую современную тенденцию в области разработки программного обеспечения.

Популярность операционной системы Windows и удобство пакета для разработки программ Visual Basic способствовали тому, что язык Visual Basic в настоящее время стал одним из наиболее известных и широко используемых языков программирования.

В качестве другого примера использования объектно-ориентированного подхода рассмотрим задачу разработки графического интерфейса пользователя. В этом случае все отображаемые на экране графические элементы реализуются как объекты. Каждый из этих объектов включает собственный набор процедур, определяющих реакцию объекта на возникновение различных событий, – выбор этого объекта, щелчок 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.

Многие из преимуществ объектно-ориентированного проектирования являются следствием модульной структуры, которая возникает как естественный побочный эффект от применения объектно-ориентированного подхода. В рамках этого подхода каждый объект реализуется в виде отдельного, точно определенного элемента. После того как свойства некоторой сущности будут определены подобным образом, полученное определение можно повторно использовать всякий раз, когда возникнет потребность в этой сущности. По этой причине сторонники объектно-ориентированного программирования утверждают, что объектно-ориентированная парадигма предоставляет естественную среду для конструирования программного обеспечения из "строительных блоков". Они предсказывают появление программных библиотек, содержащих определения различных объектов, с помощью которых новое программное обеспечение можно будет собирать точно так же, как обычные промышленные изделия собирают из готовых компонентов.

Объектно-ориентированная парадигма оказывает все большее влияние на область компьютерных наук, поэтому в разделе 5.5 мы детально обсудим ее особенности. Кроме того, в последующих главах мы вновь и вновь будем встречаться с проявлениями этой парадигмы. В частности, будет показано, какое влияние оказала объектно-ориентированная парадигма на методы разработки программного обеспечения (глава 6) и проектирования баз данных (глава 9), а в главе 7 мы увидим, как объектно-ориентированный подход к разработке программного обеспечения естественным образом обобщает результаты исследований в области структур данных.

Наконец, следует заметить, что процедуры объекта, описывающие, как объект должен отвечать на различные сообщения, в сущности, представляют собой небольшие императивные программные единицы. Поэтому большинство объектно-ориентированных языков программирования обладают свойствами императивных языков. Например, распространенный объектно-ориентированный язык C++ был создан добавлением к императивному языку C объектно-ориентированных свойств. В разделах 5.2 и 5.3 мы рассмотрим общие характеристики императивных и объектно-ориентированных языков и понятия, которые объединяют современное программное обеспечение.

Вопросы для самопроверки

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?

5.2. КОНЦЕПЦИИ ТРАДИЦИОННОГО ПРОГРАММИРОВАНИЯ

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++, FORTRAN, Java и Pascal. FORTRAN, Pascal и C – императивные языки программирования третьего поколения, тогда как C++ – объектно-ориентированный язык, который является расширением языка C. Java – это объектно-ориентированный язык, производный от C и C++. Язык Ada изначально был разработан как императивный язык третьего поколения, обладающий многими объектно-ориентированными свойствами. Однако более поздние версии этого языка больше соответствуют объектно-ориентированной парадигме, чем императивной.

В приложении Г содержится краткое описание каждого из этих языков программирования, дополненное примером того, как алгоритм сортировки по методу вставки может быть реализован в каждом из них. Вы можете обращаться к этому приложению по мере чтения данного раздела. Помните, однако, что в данном случае наша цель – понять основные свойства языков программирования. Приводимые здесь примеры предназначены просто для иллюстрации того, как обсуждаемые функции практически реализуются в существующих языках программирования, поэтому вам не следует слишком углубляться в их рассмотрение.

Операторы в языках программирования обычно подразделяются на три категории: операторы объявления, выполняемые операторы и комментарии. *Операторы объявления* (declarative statements) задают способ представления, тип и структуру данных, которые в дальнейшем будет использоваться в программе. *Выполняемые операторы* (imperative statements) описывают шаги применяемого алгоритма, а *комментарии* (comments) повышают читабельность программы, поясняя ее специфические особенности в более удобной для пользователя форме. Этот раздел мы начнем с изучения понятий, связанных с операторами объявления, затем перейдем к обсуждению выполняемых операторов и закончим рассмотрением примера документирования программы.

Переменные, константы и литералы. В разделе 5.1 говорилось о том, что языки программирования высокого уровня позволяют обращаться к ячейкам памяти через символьные имена, а не через числовые адреса. Такие имена называются *переменными* (variable), тем самым подчеркивается тот факт, что при изменении значения, хранящегося в ячейке памяти, изменяется и значение, присвоенное переменной.

Однако 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, 645 футов), когда оно потребуется. Такое явное указание конкретного значения для данных называется *литералом* (literal). Использование литералов приводит к появлению в программах операторов, подобных приведенному ниже:

```
EffectiveAlt ← Altimeter+645,
```

где `EffectiveAlt` и `Altimeter` являются переменными, а значение `645` – литералом.

Как правило, применение литералов не считается лучшим стилем программирования, поскольку они затрудняют понимание тех выражений, в которых используются. Например, как читающий программу сможет узнать, что именно означает число `645`? Кроме того, литералы могут усложнить модификацию программы, когда это станет необходимым. Если 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 `645`, то каждую такую ссылку в программе нужно найти и изменить. Задача еще более усложнится, если окажется, что литерал `645` в некоторых случаях представляет также и другую величину, а не только высоту аэропорта над уровнем моря. Как тогда узнать, какой из литералов следует изменить, а какой оставить неизменным?

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 о них рассказывается в разделе 5.3), пользователи языка C называют оба конструкта функциями. Происходит это по той причине, что в языке C процедура рассматривается как функция, не возвращающая никакого 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 языках Pascal и Ada зарезервированные слова принято выделять полужирным шрифтом, тогда как пользователи языков C, C++, Fortran и Java не придерживаются этой традиции.

Текст этой книги выдержан в нейтральном стиле благодаря использованию классической терминологии, применяемой теоретиками. Однако каждый конкретный пример представлен в форме, совместимой с традициями данного языка. Встретив подобный пример, читатель должен помнить, что это всего лишь образец того, как теоретические идеи реализованы в реальном языке программирования, и он вовсе не предназначен для обучения читателя особенностям работы с тем или иным языком программирования.

Для решения подобных проблем языки программирования позволяют давать описательные имена конкретным постоянным величинам. Такое имя называется *именованной константой* или просто *константой* (constant). Например, рассмотрим следующий оператор объявления языка Pascal:

```
const AirportAlt = 645;
```

Этот оператор связывает идентификатор `AirportAlt` с фиксированным значением, равным `645`**.** Аналогичные действия в языке Java записываются в виде следующего оператора:

```
final int AirportAlt = 645;
```

В результате подобного объявления имя `AirportAlt` можно будет использовать вместо литерала `645`**.** Используя такую константу в нашем псевдокоде, мы можем переписать оператор

```
EffectiveAlt ← Altimeter + 645
```

в виде

```
EffectiveAlt ← Altimeter + AirportAlt.
```

Последний вариант лучше представляет смысл программы. Кроме того, если в программе вместо литералов используются подобные именованные константы и эту программу потребуется перенести в другой аэропорт, расположенный на высоте 267 футов над уровнем моря, то все, что нужно сделать, для того чтобы присвоить всем ссылкам на высоту аэропорта новые значения, – это изменить одно-единственное объявление следующим образом:

```
const AirportAlt = 267;
```

Типы данных. Операторы объявления, с помощью которых данным присваиваются имена, обычно одновременно определяют и их тип. *Тип данных* (data type) определяет как область допустимых значений, так и операции, которые можно с ними выполнять. К основным типам данных относятся *integer* (целый), *real* (действительный), *character* (символьный) и *Boolean* (логический, или булев). Тип integer используется для обозначения числовых данных, являющихся целыми числами. В памяти они чаще всего представляются с помощью двоичной нотации с дополнением. С данными типа integer можно выполнять обычные арифметические операции и операции сравнения. Тип real предназначен для представления числовых данных, которые могут содержать нецелые величины. В памяти они обычно хранятся как двоичные числа 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.

Тип character используется для данных, состоящих из символов, которые хранятся в памяти в виде кодов ASCII или UNICODE. Данные этого типа можно сравнивать друг с другом (определять, какой из двух символов предшествует другому в алфавитном порядке); проверять, является ли одна строка символов частью другой, а также объединять две строки в одну, более длинную строку, дописывая одну из них после другой (операция конкатенации).

Тип Boolean относится к данным, которые могут принимать только два значения: *true* (истина) и *false* (ложь). Примером таких данных может служить результат выполнения операции сравнения двух чисел. Операции с данными типа Boolean включают проверку, является ли текущее значение переменной true или false.

Другие типы данных, которым пока не соответствуют какие-либо общепринятые элементарные конструкции в основных языках программирования, – это аудио- и видеоданные. Встроенные средства для обработки таких данных имеются в среде программирования языка Java.

В большинстве языков программирования требуется, чтобы операторы объявления не только вводили новую переменную в программу, но и определяли ее тип. На рис. 5.4 приведены примеры таких объявлений в языках Pascal, C, C++, Java и FORTRAN. В каждом случае переменным `Length` и `Width` присвоен тип real, а переменным `Price`, `Tax` и `Total` – тип integer. Обратите внимание, что в языках C, C++ и Java для ссылки на тип данных real используется ключевое слово float, поскольку данные этого типа представляются в машине как числа с плавающей точкой.

В разделе 5.5 мы увидим, как транслятор использует сведения о типах данных при переводе программы с языка высокого уровня на машинный. Заметим, что эту информацию можно использовать и для обнаружения ошибок. Например, попытка сложить два значения типа character или выполнить операцию, манипулирующую данными разных типов, может вызвать у транслятора подозрение.

Структура данных. Другим понятием, связанным с операторами объявления, является *структура данных* (data structure), определяющая общую форму представления данных. Например, текст обычно рассматривается как длинная строка символов, тогда как информация о продажах может рассматриваться как прямоугольная таблица с числовыми значениями, каждое из которых представляет число сделок, заключенных определенным работником в определенный день.

Описание переменных в языке Pascal	`var`
	`Length, Width: real;`
	`Price, Tax, Total: in`
	`teger;`
	`Symbol: char;`

Описание переменных в языках C, C++ и Java	`float Length, Width;`
	`int Price, Tax, Total;`
	`char Symbol;`

Описание переменных в языке FORTRAN	`REAL Length, Width`
	`INTEGER Price, Tax, Total`
	`CHARAKTER Symbol`

Рис. 5.4. Объявление переменных в языках Pascal, C, C++, Java и FORTRAN

а) Объявления в языках C и C++

```
int Scores [2][9];
```

б) Объявления в языке Java

```
int Scores [][] = new int[2][9];
```

в) Объявления в языке Pascal

```
var
  Scores: array[1..2,1..9] of integer;
```

г) Вид структуры, объявляемой в каждом из приведенных выше примеров

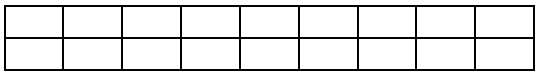


Рис. 5.5. Объявление двухмерного массива в языках C, C++, Java и Pascal

Одной из общих структур данных является *однородный массив* (homogeneous array), который представляет собой блок значений одного типа, например линейный список, двухмерную таблицу из строк и столбцов или таблицу более высокой размерности. Для объявления такого массива в большинстве языков программирования используются специальные операторы объявления, содержащие указания о длине каждой размерности массива. Например, на рис. 5.5 представлены операторы языков C, Java и Pascal, объявляющие переменную `Scores` как двухмерный массив целых чисел из двух строк и девяти столбцов.

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* (indices), указывающих номер нужной строки, столбца и т.д. Например, в программе на языке Pascal элемент, находящийся на пересечении второй строки и четвертого столбца массива `Scores`, обозначается как `Scores[2,4]`; тогда как в языках C, C++ и Java этот же элемент массива будет обозначаться как `Scores[1][3]`. В этих языках нумерация строк и столбцов начинается с нуля, т.е. элемент, находящийся на пересечении первой строки и первого столбца массива, обозначается как `Scores[0][0]`.

В противоположность однородному массиву, в котором все элементы имеют один и тот же тип, *неоднородный массив* (heterogeneous array) представляет собой блок данных, в котором отдельные элементы могут иметь разный тип. Например, блок данных, относящийся к некоторому работнику, может содержать элемент `Name` типа character, элемент `Age` типа integer, а также элемент `SkillRating` типа real.

В языках Pascal и C такой тип массива называется соответственно *записью* (record) и *структурой* (structure). Ниже показано, как можно объявить такой массив в языках C и Pascal.

а) Объявления в языках C и C++

```
struct
{    char Name[8];
     int Age;
     float SkillRating;
} Employee;
```

б) Объявление в языке Pascal

```
var
    Employee: record
      Name: packed array[1..2,1..9] of char;
      Age: integer;
      SkillRating: real;
     end;
```

в) Вид структуры, объявляемой в каждом из приведенных выше примеров

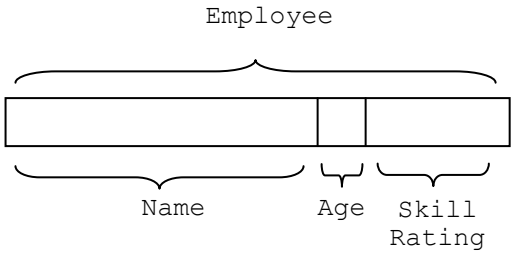


Рис. 5.6. Объявление неоднородного массива в языках C, C++ и Pascal

При ссылке на компоненты неоднородного массива обычно указывают имя массива и имя элемента, разделенные точкой. Например, в языках C, C++, Java и Pascal для доступа к элементу `Age` массива `Employee`, показанного на рис. 5.6, можно использовать имя `Employee.Age`.

В главе 7 мы увидим, 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 как концептуальные формы их представления. В действительности же "реальная" форма представления данных в запоминающих устройствах машины может совершенно отличаться от теоретической формы.

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 в качестве строительных блоков базовые типы и структуры. Такие "самодельные" типы данных принято называть *типами, определяемыми пользователем* (user-defined types).

В качестве примера рассмотрим разработку программы, 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. Например, для создания переменной Employee, имеющей указанную смешанную структуру, работающий на языке C программист мог бы написать следующий оператор (см. рис. 5.6 в главе 5):

```
Struct
{char  Name[8];      // Имя
 int   Age;          // Возраст
 float SkillRating; // Показатель квалификации
} Employee; // Структура "Работник"
```

При таком подходе проблема состоит в том, что если описание структуры повторяется слишком часто, то программа увеличивается в размерах и чтение ее становится затруднительным. Лучше было бы описать структуру лишь однажды и присвоить ей описательное имя, а затем использовать его каждый раз при ссылке на данную структуру. Язык C позволяет программисту сделать это с помощью оператора `typedef` (сокращение от type definition, определение типа):

```
typedef struct
{char  Name[8];      // Имя
 int   Age;          // Возраст
 float SkillRating; // Показатель квалификации
} EmployeeType; // Тип данных "Работник"
```

Этот оператор определяет новый тип данных, `EmployeeType`, который можно использовать при объявлении переменных наряду с базовыми типами языка. Например, переменная `Employee` теперь может быть объявлена с помощью следующего оператора:

```
EmployeeType Employee;
```

Преимущество использования таких пользовательских типов очевидно при объявлении множества переменных этого типа. С помощью приведенного ниже оператора программист на языке C может объявить, что переменные `Sleeve`, `Waist` и `Neck` относятся к базовому типу "действительный":

```
float Sleeve, Waist, Neck;
```

Аналогично этому, после определения пользовательского типа `EmployeeType`, с помощью следующего оператора можно объявить, что переменные `DistManager`, `SalesRepl` и `SalesRep2` относятся к данному типу:

```
EmployeeType DistManager, SalesRepl, SalesRep2;
```

Важно отличать определенные пользователем типы данных и сами элементы данных этих типов. Последние рассматриваются как *реализации* (instance) 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 тип пользователя `EmployeeType` использовался для создания трех реализаций этого типа: `DistManager`, `SalesRepl` и `SalesRep2`.

Указатели. Вспомните, что ячейки в оперативной памяти машины идентифицируются числовыми адресами. Эти числовые значения можно также хранить в ячейках памяти. *Указатель* (pointer) – это ячейка (или блок ячеек) памяти, содержащая адрес другой ячейки памяти. Применительно к структурам данных, указатели используются для записи адресов элементов данных. Таким образом, элемент данных может храниться в какой-либо ячейке памяти, а адрес этой ячейки – в указателе, при помощи которого можно позже получить эти данные. То есть значение указателя сообщит нам, где искать данные. В некотором смысле указатель указывает на данные, отчего и получил такое название.

Мы уже встречались с концепцией указателей в контексте счетчика команд процессора, который содержит адрес очередной инструкции для выполнения. Фактически, другое название счетчика команд – *указатель команд* (instruction pointer). Адреса, также называемые URL, которые используются для связи гипертекстовых документов, также могут служить примером концепции указателей, но они указывают местоположения в сети Интернет, а не в оперативной памяти компьютера.

Во многих современных языках программирования указатели включены в набор основных типов данных. Можно объявлять, выделять память и манипулировать указателями так же, как целыми числами или строками. При помощи такого языка программист может создавать развитые сети элементов данных в памяти машины, где каждый блок ячеек памяти содержит указатели на другие блоки. Следуя указателям, можно проследить эти пути от блока к блоку.

В качестве примера давайте представим, что в компьютерной памяти хранится список рассказов, отсортированный в алфавитном порядке по названию. Такая организация удобна во многих приложениях, но одновременно затрудняет поиск всех рассказов, написанных одним автором, так как они беспорядочно разбросаны по списку. Для решения этой проблемы можно зарезервировать в каждом блоке ячеек памяти, представляющем один рассказ, отдельную ячейку типа указатель. Тогда в каждом из этих указателей можно хранить адрес другого блока, представляющего произведение того же автора, и все рассказы одного автора будут связаны в замкнутую цепь. Отыскав один рассказ заданного автора, мы можем найти и все остальные, переходя по указателям от книги к книге.

Проблема указателей. Известно, что использование блок-схем может привести к путанице при разработке алгоритмов (см. главу 4), а беспорядочное использование команд безусловного перехода goto ведет к созданию плохо спроектированных программ (см. главу 5). 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 массива, он должен использовать инструкцию, эквивалентную следующему выражению:

```
переадресовать ссылку Next к следующему элементу массива
```

Тогда как программист, работающий на языке C, может использовать инструкцию, эквивалентную следующей:

```
присвоить ссылке Next значение 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 массива. Однако если ссылка Next уже указывает на последний элемент массива, то выполнение инструкции языка C приведет к тому, что она будет указывать на нечто, находящееся вне массива, – распространенная ошибка начинающих (и не только начинающих) программистов.

Операторы присваивания. Наиболее важным выполняемым оператором является *оператор присваивания* (assignment statement), предназначенный для присвоения переменной некоторого значения. 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 присвоении полученного результата переменной, указанной в левой части оператора. Например, в языках C, C++ и Java в результате выполнения приведенного ниже оператора, переменной Total присваивается сумма значений переменных Price и Tax:

```
Total = Price + Tax;
```

В языках Ada и Pascal эквивалентный оператор записывается в следующем виде:

```
Total := Price + Tax;
```

Обратите внимание, что эти операторы отличаются только синтаксисом операции присваивания, которая в языках C, C++ и Java обозначается просто знаком равенства, а в языках Ada и Pascal перед знаком равенства ставится двоеточие. Возможно, более удачное обозначение операции присваивания используется в языке APL (A Programming Language – язык программирования), разработанном Кеннетом Иверсеном (Kenneth E. Iverson) в 1962 году. В этом языке для представления операции присваивания используется стрелка. Таким образом, предыдущий оператор присваивания в языке APL (как и в нашем псевдокоде) будет записан следующим образом:

```
Total ← Price + Tax
```

"Мощь" оператора присваивания определяется диапазоном выражений, допустимых в правой части оператора. Как правило, разрешается использовать любые алгебраические выражения с арифметическими операциями сложения, вычитания, умножения и деления, обычно обозначаемые символами +, –, * и /, соответственно. Однако языки программирования поразному интерпретируют подобные выражения. Например, при вычислении выражения `2*4+6/2` справа налево получим результат 14, а слева направо – значение 7. Во избежание таких неоднозначностей обычно устанавливаются *приоритеты операций* (operator precedence), определяющие порядок выполнения операций в выражениях. Традиционно умножение и деление имеют более высокий приоритет, чем сложение и вычитание. Таким образом, операции умножения и деления должны выполняться до сложения и вычитания. В соответствии с этим при вычислении приведенного выше выражения получим 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.

Выражения в операторах присваивания могут содержать не только обычные алгебраические операции. Например, пусть `First` и `Last` – переменные, имеющие тип строки символов. Рассмотрим следующий оператор языка FORTRAN:

```
Both = First // Last
```

В результате его выполнения переменной `Both` в качестве значения будет присвоена строка символов, полученная посредством конкатенации строк из переменных `First` и `Last`. Таким образом, если переменные `First` и `Last` содержат строки `abra` и `cadabra`, соответственно, то переменная `Both` будет иметь значение `abracadabra`.

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 языке Java, этот символ означает операцию конкатенации, если операндами являются строки символов. Такое многозначное использование символов операций называется *перегрузкой* (overloading).

Управляющие операторы. *Управляющие операторы* (control statement) 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 ничем иным, как прямым применением машинной команды передачи 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:

```
     goto 40
20   Total = Price + 10
     goto 70
40   if Price < 50 then goto 60
     goto 20
60   Total = Price + 5
70   ...
```

Однако эти же действия можно выполнить с помощью буквально двух следующих операторов:

```
if (Price < 50)
   then Total = Price + 5
   else Total = Price + 10
```

Чтобы избежать подобных ситуаций, современные языки программирования включают более продуманный набор управляющих операторов, позволяющий представлять разветвленные структуры с помощью единственного оператора. На рис. 5.7 показаны некоторые наиболее распространенные структуры ветвления и соответствующие управляющие операторы, используемые в различных языках программирования для их представления. Заметим, что первые два типа структур уже упоминались в главе 4. В нашем псевдокоде они представляются операторами **`if-then-else`** и **`while`**. Третью структуру, известную под названием "оператор **`case`**", можно рассматривать как обобщение структуры **`if-then-else`**. В то время как структура **`if-then-else`** допускает выбор только из двух возможностей, оператор **`case`** позволяет сделать выбор одного из многих описанных вариантов.

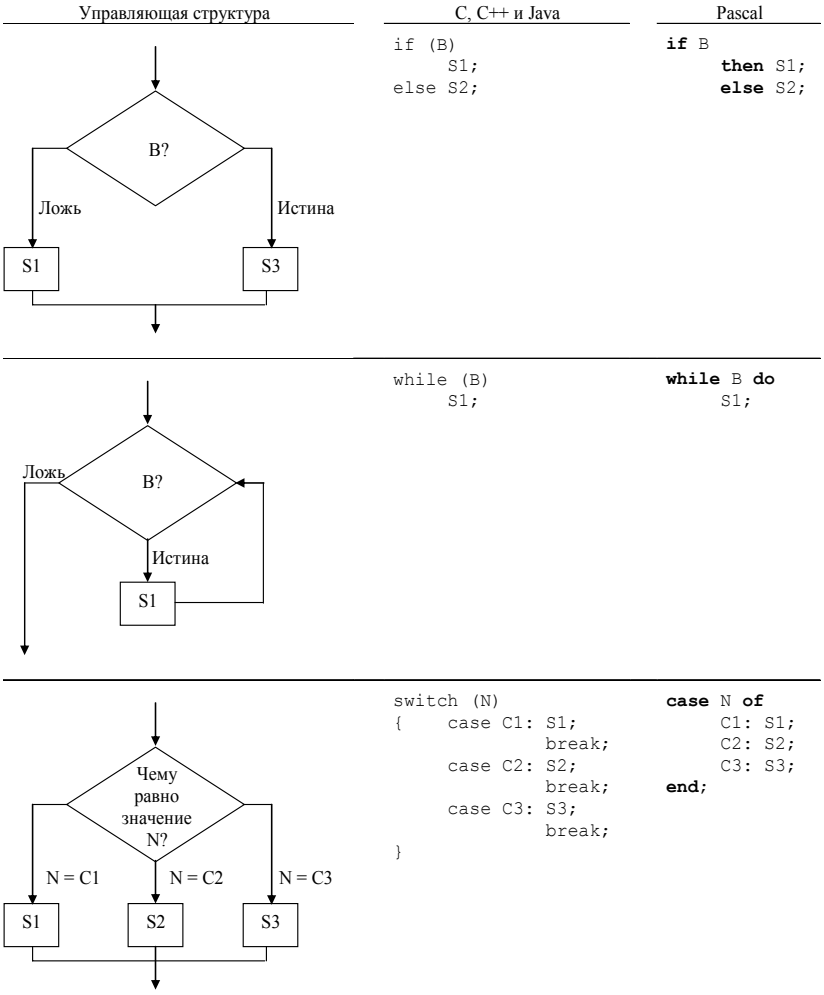


Рис. 5.7. Управляющие структуры и их представление в языках C, C++, Java и Ada

Другие широко распространенные типы структур, часто называемые структурами типа **`for`**, реализуются в разных языках программирования так, как показано на рис. 5.8. Эти циклические структуры

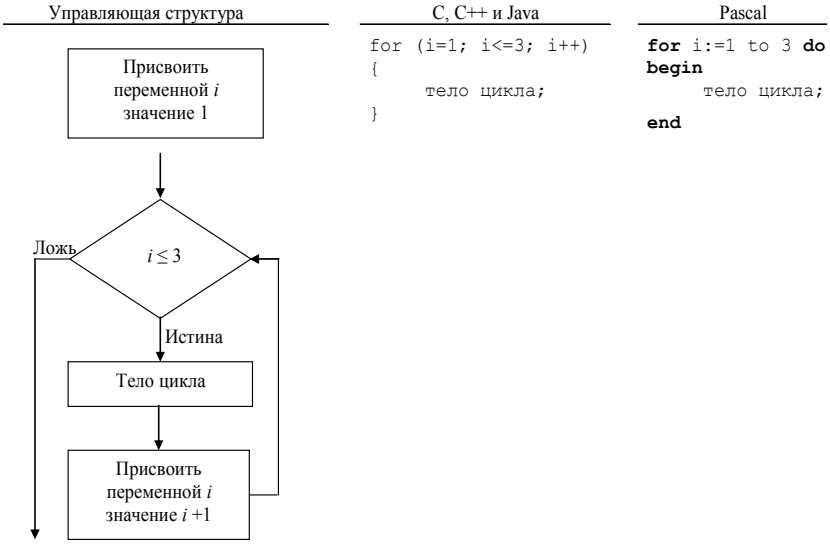


Рис. 5.8. Структура `for` и ее представление в языках C, C++, Java и Pascal

похожи на оператор **`while`** нашего псевдокода. Разница лишь в том, что в структуре цикла инициализация и модификация счетчика цикла, а также проверка условия выхода выполняются единственным оператором. Такой оператор удобен, когда тело цикла выполняется один раз для каждого значения переменной из заданного диапазона. В частности, представленные на рис. 5.8 операторы указывают, что тело цикла должно выполняться несколько раз – сначала со значением переменной `i`, равным 1, затем со значением переменной `i`, равным 2, и еще раз со значением переменной `i`, равным 3.

Назначение этих примеров – продемонстрировать, что типичные структуры ветвления программы присутствуют (с не-

значительными 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 на некотором языке программирования. Мы обсудим это позже, в главе 11. 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.

Выбор набора управляющих структур, предназначенных для включения в язык программирования, является одним из аспектов его разработки. Задача состоит не только в том, чтобы обеспечить язык средствами выражения алгоритмов в наиболее удобной форме, но и в том, чтобы помочь программисту действительно достичь этого. Для достижения этой цели необходимо отказаться от использования тех языковых конструкций, которые допускают небрежное программирование, и поощрить применение более продуманных проектных решений. Результатом такого подхода стала разработка часто неверно трактуемой методологии, известной как *структурное программирование* (structured programming). Она объединяет в себе строгие требования к организации проектирования с соответствующим использованием управляющих структур языка программирования. Назначение этого подхода заключается в создании понятных и хорошо организованных программ, полностью отвечающих требованиям своих спецификаций.

Комментарии. Как показывает практика, независимо от того, насколько хорошо разработан язык программирования и насколько правильно использованы его возможности, всякий раз, когда человек пытается разобраться в программе сколько-нибудь значительного размера, дополнительная информация о ней оказывается либо полезной, либо просто необходимой. Поэтому в языках программирования предусмотрены синтаксические конструкции для вставки в программу поясняющих операторов, называемых *комментариями* (comments). Документация, составленная из таких комментариев, называется *внутренней*, поскольку она заключена внутри программы, а не в отдельном документе.

Транслятор игнорирует внутреннюю документацию, поэтому ее 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 превышают пределы понимания программиста.

Существуют два способа выделения комментариев из остального текста программы. Первый состоит в том, чтобы пометить 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 них комментарий может размещаться между парами символов `/*` и `*/` `и содержать более одной строки`, или 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. */
// Это тоже комментарий.
```

Скажем несколько слов о том, что следует считать осмысленным комментарием. Начинающие программисты при внесении комментариев в целях создания внутренней документации обычно делают это следующим образом:

```
Total := Price + Tax; // Вычисляем Total, складывая Price и Tax
```

Подобные комментарии совершенно излишни и скорее просто увеличивают размер программы, нежели проясняют ее суть. Запомните, что задача внутренней документации – объяснить другому программисту смысл программы, а не просто перефразировать выполняемые в ней действия. Более приемлемым комментарием для приведенного выше оператора было бы краткое пояснение, зачем вычисляется значение `Total`, если это не очевидно из предыдущего текста. Например, комментарий "`Переменная Total используется ниже при вычислении значения переменной GrandTotal и после этого будет не нужна`" будет более полезен, чем предыдущий.

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 объяснения. Здесь также целесообразно будет описать задачу и общие свойства данного программного модуля. Если это 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.

Вопросы для самопроверки

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 отличаются однородные и неоднородные массивы?

5.3. ПРОЦЕДУРЫ И ФУНКЦИИ

В предыдущих главах мы убедились в преимуществах разделения больших программ на более мелкие и удобные в работе модули. Для выполнения такой декомпозиции языки программирования предоставляют много различных средств. Языки, поддерживающие функциональную парадигму, естественным образом требуют разделения программы на отдельные

функции. Языки, поддерживающие объектно-ориентированную парадигму, позволяют создавать программные модули, представляющие отдельные объекты.

В этом разделе мы сосредоточим внимание на методах модульного представления алгоритмов. Этот подход состоит в объединении шагов алгоритма в короткие, простые фрагменты с последующим их применением в качестве абстрактных инструментов для достижения желаемой цели. В результате образуется некоторая структура из подпрограмм, каждая из которых кодируется на нашем псевдокоде как отдельная процедура.

Процедуры. *Процедура* (procedure) – это модуль программы, написанный независимо от других модулей и связанный с ними с помощью процедуры передачи и возврата управления (рис. 5.9). Управление передается процедуре (посредством команды перехода машинного языка) в тот момент, когда возникает необходимость в получении предоставляемых ей услуг, и возвращается в модуль вызывающей программы после завершения работы процедуры. Процесс передачи управления часто называют вызовом процедуры. Мы будем называть модуль программы, который обращается к процедуре, *вызывающим модулем* (calling unit) или *вызывающей программой*.

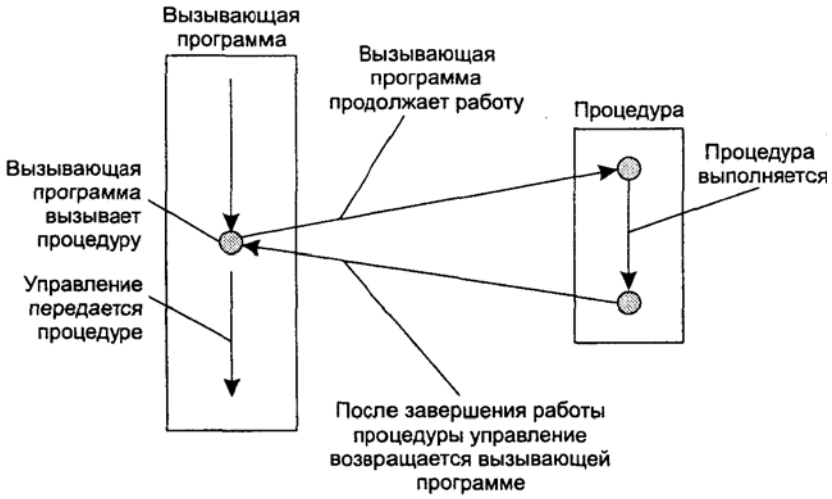


Рис. 5.9. Передача управления после вызова процедуры

В рассматриваемых языках программирования процедуры определяются почти так же, как и в нашем псевдокоде, обсуждавшемся в главе 4. Определение процедуры начинается с оператора, называемого *заголовком процедуры*, который, помимо всего прочего, содержит ее имя. За заголовком следуют операторы, детально описывающие данную процедуру. Во многих отношениях процедура – это миниатюрная программа. Она содержит как объявления используемых переменных и констант, так и выполняемые операторы, описывающие шаги алгоритма, для реализации которого эта процедура предназначена.

Как правило, переменные, объявленные в процедуре, являются *локальными* (local variable). Это означает, что на них можно ссылаться только внутри данной процедуры. Подобный подход исключает возможные недоразумения, которые могут возникнуть, когда две процедуры, написанные независимо друг от друга, используют переменные с одинаковыми именами. Однако иногда требуется, чтобы некоторые данные были доступны всем модулям внутри программы. Переменные, представляющие собой такие данные, называются *глобальными* (global variable). Многие языки программирования имеют средства для описания как локальных, так и глобальных переменных.

Событийно-управляемое программное обеспечение. В этой главе рассматриваются случаи, когда процедуры явно вызываются операторами, расположенными в различных местах программы. Однако иногда процедуры должны активизироваться неявно, при наступлении какого-то события. Например, в графическом интерфейсе пользователя (GUI) процедура, описывающая реакцию программы на щелчок на командной кнопке, вызывается не из другого программного модуля, а активизируется непосредственно в ответ на щелчок мышью на данном элементе. Программное обеспечение, содержащее подобные процедуры, называется *событийно-управляемым*. Короче говоря, такое программное обеспечение состоит из процедур, описывающих реакцию системы на различные события. При функционировании системы эти процедуры бездействуют, пока не наступит требуемое событие; после этого они активизируются, выполняют свою задачу и вновь возвращаются в состояние покоя.

Синтаксические конструкции, используемые для вызова процедур из другой части программы, в разных языках несколько отличаются. В языке FORTRAN используется оператор `CALL`. В языках Ada, C, C++, Java и Pascal просто указывается имя процедуры. Таким образом, если `GetNames`, `SortNames` и `WriteNames` – это процедуры для получения, сортировки и печати списка имен (определенного как глобальная переменная), то мы могли бы использовать их при написании приведенной ниже программы на языке FORTRAN, предназначенной для получения, сортировки и печати списков.

```
CALL GetNames
CALL SortNames
CALL WriteNames
```

В языках Ada, C, C++, Java и Pascal эта же программа выглядит следующим образом:

```
GetNames;
SortNames;
WriteNames;
```

Обратите внимание, что в этом случае программа состоит из трех команд, каждая из которых обращается к абстрактной

процедуре. Детали того, как именно каждая процедура выполняет свою задачу, скрыты от главной программы.

Параметры. Обычно не рекомендуется предоставлять информацию для совместного использования с помощью глобальных переменных, поскольку при этом трудно определить, какая часть программы использует эти данные. Лучше всего в явном виде определить, какие данные используются в каждом модуле программы. Для этого нужно перечислить данные, которые передаются процедуре оператором вызова. В свою очередь, заголовок процедуры содержит список переменных, которым будут присвоены значения, полученные при вызове процедуры, как и в нашем псевдокоде, описанном в главе 4. Элементы обоих списков называют *параметрами* (parameters).

При вызове процедуры параметры, перечисленные в вызывающем программном модуле, ставятся в однозначное соответствие параметрам, перечисленным в заголовке процедуры; первый параметр в вызывающем модуле соответствует первому параметру в заголовке процедуры и так далее. Затем значения параметров из вызывающего модуля присваиваются соответствующим параметрам процедуры, и процедура выполняется. Таким образом, как и в нашем псевдокоде, параметры, перечисленные в заголовке процедуры, указывают, где будут размещены конкретные данные при вызове процедуры. Поэтому такие параметры часто называют *формальными* (formal parameters), тогда как параметры, перечисленные в вызывающем модуле, именуют *фактическими* (actual parameters), поскольку они представляют реальные данные.

В некоторых языках программирования передача данных от фактических параметров к формальным осуществляется посредством копирования. При этом процедура может манипулировать лишь предоставленными ей копиями. Говорят, что такие параметры передаются *по значению* (passed value). Передача данных по значению защищает переменные в вызывающем модуле от ошибочного изменения плохо разработанной процедурой. Например, если вызывающий модуль передает процедуре номер социального страхования 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 процедуре состоит в предоставлении ей прямого доступа к фактическим параметрам посредством указания их адресов. В этом случае говорят, что параметры передаются *по ссылке* (passed by reference). Напомним, что передача параметров по ссылке позволяет вызываемой процедуре модифицировать данные в вызывающем модуле. Такой подход был бы желателен, например, при сортировке списка. И действительно, в этом случае вызов процедуры сортировки имел бы результатом переупорядочивание исходного списка.

Например, пусть процедура `Demo` описана так, как показано ниже:

```
procedure Demo(Formal)
Formal ← Formal+1;
```

Кроме того, предположим, что переменной `Actual` присвоено значение 5, после чего процедура `Demo` вызывается с помощью следующего оператора:

```
Demo(Actual)
```

(Здесь использована синтаксическая конструкция, более характерная для языков программирования, чем для нашего псевдокода, в котором этот оператор имел бы вид "`вызвать процедуру Demo с параметром Actual`".) В этом случае, если параметры передаются по значению, то изменения, внесенные в переменную `Formal` при выполнении процедуры `Demo`, не отразятся на значении переменной `Actual` (рис. 5.10). Однако если параметры передаются по ссылке, значение переменной `Actual` увеличится на единицу (рис. 5.11).

В разных языках программирования передача параметров осуществляется различными методами, но в любом случае использование параметров позволяет писать процедуры в абстрактном виде и применять их к конкретным данным в нужный момент.

Функции. Обычно назначение программного модуля состоит в выполнении действия и/или вычислении значения. Когда упор делается на вычислении значения, программный модуль может быть реализован в виде функции. В данном случае термин *функция* относится к программному модулю, который во всем похож на процедуру, за исключением того, что он возвращает в вызывающий модуль не список параметров, а единственное значение, которое называется "значением функции". Значение функции связано с именем функции так же, как значение переменной связано с именем переменной. Отличие заключается лишь в усилиях, которые нужно приложить, чтобы получить значение. При ссылке на переменную соответствующее значение извлекается из основной памяти, при использовании функции значение вычисляется путем выполнения инструкций, содержащихся в теле функции.

Например, если `Total` – это переменная, которой присвоена общая стоимость единицы продукции (цена плюс налог на продажу), то стоимость двух таких единиц можно найти, вычислив следующее выражение:

```
2 * Total
```

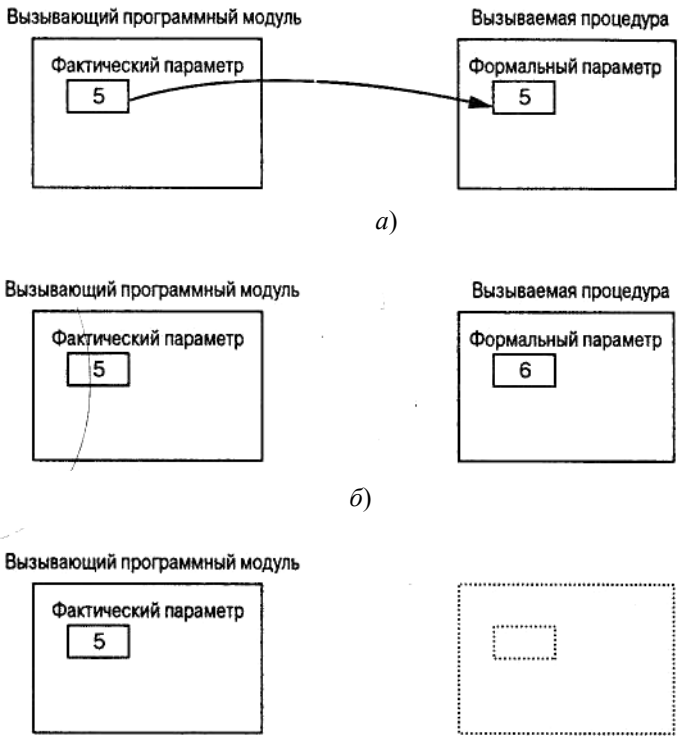


Рис. 5.10. Выполнение процедуры `Demo` с передачей параметра по значению:

а – при вызове процедуры ей передается копия переменной; *б* – процедура манипулирует этой копией; *в* – когда процедура заканчивает работу, переменная в вызывающем модуле не изменяется

В противоположность этому, если `TotalCost` – функция, которая вычисляет стоимость единицы продукции на основе ее цены и установленного налога на продажу, то стоимость двух единиц продукции можно найти, вычислив такое выражение:

```
2 * TotalCost(Price, TaxRate)
```

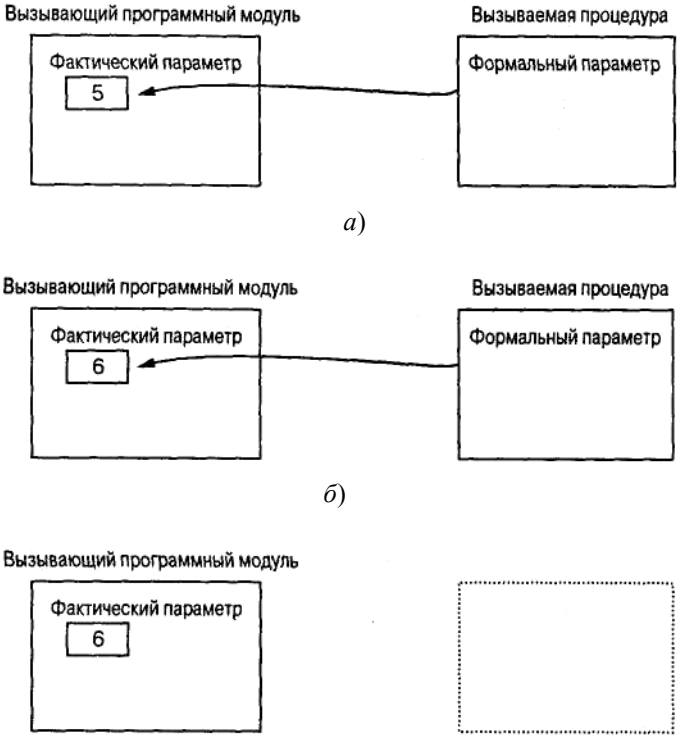


Рис. 5.11. Выполнение процедуры `Demo` с передачей параметра по ссылке:

а – при вызове процедуры формальный параметр становится ссылкой на фактический параметр; *б* – поэтому изменения, производимые процедурой, выполняются над фактическим параметром; *в* – следовательно, они сохраняются и после окончания работы процедуры

В первом случае значение переменной `Total` просто извлекается из памяти и умножается на число 2. Во втором случае

функция TotalCost выполняется для переданных ей на вход значений Price и TaxRate, после чего полученное значение умножается на 2.

Функции определяются почти так же, как процедуры. Различие состоит в том, что заголовок функции обычно начинается с описания типа возвращаемого этой функцией значения, а в теле функции присутствует оператор возврата, операндом которого является возвращаемое значение.

Операторы ввода/вывода. Процедуры и функции расширяют возможности языков программирования. Если язык не предусматривает некоторую операцию в качестве предопределенной языковой конструкции, для выполнения этой задачи можно написать процедуру или функцию, а затем вызвать ее из того программного модуля, где эта операция потребуется. Подобным же образом во многих языках программирования осуществляются и операции ввода/вывода, но за исключением того, что вызываемые процедуры и функции в конечном счете представляют собой подпрограммы, выполняемые операционной системой компьютера.

Например, чтобы ввести значение с клавиатуры и присвоить его переменной под именем Value, в языке Pascal необходимо использовать оператор

```
readln(Value);
```

Для вывода переменной Value на экран дисплея используется другой оператор:

```
writeln(Value);
```

Заметим, что синтаксис этих операторов не отличается от вызова процедуры со списком параметров.

Аналогично в языке С для выполнения операций ввода и вывода предназначены функции scanf и printf, соответственно. Эти функции используют параметры как для определения вводимых данных, так и для определения вида этих данных в напечатанном виде. Этот подход называется *форматированным вводом и выводом* (formatted I/O). Например, чтобы напечатать в отдельной строке значения переменных Value1 и Value2 в десятичном виде, в языке С можно использовать следующий оператор:

```
printf("%d %d \n", Value1, Value2);
```

Здесь строка в кавычках определяет формат данных, а остальные параметры задают выводимые значения. Каждая пара символов "%d" определяет позицию, которая должна быть заполнена очередным значением в десятичной нотации, задаваемым соответствующим параметром. Пара символов "\n" означает, что после вывода значений следует перейти на новую строку. Предположим, что значения переменных Age1 и Age2 равны 16 и 25, соответственно, и что выполняется оператор

```
printf("The ages are %d and %d. \n", Agel, Age2);
```

В этом случае на экран дисплея будет выведено следующее сообщение:

```
The ages are 16 and 25
```

Поскольку языки C++ и Java являются объектно-ориентированными, они трактуют операции ввода/вывода как передачу данных от объекта к объекту. В частности, в язык C++ встроены готовые объекты с именами cin и cout, предназначенные для представления стандартных устройств ввода (возможно, клавиатуры) и вывода (возможно, экрана дисплея), соответственно. Данные, которые вводятся с клавиатуры или выводятся на экран дисплея, передаются этим объектам или поступают от них в виде сообщений. Например, с помощью следующего оператора можно ввести с клавиатуры некоторое значение и присвоить его переменной Value:

```
cin >> Value;
```

Для того чтобы вывести значение переменной Value на экран, следует послать его на устройство вывода с помощью следующего оператора:

```
cout << Value;
```

Вопросы для самопроверки

1. Чем отличаются локальные переменные от глобальных?
2. Чем отличается процедура от функции?
3. Почему многие языки программирования реализуют операторы ввода/вывода так, будто они представляют собой вызов процедуры?
4. Чем отличаются формальные параметры от фактических?

5.4. РЕАЛИЗАЦИЯ ЯЗЫКА

В этом разделе мы познакомимся с процессом перевода программы с языка высокого уровня в такую форму, которая может быть выполнена машиной.

Процесс перевода. Процесс перевода программы с одного языка на другой называется *трансляцией* (translation). Программа в своем оригинальном виде называется *исходной программой* (sourse program), а оттранслированная ее версия называется *объектным кодом программы* (object program). Процесс трансляции состоит из трех этапов – лексического анализа, синтаксического разбора и генерации кода, которые выполняются элементами транслятора, называемыми *лексическим анализатором* (lexical analyzer), *синтаксическим анализатором* (parser) и *генератором кода* (code generator) (рис. 5.12).

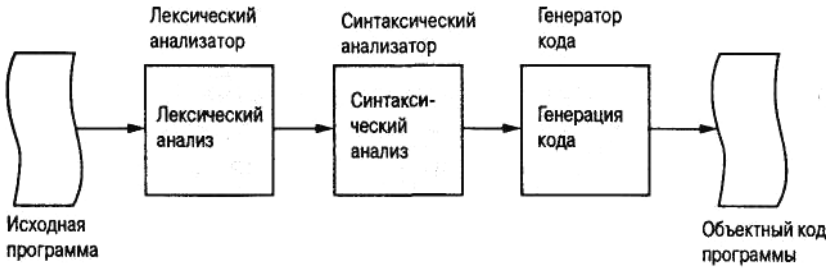


Рис. 5.12. Процесс трансляции программы

В процессе лексического анализа распознаются цепочки символов, которые представляют собой отдельные элементы. 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, 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 те отдельные буквы, из которых они состоят.

Таким образом, лексический анализатор символ за символом считывает текст исходной программы, определяя, какие группы символов образуют самостоятельные единицы текста. 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.

Реализация языка Java. При использовании анимированной Web-страницы управляющее анимацией программное обеспечение передается через Internet вместе с самой страницей. Если программное обеспечение представляет собой исходный текст программы, то при просмотре страницы возникает дополнительная задержка, связанная с трансляцией программы в соответствующий машинный язык. Однако если это программное обеспечение пересылать сразу на машинном языке, то может оказаться, что для разных типов машин потребуются различные версии программы.

Фирма Sun Microsystems решила эту проблему, разработав универсальный "машинный язык", названный *байт-кодом*, на который переводятся исходные тексты программ, написанные на языке Java. Хотя байт-код не является настоящим машинным языком, подходящий интерпретатор сможет выполнить его достаточно быстро. В настоящее время такие интерпретаторы уже стали стандартной частью любых Web-броузеров. Поэтому, если управляющее Web-страницей программное обеспечение написано на языке Java и переведено в байт-код, то эту байт-кодовую версию программы можно передавать любому броузеру, что гарантирует эффективную анимацию изображений при просмотре данной Web-страницы.

Из сказанного выше можно прийти к заключению, что синтаксический анализатор анализирует программу в терминах лексических единиц (лексем), а не отдельных символов. Задачей синтаксического анализатора является объединение этих единиц в операторы. Действительно, в процессе синтаксического анализа производится идентификация грамматической структуры программы и распознавание роли каждого ее компонента. Это – техническая сторона синтаксического разбора, которая может привести к некоторой задержке при чтении следующего предложения.

```
Человек, которого лошадь, проигравшая скачки, сбросила, не был ранен.
```

Чтобы упростить процесс синтаксического разбора, ранние языки программирования требовали, чтобы каждый оператор программы размещался в определенном месте бланка исходного текста. Такие языки называются *языками с фиксированным форматом* (fixed-format languages). 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 легко ее читать. В таких программах принято использовать отступы, чтобы помочь читателю понять внутреннюю структуру оператора. Например, рассмотрим следующий оператор:

```
if Cost < Cash on hand then pay with cash else use credit card
```

При использовании свободного формата программист может записать этот же оператор в более удобном виде:

```
if Cost < Cash on hand
   then pay with cash
   else use credit card
```

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* (key words), такие, как **if**, **then** и **else**, предназначенные для выделения начала отдельных фраз. Эти ключевые слова обычно называют *зарезервированными словами* (reserved words), чтобы подчеркнуть, что программист не может использовать их в своей программе для иных целей.

Процесс синтаксического анализа базируется на совокупности правил, определяющих синтаксис языка программирования. Один из способов представления этих правил состоит в использовании *синтаксических диаграмм* (syntax diagram), на-

глядно иллюстрирующих грамматическую структуру программы. На рис. 5.13 представлена синтаксическая диаграмма оператора **if-then-else** для нашего псевдокода, описанного в главе 4.

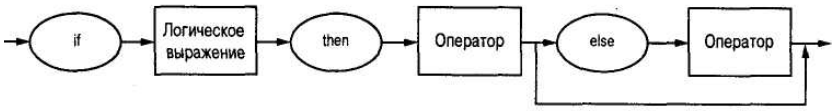


Рис. 5.13. Синтаксическая диаграмма оператора `if-then-else`

Эта диаграмма показывает, что структурно оператор **if-then-else** начинается с ключевого слова **if**, за которым следует `Логическое выражение`. Затем указывается ключевое слово **then**, после которого должен присутствовать выполняемый `Оператор`. Эта комбинация ключевых слов может дополняться (но необязательно) ключевым словом **else** и еще одним элементом `Оператор`. Обратите внимание, что члены выражения, или термы, которые действительно присутствуют в операторе **if-then-else**, заключены в овалы, а те термы, которые подлежат дальнейшему уточнению, например `Логическое выражение` или `Оператор`, помещены в прямоугольники. Термы, подлежащие дальнейшему уточнению (т.е. те, которые заключены в прямоугольники), называются *нетерминальными* (nonterminals), а термы, окруженные овалами, – *терминальными* (terminals). В полном описании синтаксиса языка программирования нетерминальные термы описываются дополнительными диаграммами.

На рис. 5.14 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 могут вначале следовать (но необязательно) символы `+` или `-`, а затем другой компонент `Выражение`. Вторая диаграмма описывает структуру компонента `Терм` как состоящую из отдельных символов `x`, `y` и `z` или другого компонента `Терм`, за которым следует один из символов `*` или `/`, а потом идет некоторый компонент `Выражение`.

Метод, с помощью которого отдельная строка программы проверяется на соответствие некоторой совокупности синтаксических диаграмм, можно наглядно представить с помощью *дерева синтаксического анализа* (parse tree) (рис. 5.15). На нем приведено дерево синтаксического анализа следующей строки:

```
x + y * 2
```

Это дерево построено на совокупности диаграмм, представленных на рис. 5.14. Заметим, 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.

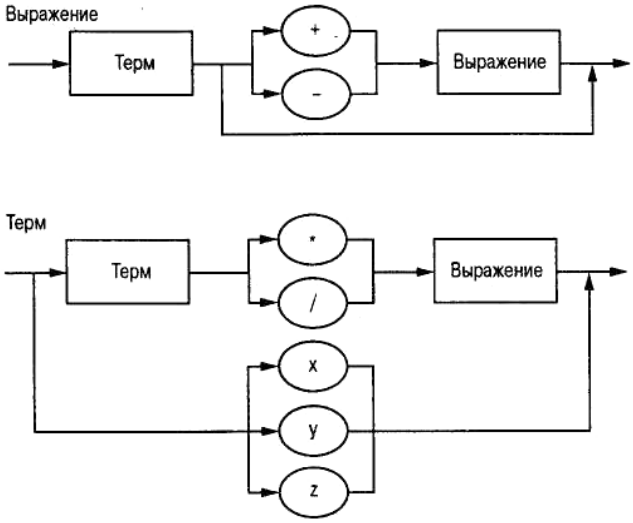


Рис. 5.14. Синтаксическая диаграмма, описывающая структуру простого алгебраического выражения

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. Такие неточности могут быть практически незаметными. И действительно, показанное на рис. 5.13 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. 5.16:

```
if B1 then if B2 then S1 else S2
```

Заметим, что эти интерпретации существенно отличаются друг от друга. 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 нашем псевдокоде этой проблемы можно избежать за счет применения скобок. В частности, этот оператор следовало бы записать следующим образом:

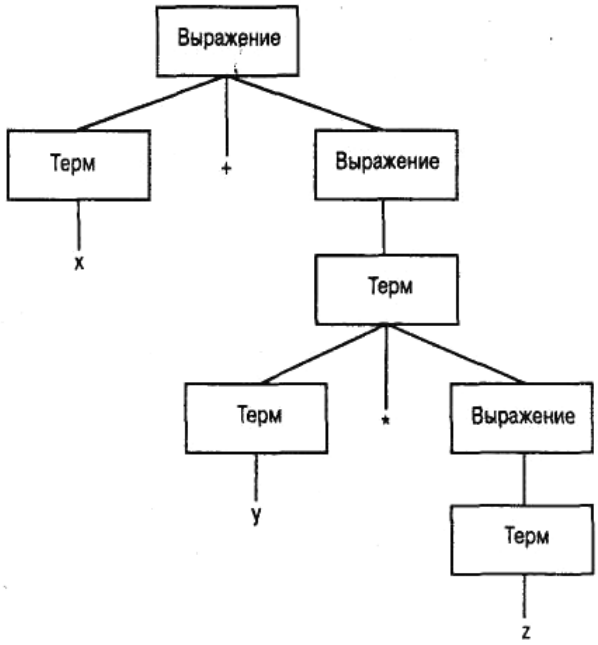


Рис. 5.15. Дерево синтаксического анализа строки `x + y * 2`

```
if B1
   then (if B2
            then SI)
   else S2
```

Возможен и другой вариант записи этого выражения:

```
if B1
   then (if B2
            then SI
            else S2)
```

В результате можно будет четко различать обе возможные интерпретации исходного оператора.

При синтаксическом анализе операторов объявления содержащаяся в них информация записывается в *таблицу символов* (symbol table). В результате таблица символов будет содержать информацию о том, какие переменные были описаны в программе и какие типы и структуры данных связаны с этими переменными. В дальнейшем синтаксический анализатор использует эту информацию при анализе выполняемых операторов. Ниже представлен один из таких операторов:

```
Total ← Price + Tax;
```

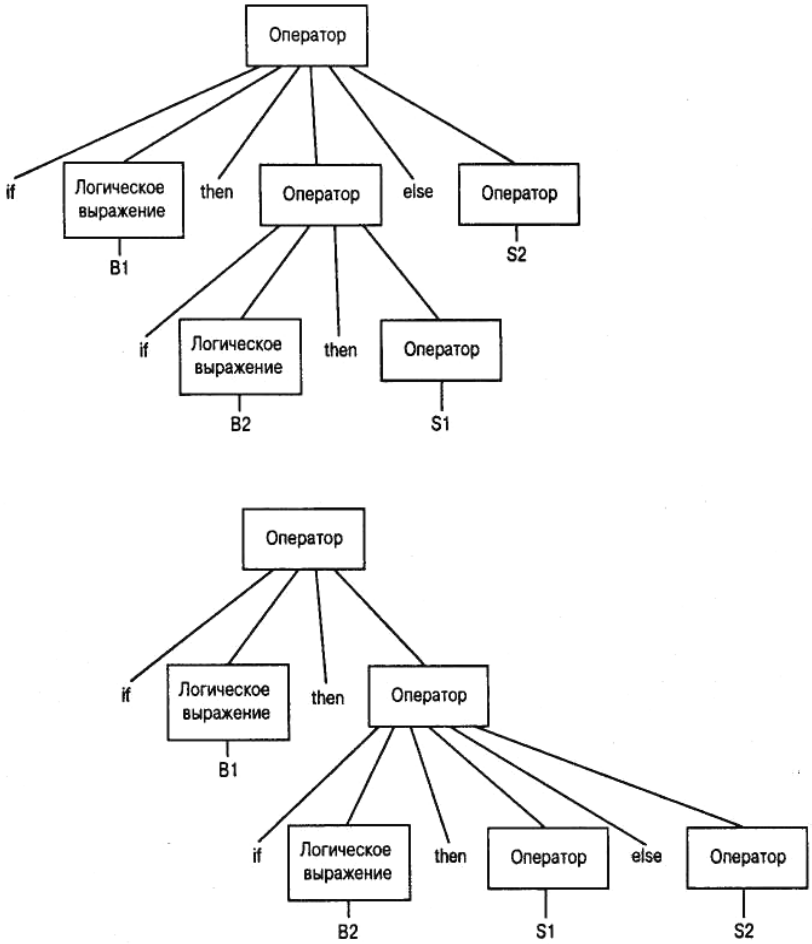


Рис. 5.16. Два варианта дерева синтаксического анализа оператора `if B1 then if B2 then S1 else S2`

Действительно, чтобы определить значение символа +, синтаксический анализатор должен знать, какой тип данных связан с переменными `Price` и `Tax`. Если переменная `Price` имеет тип real, а переменная `Tax` – тип character, то операция суммирования переменных `Price` и `Tax` не имеет смысла; ее появление должно рассматриваться как ошибка. Если обе переменные `Price` и `Tax` имеют тип integer, то синтаксический анализатор потребует от генератора кода создать на машинном языке команду с кодом операции сложения двух целых чисел. Если эти переменные имеют тип real, то синтаксический анализатор потребует использовать операцию сложения двух чисел с плавающей точкой.

Предыдущий оператор имеет смысл и в некоторых из тех случаев, когда входящие в него переменные имеют разный тип. Например, если переменная `Price` имеет тип integer, а переменная `Tax` – тип real, то вполне возможно применить операцию сложения. В этом случае синтаксический анализатор может предложить генератору кода предварительно преобразовать одну из переменных в другой тип и после этого выполнить требуемое сложение. Такое неявное преобразование типов называется *приведением типов* (coercion).

Неявное приведение типов не одобряется многими разработчиками языков. 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* (strongly typed), 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.

Последняя стадия процесса трансляции программы – это генерация кода, т.е. создание команд машинного языка, выполняющих выражения, распознанные синтаксическим анализатором. Этот процесс включает множество различных аспектов, один из которых – повышение эффективности генерируемого кода. Например, рассмотрим задачу трансляции последовательности из двух следующих операторов:

```
x ← y + z;
w ← x + z;
```

Эти операторы могут быть оттранслированы по отдельности. Однако это не позволит достичь высокой эффективности результата. 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 в построении программы называется *оптимизацией кода* (code optimization) и является важной задачей генератора кода.

Этапы лексического анализа, синтаксического анализа и генерации кода никогда не выполняются в указанной строгой 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.

Рис. 5.17. Объектно-ориентированный подход к процессу трансляции программы

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. 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. Поэтому исходный текст программы, лексический анализатор, синтаксический анализатор, генератор кода и объектный код могут рассматриваться как объекты, которые взаимодействуют, посылая друг другу сообщения по мере выполнения своих функций (рис. 5.17).

Связывание и загрузка. Объектный код программы, полученный в результате ее трансляции, хотя и записан на машинном языке, но чаще всего еще не имеет той формы, которая необходима для выполнения машиной. Одной из причин этого является то, что многие средства программирования позволяют разрабатывать программы в виде отдельных модулей, транслируемых в разное время (это способствует приданию программному обеспечению модульной структуры). Поэтому объектный код, полученный в результате отдельного процесса трансляции, чаще всего представляет собой всего лишь некоторую составную часть программы, которая должна быть связана с другими ее частями для решения задач, стоящих перед всей системой в целом. Даже если вся программа разработана и оттранслирована в виде единственного модуля, ее объектный код все же не готов для выполнения машиной, поскольку любой программе, как правило, требуются функции, выполняемые другими программами или же операционной системой. Поэтому объектный код программы в действительности представляет собой программу на машинном языке, обычно содержащую несколько неразрешенных ссылок; эту программу необходимо связать с другими программами, прежде чем можно будет получить полноценный выполняемый модуль.

Связывание объектного кода программы с другими модулями выполняет программа, называемая *редактором связей* (linker). Ее задача – связать между собой несколько объектных модулей (полученных в результате предшествующих независимо выполненных процедур трансляций), программ операционной системы и другое программное обеспечение в целях получения завершенного выполняемого модуля (иногда называемого *загрузочным модулем*), который представляет собой файл, размещаемый во внешней памяти машины.

Наконец, чтобы выполнить оттранслированную программу, ее загрузочный модуль должен быть размещен в основной памяти машины с помощью специальной программы, называемой *загрузчиком* (loader). Загрузчик обычно является частью программы-планировщика операционной системы (см. раздел 3.3). Важность этого этапа особенно велика в многозадачных системах. В подобных системах любая программа вынуждена использовать память совместно с другими параллельно выполняемыми процессами, причем набор параллельно выполняемых процессов изменяется при каждом выполнении программы. Поэтому точное расположение выделяемого для некоторой программы участка памяти остается неизвестным, вплоть до ее вызова на выполнение. Таким образом, задачей загрузчика является считывание программы в указанную операционной системой область памяти и выполнение в ней всех необходимых настроек, которые невозможно осуществить, пока не будет известно точное расположение данной программы в памяти. (Команды перехода в программе должны выполнять переход на правильные адреса внутри программы.) Желание минимизировать объем выполняемой загрузчиком окончательной настройки привело к разработке таких способов программирования, в которых запрещается использовать явные ссылки на адреса памяти в программе. В результате появились *перемещаемые модули* (relocatable module), которые выполняются правильно независимо от их размещения в памяти.

Итак, подготовка программы на языке программирования высокого уровня состоит из трех последовательных этапов: трансляции, связывания и загрузки, как показано на рис. 5.18. После выполнения трансляции и связывания программу можно повторно загружать и выполнять без обращения к ее исходному тексту. Однако если в программу потребуется внести изменения, то их придется ввести в исходный текст, а затем вновь оттранслировать и заново связать модифицированную версию исходного текста программы, чтобы создать новый вариант ее загрузочного модуля.

Рис. 5.18. Полный цикл подготовки программы к выполнению

Пакеты разработки программного обеспечения. В настоящее время прочно укрепилась тенденция объединять транслятор с другими элементами программного обеспечения в общий пакет, функционирующий как единая интегрированная 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 про-

граммы на машинный язык, и разнообразным инструментам отладки программ, позволяющим отслеживать выполнение неправильно работающих программ и обнаруживать имеющиеся в них ошибки.

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 обычно настроены для работы с тем языком программирования, который используется в данном пакете. Например, текстовые редакторы в пакетах 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.

Во многих пакетах для разработки программ используют графический интерфейс, что позволяет программистам создавать программы из заранее написанных блоков, представленных на экране пиктограммами. Отдельные блоки могут дополнительно настраиваться в среде текстового редактора. Разработка подобных пакетов отражает общую тенденцию к созданию программ из больших, заранее заготовленных блоков, вместо обычного покомандного программирования.

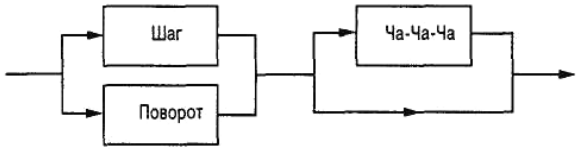
Вопросы для самопроверки

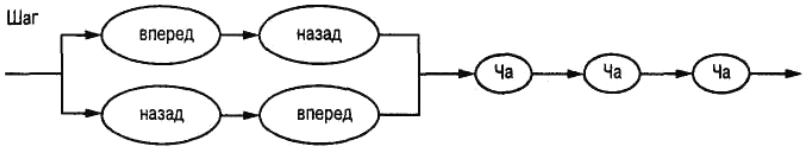
1. Опишите три основных этапа процесса трансляции.
2. Что такое таблица символов?
3. Исходя из синтаксических диаграмм, представленных на рис. 5.14, нарисуйте дерево синтаксического разбора для выражения

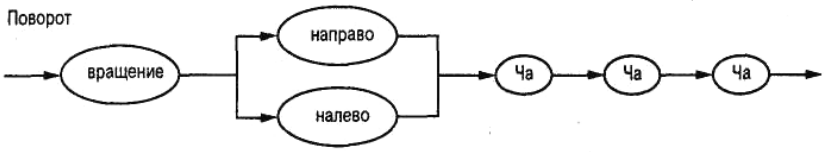
```
x * y + x + z.
```

4. Напишите несколько строк, формат которых будет соответствовать структуре `Ча-Ча-Ча`, определенной с помощью следующих синтаксических диаграмм.

Ча-Ча-Ча







5.5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ*

В разделе 5.1 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. Эти внутренние процедуры называются *методами* (или *функциями-членами* – в терминологии языка C++). 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 соответствии с этим, объектно-ориентированные языки программирования предоставляют операторы и другие средства для реализации этих идей. Некоторые из них мы рассмотрим в данном разделе.

Классы и объекты. Рассмотрим процесс разработки программы, моделирующей функционирование малого предприятия. Под малым предприятием подразумевается частное предприятие, на котором работает не более 25 сотрудников. Наша задача – создать программу для исследования влияния, которое могут оказывать на подобные предприятия различные изменения, происходящие в экономике. Следовательно, необходимо разработать программу, которая будет моделировать деятельность нескольких таких предприятий и отслеживать, как они взаимодействуют друг с другом и внешним окружением. Используя объектно-ориентированный язык, мы могли бы описать каждое такое предприятие как объект с внутренними методами, определяющими его реакцию на воздействия внешнего мира.

В языках C++ и Java для объявления, что имя `BusinessX` будет использоваться для ссылки на объект "типа" `Small-`

Business, мы могли бы применить следующий оператор:

```
SmallBusiness  BusinessX;
```

Обратите внимание, что этот оператор ничем не отличается от используемого в языке программирования C для объявления, что Cost является переменной типа integer:

```
int Cost;
```

Несмотря на общее сходство, эти операторы имеют несколько существенных отличий. Одно из них состоит в том, что "тип" объекта принято называть классом. Таким образом, класс – это описание структуры объекта. В некотором смысле класс представляет собой шаблон, который используется для создания объекта. Другое отличие заключается в том, что тип integer является встроенным типом языка программирования, тогда как класс SmallBusiness – это "тип", который должен быть описан в каком-то месте программы.

Для описания классов многие объектно-ориентированные языки программирования включают специальные операторы, подобные представленному ниже:

```
class SmallBusiness
      {
        .
        .
        .
      };
```

Первая строка этого объявления означает, что блок операторов внутри фигурных скобок определяет класс под названием Small-Business. Внутри фигурных скобок объявляются разнообразные методы, описывающие, как объект типа SmallBusiness должен реагировать на различные внешние воздействия. Объявления методов напоминают описание обычных процедур и функций.

Обобщая сказанное, можно сделать вывод, что в нашей моделирующей программе сначала следует воспользоваться оператором class для описания класса с именем SmallBusiness, а затем поместить операторы объявления приведенного выше типа для указания, что в программе будут использоваться объекты.

Наследование. К сожалению, многие малые предприятия, работу которых мы собираемся моделировать, имеют разные особенности. Предприятие, принимающее заказы на доставку товаров по почте, должно отвечать на поступающие заявки и пополнять свой запас товаров на складе. Тогда как небольшое предприятие, занимающееся консалтингом, взаимодействует со своими клиентами совершенно иным образом. Однако между этими предприятиями достаточно много общего. Например, все они подчиняются одним и тем же законам налогообложения, одинаково обрабатывают платежные ведомости и сходным образом пополняют запасы канцелярских товаров.

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* (inheritance). Предположим, что для разработки нашей программы моделирования будет использоваться язык Java. В этом случае сначала можно было бы применить приведенный выше оператор объявления для описания класса SmallBusiness, содержащего те методы, которые являются общими для всех малых предприятий в данном приложении.

А затем использовать приведенный ниже оператор для описания другого класса, например, с именем MailOrderBusiness:

```
class MailOrderBusiness extends SmallBusiness
      {
        .
        .
        .
      }
```

В языке C++ нужно просто заменить слово extends двоеточием. Этот оператор означает, что данный класс наследует все свойства класса SmallBusiness, а также дополнительно имеет свойства, указанные внутри фигурных скобок. Аналогично можно было бы описать и другой класс под названием ConsultingBusiness, который также наследует свойства класса SmallBusiness и одновременно обладает свойствами, характерными только для предприятий, занимающихся консалтингом. После того как указанные выше классы будут описаны, появится возможность использовать следующие операторы:

```
MailOrderBusiness BusinessX;
ConsultingBusiness BusinessY;
```

Эти операторы указывают, что переменная BusinessX является объектом, описывающим малое предприятие, принимающее заказы по почте, а переменная BusinessY – это объект, описывающий малое предприятие, занимающееся консалтингом.

Инкапсуляция и полиморфизм. Существование множества объектов, имеющих сходные, но все же отличающиеся характеристики, приводит к явлению, напоминающему перегрузку, речь о которой шла в разделе 5.2. (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 посылаем сообщение "нарисуй себя" каждому из объектов, образующих это изображение. Однако программы, рисующие отдельные объек-

ты, 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* (polymorphism), а соответствующее сообщение – полиморфным.

Другим свойством, связанным с объектно-ориентированным программированием, является *инкапсуляция* (encapsulation), 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). Например, рассмотрим объект нашей программы для моделирования бизнеса, имеющий "тип" `HailOrderBusiness`. Следует ожидать, что этот объект будет содержать метод, описывающий его реакцию на поступление заказа. Прочим объектам потребуется обращаться к данному методу в целях передачи некоторого заказа. Следовательно, доступ к этому методу должен быть открытым. Однако детальные сведения о том, как именно данное предприятие выполняет заказ, должны быть доступны только внутри объекта. Это означает, что все эти детали должны быть закрытыми. Большинство объектно-ориентированных языков программирования позволяет программисту определять непосредственно в описании класса, какие части объекта являются открытыми, а какие – закрытыми. Например, описание класса `MailOrderBusiness` на языке Java может выглядеть следующим образом:

```
class MailOrderBusiness extends SmallBusiness
      {
        public  ...
        private ...
        private ...
        public  ...
      }
```

Здесь ключевые слова `public` и `private` в начале описания каждого компонента используются для того, чтобы указать вид доступа к ним.

Вопросы для самопроверки

1. В чем заключается отличие между объектом и классом?
2. Предположим, что классы `PartTimeEmployee` (Временный работник) и `FullTimeEmployee` (Постоянный работник) наследуют свойства класса `Employee` (Работник). Укажите, какими свойствами, по Вашему мнению, должен обладать каждый из этих классов.
3. Что такое инкапсуляция?

5.6. ПРОГРАММИРОВАНИЕ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ*

Предположим, что нам поручили разработать программу для анимации объектов в компьютерной игре, включающей в себя бегущее стадо антилоп, марширующую колонну муравьев или, например, 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 каждой антилопы, которое (если требуется достаточно реалистичная анимация) предполагало бы, что программа должна непрерывно вычислять индивидуальные параметры для каждого из ста животных. Альтернативный подход состоит в разработке программы, управляющей анимацией отдельной антилопы, характеристики которой определяются параметрами, задаваемыми при запуске этой программы. Теперь анимацию на всем экране можно было бы создать посредством выполнения ста запусков этой программы и использования в каждом случае собственного набора параметров. Выполнив эти запуски одновременно, можно получить иллюзию, что сто отдельных антилоп бегут на экране в одно и то же время. (Хотя, строго говоря, одновременное выполнение ста запусков такой программы и превосходит возможности современных компьютеров, тем не менее, пять запусков программы, имитирующей атаку космического корабля, не должны составить проблемы; на практике этот прием применяется довольно часто.)

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 (этот вопрос обсуждался в главе 3).

Языки программирования для написания программ, выполняющих параллельную обработку данных, первоначально предназначались для разработки операционных систем. Однако, как показывает приведенный в начале раздела пример, многие современные компьютерные приложения легче реализовать с использованием 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.

Анимация в кинофильмах. Конечно, было бы неверным связывать начало современной анимации с одним-единственным кинофильмом, однако многие считают фильм *Звездные войны* (Lucasfilm, 1977) началом современной волны компьютерной анимации в кинематографе. В настоящее время технология, использовавшаяся при создании таких классических мультфильмов Уолта Диснея, как *Спящая красавица* и *Фантазия,* и предусматривающая создание многочисленных, сделанных вручную кадров, заменена технологиями компьютерной анимации, требующими применения современных многопроцессорных компьютерных систем. Кинозритель может согласиться, что примеры, описанные в начале этого раздела (бегущие антилопы и марширующие полчища муравьев), вовсе не являются надуманными. Это – примеры из реальных кинофильмов, которые созданы с использованием компьютерных технологий (хотя и не обязательно так, как это описано в тексте, поскольку, в отличие от компьютерных игр, анимация в кино-

фильмах может быть разработана прежде, чем будет выполнена). Существуют ссылки на Web-страницы, посвященные анимации в кинофильмах. В частности, заинтересованный читатель может посетить следующие Web-узлы: http://www.lucasfilm.com, http://www.disney.go.com, http://www.pixar.com и http://www.pdi.com. Особенный интерес представляют Web-узлы http://www.antz.com и http://www.dreamworksgames.com.

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 одновременно сто запусков программы анимации движения антилопы, нам потребуется синтаксическая конструкция, позволяющая выполнить эти действия. Запуск новых процессов обычно выполняется способом, очень напоминающим традиционный вызов процедуры. Разница лишь в том, что в традиционном способе вызывающий процедуру программный модуль приостанавливает свою работу, пока вызываемая процедура не закончит свое выполнение, тогда как при параллельной обработке вызывающий программный модуль продолжает свою работу одновременно с вызванной процедурой. Таким образом, чтобы воспроизвести бег ста антилоп на экране, нам нужно написать основную программу, которая просто генерирует сто запусков программы, имитирующей бег антилопы; причем каждый запуск выполняется со своим набором параметров, описывающим отличия одной антилопы от другой.

Более сложный вопрос связан с параллельной обработкой данных, которая предусматривает обмен данными между процессами. Например, при имитации бега антилоп вычислительные процессы, соответствующие отдельным антилопам, должны согласовывать между собой их взаимное расположение, что необходимо для координации движения отдельных антилоп. В случае компьютерной игры процесс, управляющий полетом ракеты, должен быть связан с процессом, управляющим другими объектами на экране, например, чтобы определить, какой из существующих объектов следует взорвать. В иных случаях один процесс должен ожидать, пока другой не достигнет определенной точки в своих вычислениях. Кроме того, один процесс, выполнив собственную задачу, может остановить другой процесс.

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, выработанные учеными. 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. 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 решения этой проблемы заключается в описании подобных процессов в программе таким образом, что возможность их доступа к совместно используемым данным исключается до тех пор, пока работа с ними не станет безопасной. (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 ошибка в описании любого из процессов, имеющих доступ к совместно используемым данным, способна вызвать повреждение всей системы в целом. По этой причине многие считают, что лучше объединять сами данные и средства контроля доступа к ним в единое целое. Такой управляющий механизм часто называют *монитором* (data monitor). Коротко говоря, в этом случае ответственность за исключение множественного доступа к данным налагается не на процесс, который стремится получить доступ, а на те данные, к которым необходим доступ. В результате контроль над доступом к данным сосредоточивается в одном месте программы, а не распыляется среди множества ее сегментов[3].

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.

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 применения, нуждающихся 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.

Вопросы для самопроверки

1. Какие свойства характерны для языков программирования, поддерживающих параллельную обработку данных, и какие характеристики отсутствуют в традиционных языках программирования?

2. Опишите два метода организации взаимно исключающего доступа к данным.

---

[3] Эта задача имеет вполне естественное решение в объектно-ориентированных языках программирования, где и данные, и получающие к ним доступ процессы собраны в пределах одного объекта. В частности, для читателя может представлять определенный интерес использование команды `synchronize` в языке Java.

5.7. ДЕКЛАРАТИВНОЕ ПРОГРАММИРОВАНИЕ*

Выше мы утверждали, 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*.

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

означает "Кермит – принц или мисс Пигги – актриса", а выражение

B AND ¬A

Означает: "Мисс Пигги – актриса, а Кермит – не принц". Для обозначения отношения "следует" мы будем использовать стрелку. Например, выражение

$A \rightarrow B$

означает: "Если Кермит – принц, то мисс Пигги – актриса".

В общем виде принцип резолюции утверждает, что из двух высказываний

P OR *Q* и *R* OR ¬*Q*

мы можем вывести высказывание

R OR Q.

В этом случае мы говорим, что два исходных высказывания сводятся к третьему, которое называется *резольвентой*. Важно подчеркнуть, что резольвента – это логическое следствие исходных высказываний. Если исходные высказывания истинны, то резольвента также должна быть истинной. (Если Q – истинно, то R должно быть истинно, 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. 5.19, где 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

$P \rightarrow Q$

это не так. Эта потенциальная проблема не очень серьезна. 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

$P \rightarrow Q$

эквивалентно высказыванию в форме предложения

Q OR ¬*P*.

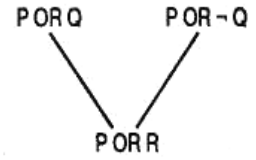


Рис. 5.19. Резолюция высказываний (P OR Q) и (R OR ¬Q) с получением высказывания (P OR R)

Совокупность высказываний является *противоречивой*, если все они не могут быть истинными одновременно. Другими словами, такая совокупность высказываний состоит из противоречащих друг другу 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. 5.20 показано, что совокупность высказываний

P OR *Q* R OR ¬Q ¬R ¬Q

является противоречивой.

Предположим, нам необходимо проверить, что из некоторой совокупности высказываний действительно следует высказывание P. Высказывание P эквивалентно отрицанию высказывания -P. Следовательно, чтобы показать, что из исходной совокупности высказываний следует высказывание P, нам нужно лишь применять процедуру резолюции к исходным высказываниям и высказыванию ¬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 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).

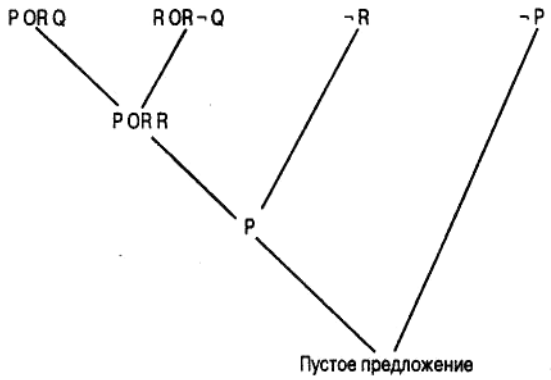


Рис. 5.20. Резолюция высказываний (P OR Q), (R OR ¬Q), ¬R и ¬Q

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 компоненты (Маша находится дома) и ¬ (Маша находится в X) почти противоречат друг другу. Задача заключается в том, чтобы установить истинность высказывания (Маша находится в X), которое является высказыванием о местонахождении вообще 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, присвоение значения "дом" переменной X), который делает возможным выполнение резолюции, называется *унификацией.* Это процесс, который позволяет применять общие высказывания к частным приложениям в дедуктивной системе.

Язык Prolog. Prolog (PROgramming in LOGic – программирование в логике) – это декларативный язык программирования, базовый алгоритм решения задач которого основан на методе повторной резолюции. 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 константы и переменные и требует, чтобы константы начинались со строчных букв, а переменные – с прописных.

Операторы в языке Prolog являются либо фактами, либо правилами. Каждый из операторов заканчивается точкой. Факт состоит из отдельного предиката. Например, тот факт, что черепаха (turtle) двигается быстрее улитки (snail), выражается оператором языка Prolog следующего вида:

```
faster(turtle,snail).
```

А тот факт, что кролик (rabbit) бегает быстрее черепахи, выражается оператором

```
faster(rabbit,turtle).
```

Правило в языке Prolog – это оператор импликации (следования). Вместо записи оператора в виде X → Y программист на языке Prolog пишет "Y, если X", используя вместо слова "если" символы : –. Таким образом, правило "из того, что X быстрее (faster) Y, а Y быстрее Z, следует, что X быстрее Z" может быть выражено специалистом-логиком в следующем виде:

```
(faster(X,Y) AND faster (Y,Z)) → faster(X,Z)
```

На языке Prolog это же правило выражается следующим образом:

```
faster(X,Z) :- faster(X,Y), faster(Y,Z).
```

Запятая, разделяющая термы faster(X,Y) и faster(Y,Z), представляет 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.

Большинство реализаций языка Prolog разработано для интерактивного применения. В этом смысле задача программиста – разработать совокупность фактов и правил, которые образуют множество исходных высказываний, используемых затем в дедуктивной системе. Установив такую совокупность высказываний, можно ввести с клавиатуры предложение (называемое в терминологии языка Prolog 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 предложения

```
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. Далее, 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).
```

Поэтому, если мы поставим задачу

```
faster{V, W),
```

то программа 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.

Вопросы для самопроверки

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. Предположим, что программа на языке Prolog состоит из операторов, выражающих отношение "экономнее" (thriftier):

```
thriftier(carol, john).
thriftier(bill, sue).
thriftier(sue, carol).
thriftier(X,Z) :- thriftier(x, Y), thriftier(Y,Z).
```

Перечислите результаты, которые можно получить для каждой из следующих целей:

а) `thriftier (sue, V);`

б) `thriftier(U, carol);`

в) `thriftier(U, V).`

Упражнения

(Упражнения, 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, описанный в приложении B.

```
x ← 0;
while (x ≠ 3) do
    (x ← x+1)
```

3. Переведите приведенный ниже оператор на машинный язык, описанный в приложении B, предполагая, что переменные `Length`, `Width` и `Halfway` представлены как числа с плавающей точкой:

```
Halfway ← Length + Width
```

4. Переведите следующий оператор языка высокого уровня на машинный язык, описанный в приложении B, предполагая, что числа `W`, `X`, `Y` и `Z` представлены в двоичном коде и занимают в основной памяти один байт:

```
if(X equals 0)
```

```
    then Z ← Y + W
    else Z ← Y + X
```

5. Для чего при трансляции операторов необходимо указывать тип данных, связанный с переменными в задаче 4? 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 "`abcd`", 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, описанной в приложении B.

12. Некий Джон Программер утверждает, что возможность объявления констант в программе является излишней, поскольку вместо них можно использовать переменные. Например, в разделе 5.2 можно описать переменную `AirportAlt`, а потом присвоить ей нужное значение в начале программы. Почему это решение хуже, чем вариант с использованием константы?

13. Опишите отличия, существующие между операторами объявления и выполняемыми операторами.

14. Объясните разницу между литералом, константой и переменной.

15. Что такое приоритет оператора?

16. Что такое структурное программирование?

17. Чем отличается смысл символа равенства в операторе

```
if(X = 5) then (...)
```

от смысла этого же символа в следующем операторе присваивания?

```
X = Z + Y
```

18. Начертите блок-схему структуры, выраженной следующими операторами языков C++ и Java:

```
for(x = 0; x < 8; ++x)
{…}
```

19. Начертите блок-схему структуры, выраженной следующими операторами языков C, C++ и Java:

```
switch(suit)
{  case "clubs": bid(1): break;
       case "diamonds": bid(2): break;
       case "hearts": bid(3): break;
       case "spades": bid(4): break;
    }
```

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 операторы `for`, представленные на рис. 5.8?

21. Перепишите следующий фрагмент программы, используя один оператор **`case`** вместо серии вложенных операторов **`if-then-else`**.

```
if (W = 5)
     then (Z ← 7)
     else (if (W = 6)
                then (Y ← 7)
                else (if (W = 7)
                          then (X ← 7) ) )
```

22. Выразите приведенную ниже запутанную последовательность операторов с помощью единственного оператора **`if-then-else`**.

```
       if X > 5 then goto 80
       X = X + 1
       goto 90
    80 X = X + 2
    90 ...
```

23. Опишите отличия между транслятором и интерпретатором.

24. Предположим, что переменная `X` в программе описана как переменная типа `integer`. Какая ошибка будет обнаружена транслятором при выполнении следующего оператора?

```
X ← 2.5
```

25. Что означает выражение "язык программирования со строгой типизацией"?

26. Почему большой массив не всегда можно передать в вызываемую процедуру по значению?

27. Предположим, что процедура modify определена следующим образом:

```
procedure modify(Y)
Y ← 7;
```

```
напечатать значение переменной Y;
```

Если параметры передаются по значению, что будет напечатано при выполнении следующего фрагмента программы? Что будет напечатано, если параметры передаются по ссылке?

```
X ← 5;
apply modify to X;
напечатать значение переменной X;
```

28. Предположим, что процедура `modify` определена следующим образом:

```
procedure modify(Y)
Y ← 9;
напечатать значение переменной X;
напечатать значение переменной Y;
```

Допустим также, что `X` – это глобальная переменная. Если параметры передаются по значению, что будет напечатано при выполнении следующего фрагмента программы? Что будет напечатано, если параметры передаются по ссылке?

```
X ← 5;
apply modify to X;
напечатать значение переменной X;
```

29. 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 упражнения 28, если параметры передаются по значению-результату?

30. Какая неоднозначность кроется в следующем операторе?

```
X ← 3 + 2 * 5
```

31. Предположим, что небольшая компания имеет пять сотрудников и планирует увеличить их число до шести. Ниже приведены фрагменты из двух эквивалентных программ (используемых компанией), которые нужно изменить, чтобы отобразить изменение количества сотрудников. Обе программы написаны на языке, напоминающем Pascal. Укажите, какие изменения следует произвести в каждой программе. Какие осложнения возникают в первой программе и не возникают при использовании констант во второй программе?

Программа 1

```
   .
   .
   .
DailySalary := TotalSal / 5;
AvgSalary := TotalSal / 5;
DailySales := TotalSales / 5;
AvgSales := TotalSales / 5;
   .
   .
   .
```

Программа 2

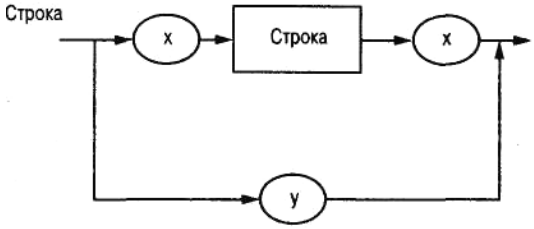
```
   .
   .
   .
const
   NumEmpl = 5;
   DayWk= 5;
   .
   .
   .
DailySalary: = TotalSal / DayWk;
AvgSalary := TotalSal / NumEpml;
DailySales := TotalSales / DayWk;
AvgSales := TotalSales / NumEmpl;
   .
   .
   .
```

32. Нарисуйте синтаксическую диаграмму, представляющую структуру оператора **`while`** из псевдокода, описанного в главе 4.

33. Разработайте совокупность синтаксических диаграмм для описания синтаксиса телефонных номеров, написанных в формате (044) 555-1234.

34. Разработайте совокупность синтаксических диаграмм для описания простого предложения на английском языке.

35. Напишите предложение, описывающее структуру строки, в соответствии с приведенной ниже синтаксической диаграммой. Затем нарисуйте дерево синтаксического анализа строки `xxyxx`.



36. Добавьте синтаксические диаграммы к диаграммам из вопроса 4 в разделе 5.4, чтобы получить совокупность диаграмм, определяющих структуру Танец, которая может быть структурой Ча-ча-ча или Вальс, где Вальс состоит из одной или более копий типа

вперед по_диагонали каданс

или

назад по_диагонали каданс.

37. Нарисуйте дерево синтаксического анализа приведенного ниже выражения, используя синтаксические диаграммы, представленные на рис. 5.14.

x * y + y / x

38. Какую оптимизацию кода может выполнить генератор кода при создании машинного кода для следующего оператора?

if (X = 5) **then** (Z ← X + 2) **else** (Z ← X + 4)

39. Упростите следующий фрагмент программы:

Y ← 5;**if** (Y = 7) **then** (Z ← 8) **else** (Z ← 9)

40. Упростите следующий фрагмент программы:

while(X ≠ to 5) do (X ← 5)

*41. Нарисуйте диаграмму (подобную диаграмме, показанной на рис. 5.20), представляющую последовательность операций резолюции, необходимых для того, чтобы показать, что совокупность высказываний (Q OR ¬R), (T OR R), ¬P, (P OR ¬T) и (R OR ¬P) противоречива.

42*. Является ли совокупность высказываний ¬R, (T OR R), (P OR ¬Q), (Q OR ¬T) и (R OR ¬P) противоречивой? Обоснуйте свой ответ.

43*. К каким выводам может прийти программа на языке Prolog, если поставить ей цель

bigger(X, Lassie).

Исходными высказываниями являются следующие:

bigger(rex, lessie).bigger(fido, rex).bigger(spot, rex).bigger(X,Z) :- bigger(X,Y), bigger(Y,Z).

44*. К каким выводам придет программа на языке Prolog, если поставить ей цель

eq(X, У.) .

Исходными высказываниями являются следующие:

qrteq(a b).qrteq(b, c).qrteq(c, a).qrteq(U, W :- qrteq(U, V), qrteq(V, W).eq(X, У) :- qrteq(X, Y), qrteq(Y, X).

45*. Какие проблемы могут возникнуть при выполнении машиной следующего фрагмента программы (описанной в разделе 1.7), в которой числа хранятся в 8-битовом формате с плавающей точкой?

X ← 0.01;**while**(X ≠ 1.00) **do** (вывести значение переменной X; X ← X + 0.01)

Ответы на вопросы для самопроверки

Раздел 5.1

1. Программа на языке третьего поколения является машинно-независимой в том смысле, что ее команды не содержат машинные атрибуты, такие, как номера регистров и машинные адреса ячеек памяти. Кроме того, она является 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 упор на описании компонентов предметной области задачи.

4. Языки третьего поколения позволяют в значительной степени формулировать текст программы непосредственно в терминах предметной области задачи, а не в терминах компьютерной тарабарщины, как это требовалось в языках предыдущих поколений.

Раздел 5.2

1. Использование описательных констант позволяет улучшить читабельность программы.

2. Оператор объявления описывает терминологию, тогда как исполняемый оператор описывает этапы выполнения алгоритма.

3. Целый, действительный, символьный и логический (булев).

4. Структура **if-then-else** и структура цикла **while** являются наиболее распространенными.

5. Все элементы однородного массива имеют один и тот же тип.

Раздел 5.3

1. Локальная переменная доступна только внутри программного модуля, например процедуры; глобальная переменная доступна в любом месте программы.

2. Функция – это процедура, возвращающая значение, связываемое с именем этой процедуры.

3. Потому что именно таковыми они и являются. Операторы ввода/вывода действительно вызывают подпрограммы той операционной системы, которая установлена на компьютере.

4. Формальный параметр – это идентификатор внутри процедуры. Он служит в качестве обозначения для значения переменной, т.е. действительного параметра, который передается процедуре при ее вызове.

Раздел 5.4

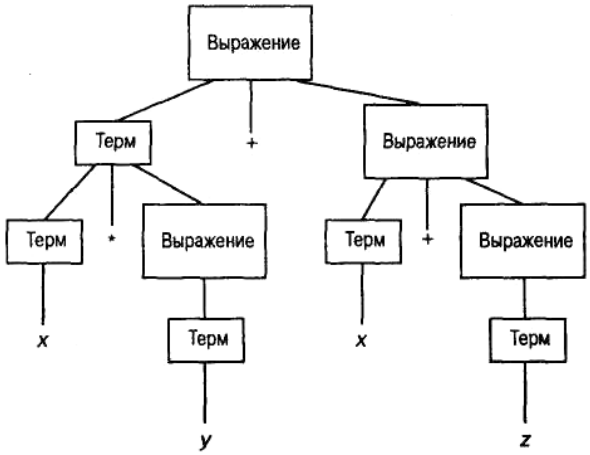
1. Лексический анализ – это процесс идентификации лексем.

Синтаксический разбор – это распознавание грамматической структуры программы.

Генерация кода – это создание команд объектного кода программы.

2. Таблица символов – это запись информации, которую синтаксический анализатор извлекает из объявлений переменных в программе.

3. В действительности грамматика неоднозначна. Здесь приведен один из возможных ответов. Можете ли вы привести другой? (Подсказка. Начните рассмотрение элемента `Выражение` как единственного элемента `Терм`.)



4. Вот несколько примеров подстрок: forward backward cha cha cha backward forward cha cha cha swing right cha cha cha swing left cha cha cha.

Раздел 5.5

1. Класс – это описание объекта.

2. Класс Employee может содержать информацию об имени сотрудника, его адресе, стаже и т.п.; класс FullTimeEmployee – сведения о пенсионных выплатах; класс PartTimeEmployee – информацию о количестве рабочих часов, отработанных за неделю, о почасовой ставке и т.п.

3. Инкапсуляция – это ограничение доступа. Инкапсулировать свойства объекта значит ограничить доступ к ним только самим этим объектом.

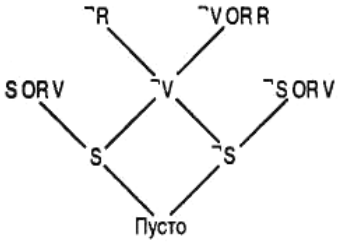
Раздел 5.6

1. Список может включать способы инициирования выполнения параллельных процессов и методы реализации обмена данными между ними.

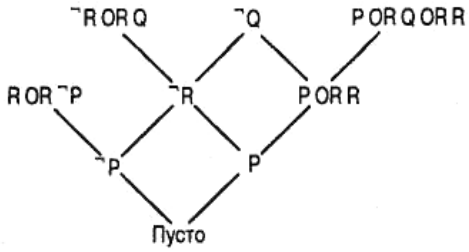
2. Первые делают упор на процессах, а вторые – на данных. Вторые имеют преимущество, заключающееся в концентрации задачи в единственной точке программы.

Раздел 5.7

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. а)

```
thriftier (sue, carol)
thriftier (sue, John)
```

б)

```
thriftier (sue, carol)
thriftier (bill, carol)
```

в)

```
thriftier (carol, John)
thriftier(bill, sue)
thriftier(sue, carol)
thriftier(bill, sue)
thriftier(sue, John)
```

6. ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

В этой главе мы обсудим процесс разработки и эксплуатации больших систем программного обеспечения. При создании таких систем возникают трудности, связанные с решением несколько иных задач, чем при написании простой программы. Например, их разработка требует усилий многих людей в течение длительного периода времени, за который могут измениться и требования системы, и персонал, работающий над этим проектом. Поэтому разработка программного обеспечения включает в себя вопросы, связанные с руководством кадрами и управлением проектом, которые скорее относятся к такой дисциплине, как управление предприятием, а не к вычислительной технике. Мы же сосредоточим наше внимание на вопросах, касающихся только вычислительной техники.

6.1. 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 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. Однако в этих рассуждениях не учитываются различия в свойствах программного обеспечения и объектов проектирования из других областей науки и техники.

Ассоциация по вычислительной технике. Основанная в 1947 г. 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. Web-узел этой ассоциации размещен по адресу http://www.acm.org/ constitution/code/html.

Одно из этих различий связано с возможностью создания 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 мотор, радиоприемник, кондиционер или дверные замки. Вместо этого он может воспользоваться уже созданными версиями необходимых ему компонентов. Однако в области программного обеспечения ранее разработанные компоненты, как правило, не являются универсальными, т.е. их внутренняя конструкция зависит от характерных особенностей конкретного приложения. Следовательно, попытка повторно использовать подобный компонент потребует его перепроектирования. В результате исторически сложилось так, что достаточно сложные системы программного обеспечения чаще всего создавались практически с нуля.

Другое различие между разработкой программного обеспечения и традиционным инженерным проектированием связано с наличием допусков. В традиционных областях проектирования разрабатываемые изделия обычно считаются приемлемыми, если они успешно выполняют поставленную задачу в определенных рамках. Стиральная машина, выполняющая цикл стирки, полоскания и отжима за установленное время, с учетом двухпроцентного допуска, будет совершенно приемлемой. В противоположность этому, программное обеспечение работает или правильно, или неправильно. Бухгалтерская система, точность расчетов в которой гарантируется в пределах двухпроцентного допуска, является совершенно неприемлемой.

IEEE. IEEE (Institute of Electrical and Electronics Engineers – институт инженеров по электротехнике и радиоэлектронике) был создан в 1963 г. в результате слияния американских обществ IAEE (основанного в 1884 г. 25-ю инженерами-электротехниками, среди которых был и Томас Эдисон) и IRE (основанного в 1912 г.). IEEE – всемирная организация инженеров в области электротехники, радиоэлектроники и радиоэлектронной промышленности со штаб-квартирой в Лондоне. Она направляет деятельность 36-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 (представляющее для нас наибольший интерес). 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. Web-узел IEEE размещен по адресу http://www.ieee.org; Web-узел 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.

Еще одно отличие связано с нехваткой количественных систем, называемых метриками, которые можно было бы использовать для измерения свойств программного обеспечения. Качество механического прибора часто определяется средним временем между поломками, поэтому именно это время характеризует, насколько качественно сделан прибор. В противопо-

ложность этому, программное обеспечение не изнашивается, поэтому данный метод не подходит для оценки его качества.

Невозможность количественно измерить свойства программного обеспечения является одной из основных причин того, что проектирование программного обеспечения до сих пор не нашло строгого обоснования в том смысле, как это имеет место при проектировании в области механики или электротехники. В то время как эти предметы опираются на законы физики, технология разработки программного обеспечения все еще находится на стадии поиска необходимых теоретических оснований. Фактически современное состояние технологии разработки программного обеспечения напоминает нам состояние физики в начале семнадцатого столетия, до того как Исаак Ньютон и другие ученые открыли, что такие фундаментальные свойства, как масса, ускорение и сила, могут быть измерены и что между ними существует строгая математическая зависимость.

Поэтому в настоящее время исследования в области технологии разработки программного обеспечения разворачиваются на двух уровнях. Исследователи-практики работают над развитием методов разработки, предназначенных для непосредственного применения, тогда как исследователи-теоретики ведут поиск таких принципов и теорий, которые могут быть положены в основу будущей разработки более надежных технологических методов. 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 огромное 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 оставляет желать лучшего. Ошибки в программном обеспечении приводили к таким катастрофическим (или почти катастрофическим) последствиям: принятие растущей луны за ядерную атаку, потеря 5 миллионов долларов банком Нью-Йорка только за один день, потеря космическим кораблем "Маринер-18" сведений о пробах космического пространства, радиационное облучение людей, вызвавшее их смерть или паралич, или же нарушение работы телефонных линий в обширном географическом регионе.

В то время как наука продолжает поиск методов разработки качественного программного обеспечения, профессиональные организации пропагандируют высокие стандарты этики и профессионального поведения среди своих членов. Например, Ассоциация по вычислительной технике (ACM) и IEEE приняли кодекс профессионального поведения и этики, который направлен на повышение профессионализма разработчиков программного обеспечения и предупреждение случаев их беспечного отношения к собственным разработкам.

В этой главе мы приводим некоторые результаты исследований в области технологии разработки программного обеспечения, включающие как основные принципы (жизненный цикл программного обеспечения, модульность, шаблоны проектирования), так и наиболее эффективные средства и методы разработки, используемые сегодня.

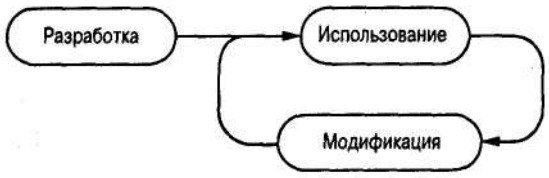
Вопросы для самопроверки

1. Почему количество строк в программе не определяет ее сложность?
2. С помощью какого метода можно определить, сколько ошибок содержится в определенной части программного обеспечения?
3. Предложите метрику для оценки качества программного обеспечения. Какие слабые места она имеет?

6.2. 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. 6.1. Здесь отражен тот факт, что, будучи однажды



6.1. Жизненный цикл программы

созданной, программа входит в цикл, включающий ее использование и модификацию; 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, поскольку они модифицируются по мере износа их частей.

Однако программы не подвержены износу. Поэтому они проходят фазу модификации из-за обнаружения ошибок, возникновения изменений в области их применения, требующих внесения соответствующих корректировок в программы, или из-за того, что прежняя модификация вызвала появление проблем в других частях программы. Например, изменения в налоговом законодательстве могут потребовать внесения корректировок в программы подготовки платежных ведомостей, связанные с расчетом удерживаемого налога. Однако внесенные изменения, в свою очередь, могут неблагоприятно повлиять на другие части программы, причем это может быть замечено не сразу.

Независимо от того, почему программа модифицируется, необходимо, чтобы некто (как правило, не автор ее исходной версии) изучил и понял программу и ее документацию; если не всю программу, то хотя бы ту часть, которая относится к делу. Любая модификация алгоритма программы может вызвать намного больше проблем, чем она предназначена решить. 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 5 было показано, как вместо безличного числового значения 645 в программе может использоваться символическое имя AirportAlt. В дальнейшем обсуждении разъяснялось, что, если возникнет необходимость в изменениях, гораздо легче изменить значение, связанное с некоторым символическим именем, чем осуществить поиск и изменение множества вхождений безличного арифметического значения 645. Как следствие, большая часть исследований в области технологии разработки программного обеспечения концентрирует свое внимание именно на фазе разработки в жизненном цикле программы. Их задача – найти способы эффективного использования тех преимуществ, которые достигаются за счет дополнительных усилий на данной стадии, понимаемых как выгодное средство достижения цели.

Традиционные этапы разработки программ. Фаза разработки в жизненном цикле программы традиционно включает следующие этапы: анализ, проектирование, реализацию и тестирование (рис. 6.2).

Фаза разработки в жизненном цикле программы начинается с анализа, основная задача которого состоит в определении нужд пользователей предлагаемой системы. Если система должна быть продуктом общего назначения, продаваемым на рынке в условиях свободной конкуренции, анализ, вероятно, будет включать широкомасштабные исследования в целях определения потребностей потенциальных пользователей. Однако если система разрабатывается для нужд конкретного пользователя, исследования будут носить более целенаправленный характер.

По мере того как выясняются потребности потенциального пользователя, они преобразуются в ряд требований, которым должна удовлетворять новая система. Эти требования формулируются в терминах, принятых в сфере приложения, без использования специальной терминологии, принятой среди специалистов по обработке данных. Одно из требований может состоять в том, что доступ к данным должен предоставляться только зарегистрированным пользователям. Другое требование может заключаться в том, чтобы данные отражали текущее состояние товарных запасов на конец последнего рабочего дня или чтобы организация данных на экране компьютера соответствовала формату используемых в настоящее время бумажных документов.

6.2. Фаза разработки в жизненном цикле программного обеспечения

После того как требования к создаваемой системе будут определены, их преобразуют в систему спецификаций, имеющих более технически организованный вид. Например, разрешение доступа к данным, только зарегистрированным в системе пользователям, должно быть представлено отдельным пунктом спецификаций; в нем указывается, что система не должна отвечать на какие-либо запросы, пока с клавиатуры не будет введен соответствующий пароль из восьми цифр, или же данные будут отображаться на экране в закодированной форме и примут осмысленный вид только после обработки некоторой стандартной программой, имя которой должно быть известно лишь зарегистрированным пользователям.

В то время как анализ определяет, что должна делать предлагаемая система, проектирование определяет, как она будет выполнять эти задачи. Именно на этом этапе определяется структура системы программного обеспечения.

Не вызывает сомнения то, что наилучшей структурой для крупной системы программного обеспечения является модульная. Именно благодаря разбиению больших систем на модули становится возможной их практическая реализация. Без этого разбиения технические детали, необходимые при реализации большой системы, превысили бы возможности человеческого восприятия. При использовании модульной конструкции разработчик может ограничиться изучением только тех деталей, которые относятся к рассматриваемому модулю. Модульная конструкция также удобна при дальнейшем сопровождении программы, поскольку позволяет вносить изменения на модульной основе. (Если изменение нужно внести в способ выполнения расчетов по больничным листам работников, то потребуется рассматривать только те модули, которые непосредственно связаны с оплатой больничных листов.)

Однако понятие модуля может трактоваться по-разному. Если подходить к задаче конструирования, используя парадигму традиционного императивного программирования, то модули будут состоять из отдельных процедур и разработка модульной структуры примет форму выявления различных заданий, которые создаваемая система должна будет выполнять. В противоположность этому, если предполагается использовать объектно-ориентированный подход, то в качестве модулей будут выступать отдельные объекты, а процесс конструирования модульной структуры превратится в процесс выявления имеющихся в предметной области сущностей (объектов) и определения их поведения в предлагаемой системе.

Следующим этапом является реализация. Он включает собственно написание программ, создание файлов данных и разработку баз данных.

Тестирование тесно связано с реализацией, так как каждый модуль системы, как правило, должен подвергаться тестированию, как только он будет написан. Действительно, в правильно сконструированной системе каждый модуль может и должен быть проверен либо независимо от других модулей, либо с использованием упрощенных версий других модулей, называемых *заглушками* (stub). Назначение заглушек состоит в том, чтобы имитировать взаимодействие тестируемых модулей и остальной части системы. Аналогичным образом можно тестировать и систему в целом, по мере завершения создания отдельных модулей и их объединения в единое целое.

К сожалению, этап тестирования и устранения обнаруженных в системе неполадок чрезвычайно сложно довести до успешного завершения. Опыт показывает, что большие системы программного обеспечения могут содержать множество оши-

бок 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 ошибок. 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.

Современные тенденции. Ранние подходы к проектированию программного обеспечения предполагали строго последовательное выполнение 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 общего анализа системы до начала ее проектирования. Точно так же необходимо полностью завершить этап проектирования системы до начала ее реализации. Подобная схема процесса разработки получила название модели водопада (по аналогии с движением потока падающей воды, поскольку процесс разработки должен был двигаться только в одном направлении).

Можно отметить сходство между четырьмя этапами решения задач, определенными математиком Полиа (см. раздел 4.3), и этапами анализа, проектирования, реализации и тестирования, входящими в фазу разработки программного обеспечения. В конце концов, разработать большую систему программного обеспечения действительно означает решить сложную задачу. Однако традиционный подход к разработке программного обеспечения, соответствующий модели водопада, полностью противоположен свободно развивающемуся методу проб и ошибок, присущему творческому решению задач. В то время как подход, соответствующий модели водопада, стремится к организации четко структурированной среды, в которой разработка программного обеспечения будет происходить в строгой последовательности, творческий подход к решению задач предполагает неструктурированную среду, в которой можно отказаться от работы по предварительно намеченному плану, следуя интуиции, не требующей пояснения, почему это происходит.

Разработка программных систем в реальном мире. Следующий сценарий типичен в отношении тех задач, с которыми сталкиваются разработчики программного обеспечения в реальности. Допустим, компания XYZ заказывает фирме, занимающейся созданием программного обеспечения, разработать и установить интегрированную систему программного обеспечения для обработки данных в масштабах этой компании. При развертывании этой системы каждому работнику выделяется персональный компьютер, подключенный к сети, общей для всей компании. Эти персональные компьютеры не только обеспечивают доступ к установленной в компании системе обработки данных, но и могут использоваться в качестве некоторого настраиваемого инструмента, с помощью которого каждый работник компании стремится повысить производительность своего труда. Например, какой-либо работник разрабатывает электронную таблицу, упрощающую или ускоряющую выполнение стоящих перед ним задач. К сожалению, подобные разработанные пользователями приложения очень часто оказываются неправильно спроектированными и недостаточно тщательно протестированными, а также включают функции, работа которых была понята работником неправильно или же лишь частично. Со временем эти самодельные приложения включаются в процедуры производственной деятельности компании. В результате то, что исходно представляло собой правильно спроектированную систему, оказывается погребенным под беспорядочным нагромождением из плохо спроектированных, недокументированных, содержащих множество ошибок приложений.

С недавних пор в методах проектирования программного обеспечения стало учитываться это основополагающее противоречие, что выразилось 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 студентов, предназначенным для секретаря университета, то ее первое приближение может включать только функцию просмотра личных дел студентов. После того как эта версия окажется работоспособной, в ней поэтапно будут реализованы дополнительные функции, такие, как добавление новых записей и обновление уже существующих.

Пошаговая модель является примером использования в технологии разработки программного обеспечения метода *создания прототипов* (prototyping) или *макетирования*, который заключается в создании и последующей оценке неполных версий разрабатываемой системы, называемых *прототипами* (prototypes). В пошаговой модели прототип развивается в конечную версию системы, поэтому данный вариант метода создания прототипов именуется *эволюционным макетированием* (evolution prototyping). В других случаях прототипы могут отбрасываться, уступая место новым реализациям конечного проекта. Такой вариант метода создания прототипов называется методом с *временным макетированием* (throwaway prototyping). 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.

Другое усовершенствование методов проектирования программных систем предусматривает применение компьютерных технологий к самому процессу разработки программного обеспечения, что привело к появлению *CASE-технологий* (CASE – Computed-Aided Software Engineering). Эти компьютеризованные системы, известные как CASE-инструменты, включают средства планирования проекта (предназначенные для оценки стоимости календарного планирования и распределения исполнителей), средства управления проектом (для контроля за процессом разработки проекта), средства документирования (для написания и систематизации проектной документации), средства создания прототипов и имитации (для создания прототипов), средства разработки интерфейсов (для разработки графических интерфейсов пользователя), а также средства программирования (для написания и отладки программ). Некоторые из этих инструментов немногим отличаются от

обычных текстовых редакторов, приложений электронных таблиц и систем электронной почты, используемых в других приложениях. Однако другие представляют собой достаточно сложные пакеты, предназначенные преимущественно для использования при разработке систем программного обеспечения. Например, некоторые CASE-инструменты включают генераторы кода, которые по заданной спецификации на некоторую часть системы генерируют программу на языке высокого уровня, представляющую собой реализацию этой части системы.

Вопросы для самопроверки

1. В чем состоит отличие между требованиями к системе и спецификацией на систему?
2. Кратко охарактеризуйте каждый из четырех этапов (анализ, проектирование, реализация и тестирование) фазы разработки в жизненном цикле программного обеспечения.
3. Охарактеризуйте различия между традиционной моделью разработки программного обеспечения (модель водопада) и новым подходом, предусматривающим создание прототипов.

6.3. МОДУЛЬНОСТЬ

Одним из ключевых положений в разделе 6.2 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, каждый из которых сконструирован так, чтобы выполнять только определенную часть общей задачи.

Реализация модулей. Модульности можно достичь многими способами. В главах 4 и 5 было показано, как можно реализовать модули в виде отдельных процедур, которые впоследствии используются в качестве конструкционных элементов для построения систем большего размера. В главе 5 модульность также обсуждалась в контексте объектно-ориентированного подхода. В этом случае модули принимали форму объектов, каждый из которых имел собственную внутреннюю организацию, не зависящую от содержимого других объектов. Фактически рост популярности объектно-ориентированного подхода к разработке программного обеспечения в значительной степени можно объяснить именно модульной структурой, изначально присущей этому методу.

Структурная схема – это традиционное средство представления модульной структуры, достигаемой за счет создания процедур. На этой схеме каждый модуль (процедура) представляется прямоугольником, а связи между ними – стрелками, соединяющими эти прямоугольники. Структурная схема, представленная на рис. 6.3, отображает модульную структуру простой ролевой игры в жанре фантази, в которой игрок должен передвигаться по комнатам средневекового замка, решая в каждой некоторую задачу, прежде чем ему будет позволено перейти в следующее помещение. На схеме показано, что модуль с именем CoordinateGame (Координировать игру) использует модули InitializeGame (Инициализировать игру), SimulateGreatHall (Имитация большого зала), SimulateDungeon (Имитация темницы) и SimulateTurret (Имитация

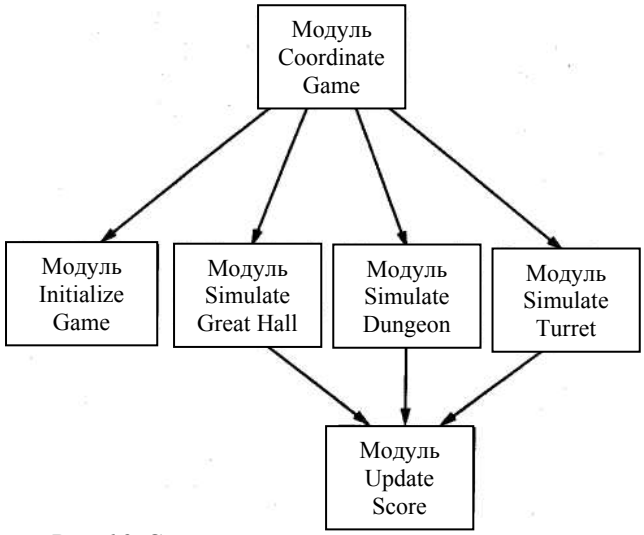


Рис. 6.3. Структурная схема простой ролевой игры

башни) в качестве абстрактных средств решения задачи. Точнее говоря, процедура CoordinateGame вызывает процедуру InitializeGame для организации начала игры (узнать имя игрока, установить уровень сложности игры – начинающий, средний или сложный – и т.д.). После каждого перехода игрока на новый уровень (т.е. перехода в следующую комнату) процедура CoordinateGame вызывает другую процедуру, выполняющую имитацию требуемой комнаты, чтобы организовать выполнение происходящих в этой комнате событий. Кроме того, на схеме показано, что каждый из модулей имитации комнаты обращается к процедуре UpdateScore (Обновление счета), чтобы записать результаты, достигнутые игроком в данной комнате.

В то время как структурные схемы используются для представления внутренней структуры программного обеспечения с процедурной организацией, *диаграммы классов* применяются для представления структуры систем с объектно-ориентированной организацией. Диаграмма классов определяет набор существующих в системе классов объектов и описывает взаимоотношения этих классов. На рис. 6.4 представлена простая диаграмма классов для упомянутой выше ролевой игры. На этой диаграмме показано, что система состоит из объектов двух типов, PlayRecord (Запись игрока) и Room (Комната), связанных отношением IsCurrentlyln (В данный момент внутри). Здесь объект типа PlayRecord содержит данные и процедуры, относящиеся к конкретному игроку (имя, уровень, результат), а объект типа Room – данные и процедуры, относящиеся к определенной комнате (образ комнаты, возникающий на экране, и связанные с этой комнатой задачи).

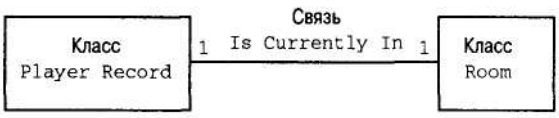


Рис. 6.4. Диаграмма классов для простой ролевой игры

Отношение IsCurrentlyln представляет связи, существующие между комнатами и игроками, т.е. запись игрока связывается с комнатой, в которой он в данный момент находится. Числа, указанные на концах линий, представляющих данное отношение, показывают, что один игрок в конкретный момент времени может быть связан только с одной комнатой. Такая связь типа "один к одному" отличается от связи "один ко многим", примером которой может служить связь "обучать", когда один преподаватель ведет на протяжении семестра сразу несколько курсов.

В последние годы был достигнут значительный прогресс в разработке стандартной системы обозначений для представления элементов в процессе объектно-ориентированного проектирования. Наиболее ярким примером является язык *UML* (Unified Modeling Language – унифицированный язык моделирования), представляющий собой унифицированную систему для представления множества объектно-ориентированных понятий. Обозначения, используемые на рис. 6.4, основаны на стандартах языка UML.

Связанность модулей. Выше в этой главе модульность была предложена как способ 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. 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 задача состоит в обеспечении максимальной независимости отдельных модулей. Однако 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.

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 между отдельными процедурами в программе. Другая форма – это *связанность по данным*, т.е. совместное использование одних и тех же данных несколькими модулями.

Структурная схема, представленная на рис. 6.3, отражает связанность модулей по управлению при процедурном подходе к построению упоминаемой выше ролевой игры. На структурной схеме связанность по данным обычно представляется с помощью дополнительных стрелок (рис. 6.5). На этой схеме изображено, какие элементы данных передаются модулю при обращении к предоставляемым им сервисам и какие возвращаются вызывающему модулю при завершении выполнения запрошенной функции. В частности, на данной схеме показано, что модуль CoordinateGame при обращении к процедуре SimulateGreatHall передает ей значение уровня игрока. Обратите внимание, что модуль CoordinateGame, в свою очередь, получает значение уровня игрока от процедуры InitializeGame. Кроме того, на схеме показано, что когда любой из модулей имитации помещения обращается с запросом к процедуре UpdateScore, он передает ей данные о количестве очков, уже набранных игроком.

Минимизация связанности по данным является одним из главных преимуществ объектно-ориентированного подхода. Побудительным мотивом при создании концепции объекта было желание собрать в единый модуль программы, работающие с определенным элементом данных.

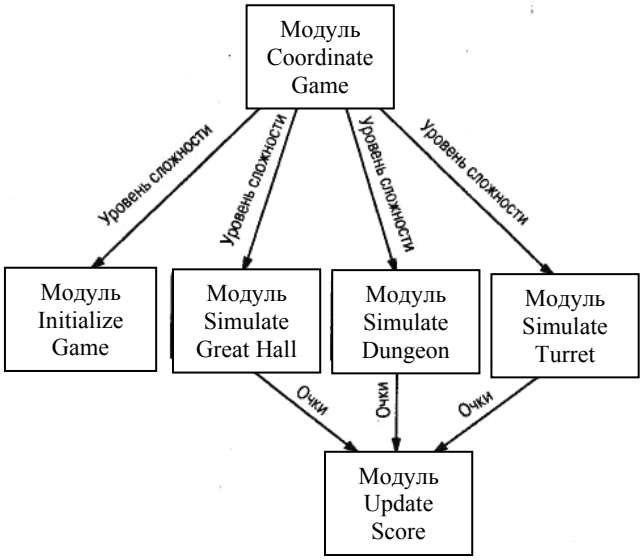


Рис. 6.5. Структурная схема, содержащая сведения о связанности по данным

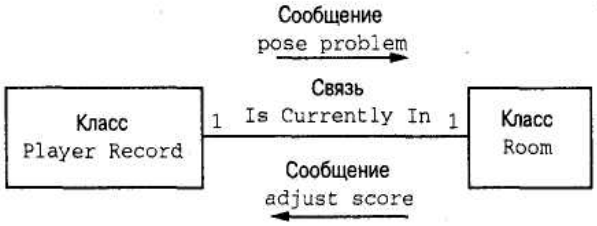


Рис. 6.6. Диаграмма взаимодействий для простой ролевой игры

Поэтому в объектно-ориентированной системе большинство межмодульных связей имеет форму связей, устанавливаемых между объектами, которые, как правило, реализуются как связи по управлению. Таким образом, обращение к объекту с запросом о выполнении некоторого задания является, по существу, запросом о выполнении некоторого метода (процедуры) внутри объекта. Поэтому такой запрос вызывает передачу управления объекту аналогично тому, как управление передается вызываемой процедуре при императивном подходе.

Один из способов представления связей между объектами в объектно-ориентированном проектировании состоит во внесении информации в диаграмму классов в целях построения *диаграммы взаимодействий*, которая, по существу, является диаграммой классов, показывающей, как различные объекты системы взаимодействуют друг с другом. Простая диаграмма взаимодействий для нашей ролевой игры (с использованием обозначений языка UML) представлена на рис. 6.6. На ней показано, что объект класса Room может послать сообщение объекту класса PlayerRecord, предлагая ему обновить значение счета, а объект класса PlayerRecord может послать сообщение объекту класса Room, запрашивая у него сведения об очередном помещении.

Независимо от типа связей, программисту следует стремиться к тому, чтобы они были явно видны в конечной версии программы. Скрытая связанность называется *неявной* и приводит к множеству ошибок в программном обеспечении. Обычно образование неявных связей происходит при использовании *глобальных данных* – элементов данных, которые автоматически доступны всем модулям системы. В отличие от них, *локальные данные* доступны только внутри определенного модуля, если они не пересылались другому в явной форме. В большинстве языков высокого уровня существуют методы реализации как глобальных, так и локальных данных.

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, 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 отличаться. Например, одни действия могут быть связаны с получением данных, а другие – с выдачей сообщений об ошибках.

Более сильная форма связности – *функциональная связность*; она означает, что все части модуля собраны вместе для выполнения одних и тех же действий. Модуль SimulateGreatHall, показанный на рис. 6.3, нельзя считать функционально связным, если он содержит сведения, необходимые для создания изображения на экране большого зала замка, представления всех связанных с этим залом задач и обработки ответных действий игрока. Однако если эти детали локализованы в других модулях и используются модулем SimulateGreatHall в качестве абстрактных инструментов, то каждый этап работы этого модуля можно рассматривать в общем контексте наблюдения за действиями игрока, связанными с большим залом замка. В этом случае модуль SimulateGreatHall выглядит более функционально связным.

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 ролевой игре каждый объект помещения будет, вероятно, содержать метод для создания изображения этого помещения наряду с методами, необходимыми для представления связанных с ним задач и получения ответа игрока. 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 (показано на рис. 6.7).

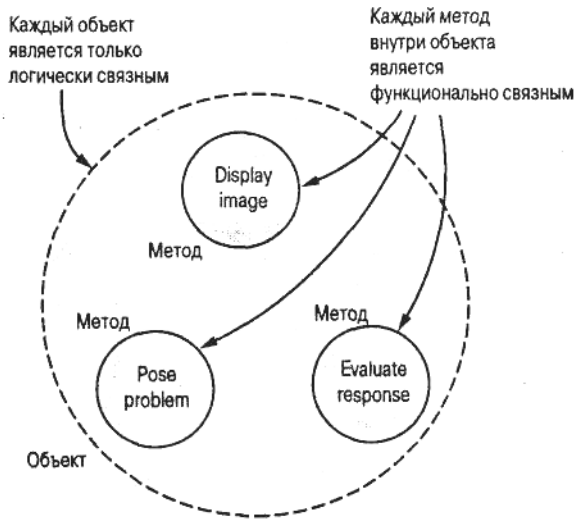


Рис. 6.7. Логическая и функциональная связность внутри объекта, представляющего отдельную комнату в простой ролевой игре

Вопросы для самопроверки

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. Игра в бридж состоит из двух этапов: торг и розыгрыш. Проанализируйте связанность этих фаз, предварительно определив, какая информация передается из первой фазы во вторую в явной форме. Какая информация передается в этом случае неявно?

3. Совместимы ли задачи максимизации связности и минимизации связанности? Другими словами, будет ли естественным образом уменьшаться связь модулей системы при возрастании связности элементов модулей?

4. Расширьте диаграмму взаимодействий, представленную на рис. 6.6, включив в нее другие сообщения, которые должны передаваться между объектами классов Room и PlayerRecord.

6.4. МЕТОДЫ ПРОЕКТИРОВАНИЯ

Разработка методов проектирования систем программного обеспечения является основным предметом поиска в области технологии разработки программного обеспечения. В этом разделе мы рассмотрим несколько разработанных ранее методов и познакомимся с направлениями текущих исследований.

Нисходящие и восходящие методы разработки. Возможно, наиболее известной стратегией проектирования программных систем является нисходящая методология. Суть ее состоит в том, что не следует пытаться решить сложную задачу за один этап. Приступая к решению поставленной задачи, необходимо сначала разбить ее на меньшие, более простые подзадачи, которые, в свою очередь, следует разбить на еще меньшие. В результате сложная задача будет сведена к набору более простых задач, решение которых приведет к выполнению и самой исходной задачи.

При использовании нисходящей технологии проектирования обычно образуется иерархическая система последовательных уточнений, которая, как правило, может быть прямо отображена в модульную структуру, совместимую с парадигмой императивного программирования. Решение задач самого нижнего уровня этой иерархии обеспечивается процедурными модулями, выполняющими элементарные задания. В системе эти модули используются в качестве абстрактных инструментов модулями более высоких уровней, предназначенными для решения сложных задач.

В противоположность этому методу проектирования, при восходящем методе проектирование системы начинается с определения отдельных задач внутри системы. Затем изучается, как решение этих задач может использоваться в качестве абстрактных инструментов для решения более общих задач. Многие годы этот подход считался хуже нисходящего метода проектирования. Однако в настоящее время методология восходящего проектирования получила мощную поддержку. Одна из причин этого заключается в том, что нисходящая методология ищет решение, в котором основной модуль использует подмодули, каждый из которых основывается на подмодулях, и т.д. Однако для многих систем наилучший вариант решения лишен подобной иерархической природы. Действительно, при построении приложения по принципам архитектуры "клиент/сервер" или использовании методов параллельной обработки проектное решение, в котором два или несколько модулей взаимодействуют как равноправные партнеры, может оказаться более удачным, чем проект, состоящий из управляющего модуля, обращающегося к модулям подпрограмм для решения стоящих перед ним задач.

Другой причиной такого интереса к восходящему проектированию является то, что оно лучше соответствует задачам построения сложных систем программного обеспечения из предварительно созданных, стандартных, свободно продаваемых компонентов. Однако именно этот подход является наиболее современной тенденцией в технологии разработки программного обеспечения. Подробно мы рассмотрим эту стратегию ниже.

Общие инструменты разработки. В технологии разработки программного обеспечения создано множество систем обозначения, предназначенных для анализа и проектирования систем. Мы уже встречались с некоторыми из них, например структурными схемами, диаграммами классов и диаграммами взаимодействия. Еще одна разновидность подобных инструментов – *диаграмма потоков данных*, содержащая графическое представление путей перемещения данных в системе. На диаграмме потоков данных указываются места происхождения, конечного назначения и промежуточной обработки данных в системе. Каждый из символов, который указывается на такой диаграмме, имеет специальное значение: стрелки представля-

ют потоки данных; сплошные линии – источники и приемники данных; кружки указывают, где с данными производятся некоторые действия; а жирные прямые линии отмечают места сохранения данных. В каждом случае символ помечается именем представляемого объекта. Схема потоков данных для нашей простой ролевой игры представлена на рис. 6.8.

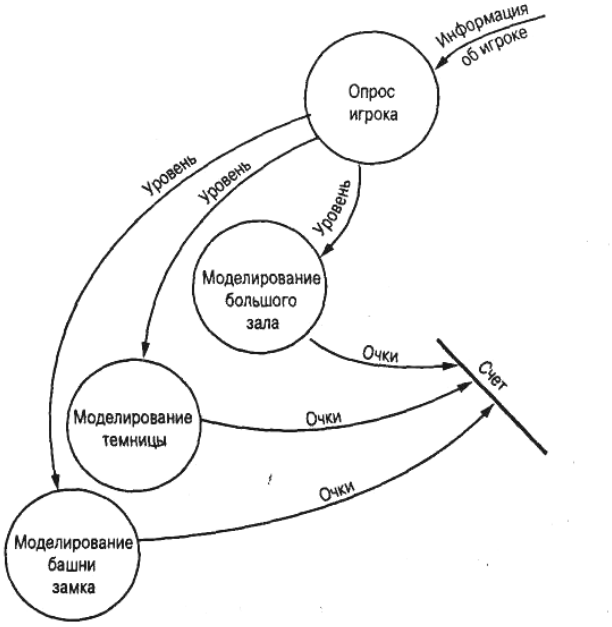


Рис. 6.8. Диаграмма потоков данных в простой ролевой игре

Метод разработки систем программного обеспечения, основанный на анализе потоков данных, возник в среде императивного программирования. Изначально задача состояла в том, что, отслеживая пути данных в предлагаемой системе, можно обнаружить, где элементы данных сливаются, разделяются или изменяются каким-либо другим образом. В таких местах системы обязательно потребуются те или иные вычислительные действия, которые (взятые по отдельности или сгруппированные) позволят образовать процедурные модули системы. Следовательно, изучение потоков данных поможет определить модульную структуру системы.

Хотя разработка метода анализа потоков данных происходила в рамках императивной парадигмы программирования, он нашел широкое применение и в объектно-ориентированной среде. В частности, определение элементов данных в системе помогает выявить необходимые объекты, а определение происходящих с данными изменений – уточнить действия, которые эти объекты должны выполнять.

Еще одним инструментом анализа и проектирования систем программного обеспечения является *диаграмма "сущность – связь"*, на которой наглядно представляются присутствующие в системе элементы информации (сущности), а также существующие между этими элементами отношения. В качестве примера рассмотрим часть диаграммы "сущность – связь" для системы программного обеспечения, предназначенной для сбора информации о преподавателях, студентах и занятиях.

Прежде всего определим те сущности, которые должны содержаться в системе. Сущность Professor (Профессор) представляет отдельного преподавателя университета; сущность Student (Студент) – отдельного студента и сущность Class (Занятие) – раздел определенного курса. С каждым экземпляром сущности Professor связывается фамилия, адрес, идентификационный номер, оклад и т.д., с каждым экземпляром сущности Student – фамилия, адрес, идентификационный номер, средний балл и т.д., с каждым экземпляром сущности Class – название курса (История 101), семестр и год, аудитория, время проведения и т.д.

Определив присутствующие в создаваемой системе сущности, следует рассмотреть связи, существующие между ними. Прежде всего, заметим, что каждый преподаватель ведет занятия, а каждый студент посещает их. Следовательно, связь между сущностями Professor и Class можно определить как Teaches (Проводит), а между сущностями Student и Class – как Attends (Посещает). (Обратите внимание, что сущности обозначаются существительными, а связи между ними – глаголами.)

Для отображения этих сущностей и связей составляют диаграмму "сущность – связь", показанную на рис. 6.9, где каждая сущность представлена прямоугольником, а каждая связь – ромбом. Из этой диаграммы следует, что преподаватели связаны с занятиями посредством связи Teaches, а студенты – посредством связи Attends.

Рис. 6.9. Пример диаграммы "сущность – связь"

Однако для двух существующих в данном примере связей характерна различная структура. Связь между сущностями Professor и Class имеет тип "один ко многим", поскольку каждый преподаватель проводит несколько различных занятий, однако каждое занятие проводится только одним преподавателем. В противоположность этому, связь между сущностями Student и Class имеет тип "многие ко многим", так как каждый студент посещает несколько занятий и каждое занятие посещают несколько студентов. Эта дополнительная информация представлена на рис. 6.9 с помощью стрелок, соединяющих сущности и связи. В частности, одинарная стрелка, направленная в сторону сущности, указывает, что только одна сущность данного вида входит в каждую связь этого типа, тогда как двойная стрелка означает, что в этой связи могут участвовать несколько экземпляров данной сущности. Например, одинарная стрелка, направленная на рис. 6.9 в сторону сущности Professor, означает, что каждое занятие проводит только один преподаватель, в то время как двойная, направленная в сторону

сущности Class от связи Teaches, показывает, что каждый преподаватель может проводить более чем одно занятие.

Можно предположить, что среди различных средств, используемых разработчиками программного обеспечения в прошлом, диаграммы "сущность – связь" имели больше шансов выжить при переходе к объектно-ориентированным методам. Действительно, определение сущностей является, по сути, определением объектов системы, а классификация связей между ними – первый шаг на пути к определению необходимых связей и взаимодействий этих объектов. Поэтому в традиционной диаграмме "сущность – связь" легко распознать предшественницу диаграммы классов языка UML (рис. 6.4), которая используется в объектно-ориентированной среде проектирования.

Существует также такой полезный инструмент разработки систем программного обеспечения, как *словарь данных*. Он представляет собой центральное хранилище информации обо всех элементах данных из всех частей системы. Сюда входят идентификатор, используемый для обращения к каждому элементу; сведения о том, из каких символов может состоять каждый элемент (Будет ли элемент состоять только из цифр или, возможно, только из букв? Каким будет диапазон значений, которые могут присваиваться данному элементу?); данные о том, где хранится этот элемент данных (Будет элемент записан в файл или базу данных; если да, то в какую именно?); а также указания, где именно в программе осуществляются обращения к элементу (Каким модулям требуется информация из данного элемента данных?).

Разработка словаря данных преследует сразу несколько целей. Одна из них состоит в расширении взаимодействия потенциальных пользователей системы и аналитика, перед которым поставлена задача превращения пожеланий пользователя в требования и спецификации. Было бы весьма досадно обнаружить уже после завершения реализации системы, что номера деталей в действительности не являются числовыми или размер инвентаризационной ведомости превышает допустимый в системе. Процесс создания словаря данных помогает избежать подобных недоразумений.

Другая цель создания словаря данных – установить в системе необходимое единообразие. Обычно именно за счет создания словаря можно избежать избыточности и противоречивости данных. Например, элемент данных, который в записи складского учета был назван PartNumber, в записях о продажах может получить имя PartId. Кроме того, отдел кадров может использовать элемент Name (Имя) по отношению к сотруднику, в то время как в записи складского учета элемент Name (Название) может входить в описание детали.

Наконец, следует упомянуть и *CRC* (Class-Responsibility-Collaboration – карты взаимодействия классов), которые могут оказаться полезными при проектировании объектно-ориентированных систем. Карта CRC является, в сущности, традиционной индексной картой, содержащей описание объекта. Метод использования карт CRC состоит в том, что проектировщик создает карту CRC для каждого объекта предлагаемой системы, а затем использует эти карты для моделирования действий в системе – например, на поверхности рабочего стола или с помощью "театрализованного" эксперимента, в котором каждый член команды проектировщиков играет роль объекта, выполняя все действия именно так, как это предписывается соответствующей картой. Такое моделирование используется для обнаружения недостатков в проекте.

Шаблоны проектирования. В своих поисках, предпринимаемых с целью найти способы создания программного обеспечения из готовых компонентов, разработчики программного обеспечения обратились и к области архитектуры. Особый интерес здесь представляет книга Кристофера Александера (Christopher Alexander) и др. "A Pattern Language", в которой описывается набор шаблонов для сообщества архитекторов-проектировщиков. Каждый шаблон состоит из постановки задачи, за которой следует предлагаемое решение. Приведенные задачи в основном универсальны, а предлагаемое решение является обобщенным в том смысле, что оно относится к общей природе задачи, а не к конкретному случаю.

Например, один из шаблонов, названный "Quiet Backs" ("тихие уголки"), предназначен для создания в деловом центре тихих местечек для отдыха. Предлагаемое решение заключается в том, чтобы спроектировать в деловых районах "тихие уголки". В некоторых случаях следует проектировать район вокруг главной улицы, к которой все здания повернуты фасадами, обеспечивая, таким образом, тихую сторону на улицах, проходящих позади этих зданий. В других случаях "тихие уголки" могут быть получены с помощью парков, рек или соборов.

Важным для нашего обсуждения является то, что в данной работе Александера сделана попытка определить общие задачи и предложить типовые шаблоны их решения. Сегодня многие разработчики программного обеспечения пытаются применить аналогичный подход к проектированию больших систем программного обеспечения. В частности, исследователи используют шаблоны проектирования в качестве средств создания обобщенных строительных блоков, из которых можно конструировать системы программного обеспечения.

Среда программирования языка Java. Java был бы просто еще одним объектно-ориентированным языком программирования, если бы не имел набора структур, разработанных специально для упрощения процесса программирования на этом языке. Набор этих структур получил общее название "Прикладной программный интерфейс", или API (Application Program Interface), и является частью набора инструментальных средств языка Java (JDK – Java Development Kit), разработанного компанией Sun Microsystems. Структуры API языка Java включают шаблоны для решения таких задач, как разработка графических интерфейсов пользователя (GUI), работа с аудио- и видеоданными, передача данных через Internet или разработка анимированных Web-страниц. Таким образом, среда программирования языка Java – это прогресс в направлении конструирования программного обеспечения из готовых компонентов.

Примером такого шаблона является Publisher – Subscriber (Издатель – Подписчик), состоящий из модуля (издателя), который должен посылать копии своих "публикаций" другим модулям (подписчикам), как показано на рис. 6.10. Как частный случай рассмотрим группу данных, которые изображаются на экране компьютера одновременно в нескольких форматах, например в виде круговой диаграммы и гистограммы. В этом случае любое изменение данных должно быть отражено сразу в обоих форматах. Следовательно, модули программного обеспечения, отвечающие за построение диаграмм, должны оповещаться об изменении данных. В данном случае модуль программного обеспечения, управляющий данными, выполняет роль издателя, посылающего сообщения о произошедших изменениях сразу всем его подписчикам, т.е. модулям, отвечающим за построение диаграмм.

Рис. 6.10. Шаблон Publisher – Subscriber

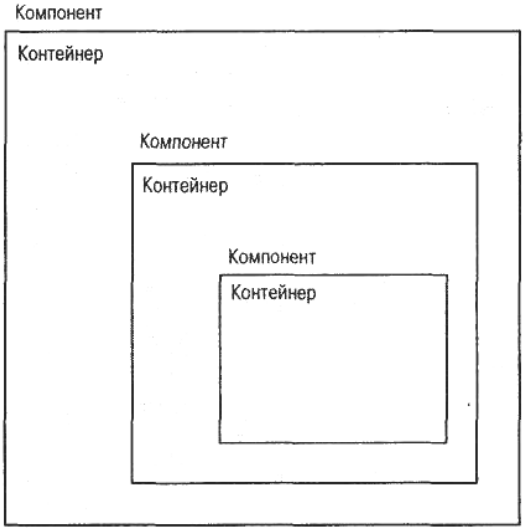


Рис. 6.11. Шаблон Container – Component

Еще одним примером шаблона проектирования программного обеспечения является Container – Component (Контейнер – Компонент). Он воплощает обобщенную концепцию контейнера, содержащего компоненты, которые, в свою очередь, также могут быть контейнерами (рис. 6.11). Примером использования этого шаблона может служить иерархия каталогов или папок, создаваемая программой управления файлами операционной системы. Каждый из этих каталогов обычно содержит другие каталоги, которые также могут содержать каталоги, и т.д. Короче говоря, шаблон Container – Component служит для описания рекурсивной концепции контейнеров, содержащих такие же контейнеры.

После того как структура шаблонов, подобных Publisher – Subscriber или Container – Component, будет определена, разработчики программного обеспечения выполняют разработку каркасных программных элементов, называемых *структурами*, в которых реализуются основные особенности решений, предлагаемых шаблонами. Тогда как реализация специфических функций, присущих конкретным приложениям, откладывается на более поздний срок за счет организации соответствующих слотов, подлежащих заполнению конкретными значениями. Как дополнение к структурам, разработчики программного обеспечения обычно предлагают документацию, описывающую, как следует заполнять эти структуры в целях получения завершенной реализации положенного в ее основу шаблона в рамках конкретной разработки. Такую документацию называют рецептом, поэтому подборки структур вместе с их рецептами шутливо именуют кулинарными книгами.

Исследователи надеются, что с помощью кулинарных книг разработчики программного обеспечения получат, наконец, возможность конструировать большие сложные системы программного обеспечения из готовых компонентов, таких как структуры. Первые результаты показали, что при разработке новой системы такой подход позволяет значительно уменьшить необходимый объем программирования.

Несмотря на ажиотаж, вызванный в сообществе разработчиков программного обеспечения вокруг шаблонов проектирования, интересно отметить, что сам Александер не был удовлетворен результатами использования его шаблонов в архитектуре. Он обнаружил, что системам, разработанным на основе его шаблонов, недостает индивидуальности, и поэтому с начала 80-х направил все свои усилия на поиск путей включения в новые разработки этого ускользающего качества. Однако разработчики программного обеспечения сходятся на том, что при разработке программного обеспечения важна не красота и индивидуальность, а прежде всего, точность и эффективность. Поэтому, считают они, применение шаблонов проектирования в области создания программного обеспечения окажется более успешным, чем в архитектуре.

Вопросы для самопроверки

1. Предположим, что секретарь получает просьбы о предоставлении встреч с его боссом. Действия секретаря заключаются или в назначении встречи на ближайшие дни, или в немедленном предоставлении желаемой встречи. Нарисуйте схему потоков данных, представляющую эту часть работы секретаря.
2. Нарисуйте диаграмму "сущность – связь", представляющую авиационные компании; полеты, выполняемые каждой компанией; и пассажиров различных рейсов.

3. Приведите примеры некоторых программных структур, обсу-ждаемых в предыдущих главах, которые можно рассматривать в качестве шаблонов проектирования.

4. Какую роль, как надеются исследователи, будут играть структуры в процессе создания программного обеспечения в будущем?

6.5. ТЕСТИРОВАНИЕ

В разделе 4.6 рассматривались методы верификации алгоритмов в строгом математическом смысле. Однако было сделано заключение, что большая часть создаваемого сегодня программного обеспечения "верифицируется" посредством тестирования. К сожалению, тестирование в лучшем случае можно расценивать как неточную науку. Выполнив тестирование, мы не можем утверждать, что некоторая часть программы правильна, если не были выполнены все проверки, исчерпывающие возможные сценарии. Но даже для простой структуры цикла, содержащей единственную инструкцию if-then-else, существует более 1000 различных пересекающихся путей выполнения, если цикл повторяется всего лишь десять раз. Следовательно, проверить все возможные пути выполнения в сложной программе – просто невыполнимая задача.

С другой стороны, создатели программного обеспечения разработали методы тестирования программ, повышающие вероятность обнаружения существующих в них ошибок. Один из таких методов основан на наблюдении, что ошибки в программном обеспечении имеют тенденцию к группированию. (Это часто называют принципом Парето, в честь Вильфредо Парето (Vilfredo Pareto), который заметил, что малая часть населения Италии контролирует большую часть богатств этой страны.) Как показывает опыт, в крупной системе программного обеспечения всегда существует определенное число модулей, являющихся более проблематичными, чем остальные. Следовательно, обнаружив эти модули и проверив их более тщательно, можно выявить большую часть существующих в системе ошибок, даже если все остальные модули будут протестированы обычным, менее тщательным образом. Задача, таким образом, заключается в обнаружении таких проблематичных модулей.

Еще один метод, 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. 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, который обычно относится к концепции тестирования по принципу "черного ящика", именуется анализом граничных условий. Этот метод заключается в определении граничных условий, указанных в спецификации программного обеспечения, с последующей проверкой функционирования программы при этих условиях. Например, если предполагается, что в программе допускается введение исходных значений только из конкретно заданного диапазона, то 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.

Therac-25. Необходимость жестких требований к качеству проектных работ подтверждается проблемами, возникшими при использовании Therac-25, – системы радиационной терапии, включающей управляемый компьютером ускоритель электронов. Эта система применялась медиками в середине 80-х гг. XX в. Недостатки, присущие этому проекту, вызвали шесть случаев передозировки радиационного облучения, три из которых привели к смертельному исходу. К недостаткам системы можно отнести неудачный проект интерфейса пользователя, позволявший оператору начинать облучение до того, как машине будет задана соответствующая дозировка, а также плохую координацию взаимодействия аппаратного и программного обеспечения, вызвавшую снижение необходимого уровня безопасности. Более подробно познакомиться с данным вопросом можно в форуме Risks на Web-узле http: //catless.ncl.as.uk/, посвященном вопросам возникновения рисков.

Еще одним методом тестирования по принципу "черного ящика" является применение избыточности. В данном случае две системы для выполнения одной и той же задачи разрабатываются независимо различными группами или даже компаниями. Затем обе системы проверяют путем задания им одинаковых данных и сравнения результатов. Ошибки проявляются при расхождении в полученных результатах. Такие методы часто применяются в космических исследовательских системах.

Следующим методом тестирования по принципу "черного ящика" является метод "упрощенных версий", который все шире используется разработчиками программных средств, предназначенных для рынка персональных компьютеров. Он заключается в представлении части предполагаемой аудитории предварительной версии программного обеспечения, называемой бета-версией. Основная задача – изучить, как программное обеспечение функционирует в реальных условиях, перед тем как конечная версия продукта будет утверждена и выпущена на рынок.

Достоинства подобного тестирования опытного образца превышают рамки традиционного обнаружения ошибок. Получение отзывов потребителей (как положительных, так и отрицательных) может существенно помочь в уточнении рыночных стратегий. Более того, раннее распространение бета-версий программного обеспечения помогает другим производителям программного обеспечения в разработке совместимых продуктов. Например, распространение бета-версии новой операционной системы ускорит разработку совместимых с ней обслуживающих программ, в результате окончательная версия операционной системы окажется на полке магазина сразу в окружении сопутствующих программных продуктов. Наконец, существование бета-версий программного обеспечения помогает создать на рынке атмосферу предвкушения, которая повышает популярность продукта, а значит, и увеличивает объем его продаж.

Вопросы для самопроверки

1. Какая проверка при тестировании программного обеспечения является успешной, нашедшая или не нашедшая ошибки?

2. Какие методы можно предложить для обнаружения в системе модулей, которые необходимо тестировать более тщательно, чем все остальные?

3. Что можно считать хорошим тестом для проверки работы пакета программного обеспечения, разработанного для сортировки списка, содержащего не более 100 элементов?

6.6. ДОКУМЕНТИРОВАНИЕ

Чтобы успешно пользоваться и управлять работой системы программного обеспечения, люди должны иметь возможность изучить ее. Следовательно, важной частью пакета программного обеспечения является документация, а ее разработка – не менее важная тема в технологии разработки программного обеспечения.

При создании документации к пакетам программного обеспечения обычно преследуются две задачи. Одна из них – описание функций программного обеспечения и методов его использования. Эта часть называется *документацией пользователя*, так как она предназначена для нужд пользователей данного программного обеспечения. Следовательно, документация пользователя должна носить нетехнический характер.

Сегодня документация пользователя считается важным средством маркетинга. Хорошая документация (в сочетании с хорошо разработанным интерфейсом пользователя) делает программный пакет более доступным и 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 котором представлен вводный материал о наиболее часто используемых функциях программного обеспечения (часто в форме учебника), специальный раздел, объясняющий, как установить данное программное обеспечение, а также справочный раздел, содержащий подробное описание каждой его функции. Это руководство, как правило, издается в виде книги, но во многих случаях эта же информация входит в само программное обеспечение. Такое решение предоставляет пользователю возможность обращаться к документации при работе с программным обеспечением. В этом случае информация может быть разбита на небольшие элементы, иногда называемые модулями справочной информации, которые могут автоматически появляться на экране, если пауза между командами пользователя становится слишком долгой.

Другая задача программной документации состоит в описании внутренней структуры программного обеспечения для поддержки последующего сопровождения системы на протяжении всего ее жизненного цикла. Данная часть документации называется *системной* и по своей сути является более технической, чем документация пользователя. 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.

Другим компонентом является проектная документация, описывающая спецификации системы и то, как они были получены. Создание этой документации – длительный процесс, начинающийся с проведения начального анализа. Из-за длительности процесса при создании подобной документации часто возникает конфликт между задачами проектирования программного обеспечения и особенностями человеческой натуры. Весьма вероятно, что начальные спецификации и начальный проект программного обеспечения будут изменены в процессе разработки. Появляется искушение выполнить эти изменения, не обновляя созданные раньше проектные документы. В результате возникает большая вероятность того, что эти документы к моменту завершения разработки окажутся неправильными и их использование в конечной документации приведет к заблуждениям.

Все сказанное выше является еще одним аргументом в пользу широкого использования инструментов CASE-технологии. Они существенно упрощают задачи повторного составления разнообразных диаграмм и обновления словарей данных в сравнении с традиционными ручными методами. Следовательно, при использовании CASE-инструментов более вероятно, что требуемые изменения действительно будут внесены и конечная документация будет точной.

В завершение подчеркнем, что пример с обновлением документации – всего лишь один из многих случаев, когда при проектировании программного обеспечения приходится бороться с недостатками, присущими человеческой натуре. Другими примерами могут служить возникновение неизбежных конфликтов персонала, проявление зависти, столкновение эгоистических интересов и прочие негативные явления, почти всегда возникающие в тех случаях, когда люди работают вместе. Поэтому, как мы отмечали раньше, проектирование программных систем (в широком смысле) включает не только те аспекты, которые непосредственно связаны с информатикой.

Вопросы для самопроверки

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?

6.7. ПРАВО СОБСТВЕННОСТИ И ОТВЕТСТВЕННОСТЬ ЗА СОЗДАВАЕМОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Не вызывает сомнения, что компания или отдельный человек должны иметь возможность возмещать затраты и получать прибыль от тех инвестиций, которые потребовались для разработки качественного программного обеспечения. Без средств защиты этих инвестиций мало кто захочет взять на себя задачу производства программного обеспечения, необходимого обществу. Но вопросы, касающиеся прав собственности на программное обеспечение и прав его владельцев, часто попадают в "провалы", существующие между хорошо разработанными законами об авторских правах и патентах. Эти законы были разработаны для того, чтобы дать возможность производителю "продукта" представить его общественности и защитить при этом его право собственности, однако особенности программного обеспечения неоднократно мешали судам в их попытках применить принципы закона об авторском праве и патентах к разрешению споров о праве собственности на программное обеспечение.

Законы об авторском праве были первоначально сформулированы для защиты авторских прав на литературные работы. В этом случае ценность произведения состоит в том, как идеи описаны, а не в идеях самих по себе. Ценность поэмы в ее рифмах, стиле и стихотворном размере, а сам предмет поэмы – далеко не главное. Ценность романа состоит в авторском представлении сюжета, а не в самом сюжете. Таким образом, труд поэта или писателя может быть защищен, если ему дается право собственности на определенное описание идеи, но не на саму идею. Другой человек имеет право описывать ту же идею, пока его описание не является "фактическим подобием" оригинала.

В противоположность поэме или роману, ценность программного обеспечения обычно состоит не в способе написания программы. Напротив, она заключается в представленном в ней алгоритме (идее). Следовательно, прямое применение законов об авторском праве не защитит инвестиции разработчика программного обеспечения. Действительно, ничто не помешает конкуренту использовать тот алгоритм, на создание которого разработчик мог затратить значительные средства, если только это представление не является "фактическим подобием" оригинала.

Коротко говоря, законы об авторском праве защищают скорее форму, а не функцию, но ценность программы чаще всего состоит в ее функции, а не в форме. В результате закон об авторском праве защищает скорее программы, реализующие широко известные, неоригинальные алгоритмы, чем инвестиции, вложенные в создание новых алгоритмов. Если алгоритм хорошо известен, единственную ценность представляет текст программы; однако если описываемый алгоритм новый и творческий, главной ценностью программы является именно алгоритм, который не защищается авторским правом. В этом есть что-то парадоксальное – чем больше творческих усилий затрачено на производство программы, тем менее вероятно, что закон об авторском праве сможет защитить вложенные в это инвестиции.

С целью применить закон об авторском праве к программному обеспечению были сделаны попытки обратиться к концепции сценария диалога с пользователем (look and feel) в системе программного обеспечения. Хотя сама фраза look and feel не использовалась до 1985 г., данная концепция уходит своими корнями в начало 1960-х гг., когда компания IBM выпустила новую серию машин System/360. Эта серия состояла из нескольких машин, предназначенных для удовлетворения самых разнообразных потребностей: от приложений мелкого бизнеса до мощных корпоративных систем. Все эти машины поставлялись с операционными системами, которые общались с окружающей средой аналогичным образом. Иными словами, машины этой серии имели стандартизованный интерфейс пользователя. По мере роста бизнеса можно было заменить используемую машину на более мощную из той же серии, не затрачивая усилий на переделку программ и переобучение персонала. Действительно, внешний вид (look) (оформление, поддерживаемое системным программным обеспечением) и способы работы (feel) (то, как пользователь взаимодействует с системным программным обеспечением) были одинаковы для всех машин этой серии.

Сегодня преимущества стандартизованного интерфейса очевидны, и разработчики стараются добиться его во всех сферах программного обеспечения. Когда интерфейс, разработанный одной компанией, становится популярным, конкурирующим компаниям выгодно разрабатывать свои системы с аналогичным сценарием диалога с пользователем. Это упрощает пользователям хорошо известной системы переход к системе конкурента даже в том случае, если внутренние проекты двух систем значительно отличаются. Компании, столкнувшиеся с такими действиями конкурентов, стали искать защиты в законах об авторском праве, объявив своей собственностью сценарий диалога с пользователем исходной системы. В конце концов, сценарий диалога с пользователем в пакете программного обеспечения имеет много особенностей, характерных для собственности, защищаемой законом об авторском праве.

Первую проверку аргумент схожести сценариев диалога с пользователем прошел в 1987 г., когда корпорация Lotus Development Corporation судилась с компанией Mosaic Software, заявив, что последняя скопировала сценарий диалога с пользователем системы электронных таблиц Lotus 1-2-3. Дело было выиграно. Однако в дальнейшем аргументы схожести сценариев диалога с пользователем имели переменный успех.

Попытка применить патентное право для защиты прав владельцев программного обеспечения тоже приводит к определенным проблемам. Одним из препятствий является давнишний принцип, что никто не может быть собственником таких явлений природы, как законы физики, математические формулы и мысли. Суды обычно считали, что алгоритмы попадают в эту же категорию. Таким образом, как и авторское право, патентное право не может защитить главную ценность программы – ее алгоритм. Кроме того, получение патента – дорогостоящий и длительный процесс, часто занимающий несколько лет. За это время программный продукт может устареть; пока патент еще не выдан, претендент на его получение имеет весьма спорное право препятствовать другим присваивать его продукт. Однако существуют и прецеденты выдачи патентов на алгоритмы. Примером может служить алгоритм кодирования RSA, активно используемый во множестве современных систем кодирования с открытым ключом.

Закон об авторском праве и патентное право разработаны с той целью, чтобы помочь распространению изобретений и расширить свободный обмен идеями на пользу обществу. Когда права собственности защищены, более вероятно, что творцы и изобретатели сделают свои достижения достоянием гласности. Напротив, закон о коммерческой тайне является средством ограничения распространения идей. Разработанный для поддержания этики взаимоотношений между конкурентами, этот

закон противодействует неуместному разглашению и незаконному присвоению внутренних достижений компании. Часто компании пытаются защитить свои коммерческие секреты путем подписания закрытых соглашений, в которых персонал, имеющий доступ к секретам компании, обязуется не раскрывать их другим. Суды в основном принимают во внимание такие соглашения.

Чтобы избежать ответственности, разработчики программного обеспечения зачастую сопровождают свои продукты оговорками, ограничивающими уровень их ответственности. Часто можно встретить предложения следующего вида: "Компания X не несет никакой ответственности за ущерб, нанесенный в связи с использованием этого программного обеспечения". Суды, однако, редко принимают во внимание такие оговорки, если истец может указать на небрежное отношение со стороны ответчика. Следовательно, при разрешении вопроса об ответственности исходят из того, насколько тщательно был разработан продукт и соответствует ли он предполагаемому использованию. Уровень тщательности, приемлемый при разработке редакторской системы, может считаться недостаточным, если речь идет о программном обеспечении для управления ядерным реактором. Поэтому при разработке программного обеспечения лучшей защитой против исков о возмещении убытков является выбор общего с заказчиком подхода к процессу разработки.

Вопросы для самопроверки

1. Какой тест можно использовать для определения того, является ли одна программа фактическим подобием другой?
2. Каким образом закон об авторском праве, патентное право и закон о коммерческой тайне служат на благо обществу?
3. В каких случаях оговорки об отказе от ответственности не принимаются во внимание судами?

Упражнения

1. Приведите пример, каким образом усилия, затраченные при разработке программного обеспечения, могут окупиться позднее при сопровождении программы.
2. Что такое пошаговая модель?
3. Охарактеризуйте, как изменило использование инструментов CASE-технологии процесс разработки программного обеспечения.
4. Объясните, как влияет на технологию разработки программного обеспечения отсутствие метрик для измерения точных характеристик программного обеспечения.
5. Чем отличаются технологии разработки программного обеспечения от традиционных технических дисциплин?
6. 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?

7. Как помогает в разработке высококачественного программного обеспечения принятие кодексов о профессиональной этике?
8. Опишите, как может упростить модификацию программного обеспечения использование констант вместо литералов?
9. В чем заключается различие между связанностью и связностью модулей? Что следует минимизировать, а что – максимизировать?
10. Какое из следующих предложений является аргументом в пользу связанности, а какое – в пользу связности?

а) При изучении предмета студентами материал должен быть представлен в виде хорошо организованных разделов, имеющих конкретные задачи.

б) Студенты не поймут предмет по-настоящему, пока не овладеют всем материалом в целом и не изучат его связи с другими предметами.

11. В тексте упоминалось понятие связанности по управлению, но эта тема не получила достаточного развития. Сравните связанность между двумя программными единицами, достигаемую с помощью команды goto, со связанностью, получаемой при использовании механизма вызова процедур.
12. Ответьте на следующие вопросы, пользуясь приведенной ниже структурной схемой:

а) Какому модулю возвращает управление модуль Y?

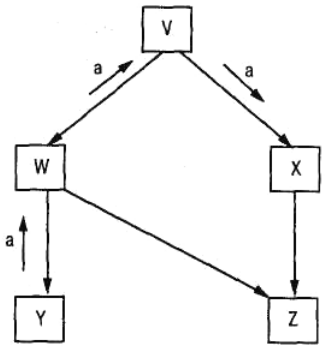
б) Какому модулю возвращает управление модуль Z?

в) Являются ли модули W и X связанными по управлению?

г) Связаны ли модули W и X данными?

д) Какие данные совместно используются модулями W и Y?

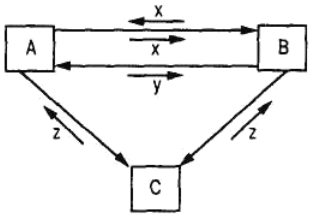
е) В каких отношениях находятся модули Y и X?



13. Используя структурную схему из предыдущей задачи, определите, какие заглушки необходимы для тестирования модуля V? Какие характеристики должны иметь эти модули заглушек?
14. Ответьте на следующие вопросы, пользуясь прилагаемой ниже схемой:

а) Чем отличаются способы использования модулями A и B элементов данных x и y?

б) Если один из модулей отвечает за получение элемента данных z от пользователя удаленного терминала, то какой именно?



15. Начертите простую диаграмму классов, представляющую взаимоотношения между издателями, журналами и подписчиками.

16. Расширьте предыдущую диаграмму классов до диаграммы совместно выполняемых действий.

17. Чем диаграмма классов отличается от диаграммы взаимодействий?

18. Что такое UML?

19. Воспользовавшись структурной схемой, представьте структуру процедур системы реального времени для обработки заказов и учета клиентов компании по доставке товаров почтой. Какие модули этой системы потребуется изменить в случае внесения изменений в закон о налогах на продажи? Что произойдет в случае изменения длины почтового индекса?

20. Разработайте решение для предыдущей задачи с использованием объектно-ориентированной парадигмы и представьте его в виде диаграммы классов.

21. Приведите несколько примеров шаблонов проектирования в областях, отличных от архитектуры и программирования.

22. Охарактеризуйте роль шаблонов проектирования в технологии разработки программного обеспечения.

23. Начертите схему потоков данных, наглядно отображающую процесс регистрации студента в университете.

24. Сравните информацию, представленную на схеме потоков данных, с информацией, отображаемой на структурной схеме.

25. Чем отличаются связи типов "один ко многим" и "многие ко многим"?

26. Начертите диаграмму "сущность – связь" для отношений между поварами, официантами, посетителями и кассирами в ресторане.

27. Начертите диаграмму "сущность – связь", представляющую отношения между журналами, издателями и подписчиками.

28. В каждом из следующих случаев определите, о чем идет речь – о структурной схеме, схеме потоков данных, диаграмме "сущность – связь" или словаре данных.

а) Определяет данные, относящиеся к разрабатываемой системе.

б) Определяет взаимоотношения между различными элементами данных, существующими в системе.

в) Определяет характеристики каждого элемента данных в системе.

г) Определяет, какие элементы данных совместно используются различными частями системы.

29. В чем отличие между диаграммой классов и диаграммой "сущность – связь"?

30. Охарактеризуйте различие между нисходящей и восходящей стратегиями проектирования.

31. В чем отличие между тестированиями по принципам "черного ящика" и "прозрачного" ящика?

32. Предположим, что перед окончательным тестированием крупной системы программного обеспечения в нее было намеренно внесено 100 ошибок. Допустим, во время этого тестирования было обнаружено и исправлено 200 ошибок, из которых 50 оказались из группы намеренно помещенных в систему. Если исправить оставшиеся 50 известных ошибок, сколько не выявленных ошибок, по-Вашему, еще останется в системе? Объясните, почему.

33. В каких случаях закон о защите авторских прав не может защитить инвестиции разработчиков программного обеспечения?

34. По каким причинам патентное право не может защитить инвестиции разработчиков программного обеспечения?

Ответы на вопросы для самопроверки

Раздел 6.1

1. В контексте разработки программы длинная последовательность операторов присваивания не сложнее нескольких вложенных операторов if.

2. Один подход заключается в преднамеренном внесении некоторых ошибок в программное обеспечение при его разработке. Затем, после того как программное обеспечение предположительно отлажено, проверяют, много ли внесенных ошибок осталось неисправленными. Если из 7 внесенных ошибок 5 было исправлено, можно сделать вывод, что только $^{5}/_{7}$ от общего количества ошибок было исправлено.

3. 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.

Раздел 6.2

1. Системные требования формулируются в терминах предметной области приложения, тогда как спецификации формулируются в технических терминах и определяют, как именно будут удовлетворяться системные требования.

2. При анализе определяются задачи, которые должна решать предполагаемая система. На стадии разработки уточняется, как именно система будет выполнять свои задачи. При реализации осуществляется реальное создание системы. Стадия

тестирования предназначена для того, чтобы удостовериться, что система действительно делает то, для чего она была предназначена.

3. Традиционный нисходящий подход требует, чтобы анализ, разработка, реализация и тестирование выполнялись строго последовательно. Модель с построением прототипов предусматривает применение более гибкого метода проб и ошибок.

Раздел 6.3

1. Главы романа следуют одна из другой в единой сюжетной линии, в то время как статьи энциклопедии в значительной степени независимы друг от друга. 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. Явное связывание включает определение козырной масти, кто пасует, кто имеет право хода и т.д. Сведения, полученные при объявлении ставок, например о том, у кого какие карты, можно рассматривать как неявную связь.

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 затем начать деление этого отдельного 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.

4. С другой стороны, предположим, что рассматриваемая задача естественным образом разделяется на три связных модуля, которые мы назовем A, B и C. Если исходный проект не будет отражать это естественное разделение (например, половина задачи A оказалась объединена в одном модуле с половиной задачи B и т.д.), то можно ожидать, что связность будет низкой, а связанность модулей – высокой. В этом случае пересмотр структуры системы с целью выделения задач A, B и C в отдельные модули, скорее всего, приведет к ослаблению связей между модулями, причем внутренняя связность модулей будет возрастать.

5. Для того чтобы придать связи персональный характер, объект класса Room мог бы использовать имя игрока при обмене сообщениями с ним. Чтобы получить это имя, объекту класса Room потребуется послать соответствующий запрос объекту класса PlayerRecord. Кроме того, объекту класса PlayerRecord может потребоваться передать объекту класса Room сведения об уровне игрока.

Раздел 6.4

1.

2.

3. В некотором смысле структуру цикла, представленную в нашем псевдокоде оператором while, можно рассматривать как шаблон проектирования. Другим хорошим примером является модель "клиент/сервер".

4. Исследователи надеются, что структуры будут служить готовыми строительными блоками, из которых можно будет собирать большие системы программного обеспечения подобно тому, как из готовых компонентов собираются различные сложные приборы.

Раздел 6.5

1. Задача тестирования программного обеспечения состоит в обнаружении ошибок. Поэтому разработчики программного обеспечения рассматривают как неудачный такой тест, который не позволил обнаружить новую ошибку.

2. Можно оценить 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.

3. Анализ крайних значений предполагает, что тестирование программного обеспечения будет выполнено 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 6.6

1. В виде сопроводительной документации; внутри программы в форме комментариев и ясно написанного кода; в виде интерактивных сообщений, которые программа может выводить на экран дисплея; в виде словарей данных и в виде проектной документации, такой, как структурные схемы, диаграммы классов, схемы потоков данных и диаграммы "сущность – связь".

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 будут описаны в руководстве пользователя. Более того, широко распространены книги, написанные об использовании и разработке популярных систем программного обеспечения. Иногда их пишут люди, которые не принимали участия в их разработке, после того, как программное обеспечение уже некоторое время использовалось и приобрело определенную популярность.)

3. Разные люди будут иметь различные мнения об этом. Некоторые 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 также невозможно. Более того, при наличии хорошей документации задача создания программы может быть "легко" решена заново.

Раздел 6.7

1. Это проблема, которая относится к компетенции судебных органов. Ее решение определенно затронуло бы много дополнительных аспектов, помимо учета формата программы и выбора имен переменных.

2. Авторские права и патентное законодательство приносят обществу выгоду, поскольку они стимулируют создателей новой продукции делать ее общедоступной. Законы о коммерческой тайне также приносят обществу пользу, поскольку они позволяют компании защищать от конкурентов свою продукцию, пока она находится на стадии разработки.

3. Отказ от ответственности не защищает компанию от юридического преследования за небрежность.

СПИСОК ЛИТЕРАТУРЫ

1. Брукшир, Дж. Глен. Введение в компьютерные науки. Общий обзор / Дж. Глен Брукшир. – 6-е изд. – М. : Издательский дом "Вильямс", 2001. – 688 с.
2. Симонович, С.В. Информатика : базовый курс / С.В. Симонович и др. – СПб. : Питер, 2002. – 640 с.
3. Острейковский, В.А. Информатика : учебник для вузов / В.А. Острейковский – М. : Высшая школа, 1999. – 511 с.
4. Каймин, В.А. Информатика : учебник для вузов / В.А. Каймин. – М. : ИНФРА-М, 2000. – 232 с.
5. Информатика : учебное пособие для пед. учеб. заведений / А.В. Могилев и др. ; под ред. Е.К. Хеннера. – М. : Академия, 2000. – 816 с.
6. Информатика : энциклопед. словарь для начинающих / под общ. ред. Д.А. Поспелова. – М. : Педагогика Пресс, 1994. – 352 с.
7. Экономическая информатика / под ред. П.В. Конюховского, Д.Н. Колесова. – СПб. : Питер, 2000. – 560 с.
8. Информатика для юристов и экономистов / под ред. С.В. Симоновича. – СПб. : Питер, 2001. – 688 с.
9. Информатика : учебник для вузов / под ред. Н.В. Макаровой. – 3-е изд., перераб. – М. : Финансы и статистика, 2001. – 768 с.
10. Жаров, А.В. "Железо" IBM 99 или все о современном компьютере / А.В. Жаров. – 6-е изд., испр. и доп. – М. : МикроАрт, 1999. – 352 с.
11. Компьютерные сети : учебный курс / пер. с англ. – 2-е изд., испр. и доп. – М. : Изд. отдел "Рус. редакция" ТОО "Channel Traing Ltd", 1998. – 696 с.
12. Бекон, Д. Операционные системы / Д. Бекон, Т. Харрис. – СПб. : Питер, 2004. – 800 с.
13. Олифер, В.Г. Компьютерные сети: Принципы, технологии, протоколы : учебное пособие для вузов / В.Г. Олифер, Н.А. Олифер. – СПб. : Питер, 2001. – 672 с.
14. Анин, Б.Ю. Защита компьютерной информации / Б.Ю. Анин. – СПб. : БХВ-Санкт-Петербург, 2000. – 384 с.
15. Епанешников, А.М. Программирование в среде TurboPascal 7.0 / А.М. Епанешников, В.А. Епанешников. – М. : ДиалогМИФИ, 1995. – 288 с.
16. Системный анализ в информационных технологиях / Ю.Ю. Громов, Н.А. Земской, А.В. Лагутин, О.Г. Иванова, В.М. Тютюнник. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2004. – 176 с.
17. Немнюгин, С.А. TurboPascal : практикум / С.А. Немнюгин. – СПб. : Питер, 2001. – 256 с.
18. Решение инженерных и экономических задач на языке С++ / Ю.Ю. Громов, С.И. Татаренко, В.М. Тютюник, А.В. Лагутин, О.Г. Иванова. – Тамбов : Изд-во МИНЦ, 2003. – 312 с.
19. Орлов, С. Технология разработки программного обеспечения / С. Орлов. – СПБ. : Питер, 2002. – 464 с.

А. АРИФМЕТИКО-ЛОГИЧЕСКИЕ ОСНОВЫ АРХИТЕКТУРЫ КОМПЬЮТЕРОВ

Позиционные системы счисления. Основные понятия. Система счисления – это совокупность правил и приемов записи чисел с помощью набора цифровых знаков (алфавита). Различают два типа систем счисления: позиционные, когда значение каждой цифры числа определяется ее местом (позицией) в записи числа; и непозиционные, когда значение цифры в числе не зависит от ее места в записи числа. Примером непозиционной системы счисления являются римские цифры: IX, IV, XV, LX и т.д., а примером позиционной системы счисления можно назвать арабские цифры, используемые нами повседневно: 12, 67, 329 и т.д.

Позиционные системы счисления характеризуются *основанием* – количеством знаков или символов, используемых в разрядах для изображения числа в данной системе счисления.

Системы счисления	Значения							
Десятеричная	0	1	2	3	4	5	6	7
Двоичная	0	1	10	11	100	101	110	111
Восьмеричная	0	1	2	3	4	5	6	7
Шестнадцатеричная	0	1	2	3	4	5	6	7
Десятеричная	8	9	10	11	12	13	14	15
Двоичная	1000	1001	1010	1011	1100	1101	1110	1111
Восьмеричная	10	11	12	13	14	15	16	17
Шестнадцатеричная	8	9	A	B	C	D	E	F

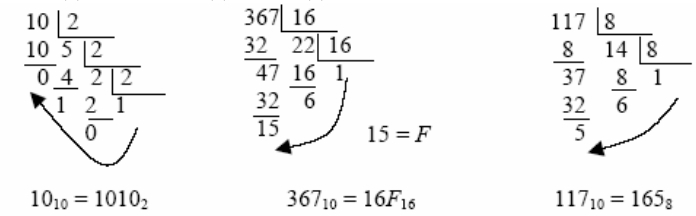
Для позиционной системы счисления с общим основанием s справедливо следующее равенство, позволяющее в то же время переводить произвольное число X_s в десятичную систему счисления:

$$X_s = \{ A_{n-1} A_{n-2} \ldots A_1 A_0, A_{-1} A_{-2} \ldots A_{-m} \}_s = \\ = A_{n-1} S^{n-1} + A_{n-1} S^{n-2} + \ldots + A_1 S^1 + A_0 S^0 + A_{-1} S^{-1} + A_{-2} S^{-2} + \ldots + A_{-m} S^{-m},$$

где A_i – цифры в системе счисления s; n, m – количество целых и дробных разрядов в числе X_s.

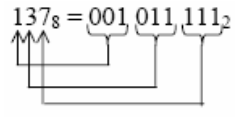
Например: $2971.3_{10} = 2 \cdot 10^3 + 9 \cdot 10^2 + 7 \cdot 10^1 + 1 \cdot 10^0 + 3 \cdot 10^{-1}$;
$1010.1_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} = 10.5_{10}$;
$16F_{16} = 1 \cdot 16^2 + 6 \cdot 16^1 + 15 \cdot 16^0 = 367_{10}$.

Для обратного перевода из десятеричной системы счисления в систему счисления с основанием s необходимо выполнить последовательное деление десятичного числа на основание s и прочитать число в обратном порядке.

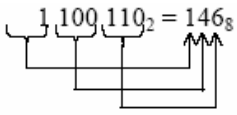


$10_{10} = 1010_2$ $\quad 367_{10} = 16F_{16}$ $\quad 117_{10} = 165_8$

Переход от восьмеричной системы счисления к двоичной осуществляется заменой каждой восьмеричной цифры трехзначным двоичным числом (триадой).

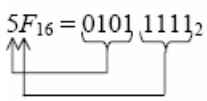


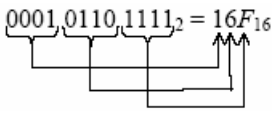
Обратный переход от двоичной системы счисления осуществляется заменой каждой триады, начиная справа, восьмеричной цифрой.



При этом слева любого числа можно приписать сколько угодно нулей, не изменив начального значения числа.

Аналогично осуществляется переход от шестнадцатеричной системы счисления к двоичной и обратно, только вместо триад берутся четырехзначные двоичные числа (тетрады).





Правила выполнения арифметические операций в десятичной системе хорошо известны – это сложение, вычитание, умножение столбиком и деление углом. Эти правила применимы и ко всем другим позиционным системам счисления. Только таблицами сложения и умножения надо пользоваться особыми для каждой системы.

а) двоичная система

+	0	1
0	0	1
1	1	10

б) восьмеричная система

+	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	10
2	2	3	4	5	6	7	10	11
3	3	4	5	6	7	10	11	12
4	4	5	6	7	10	11	12	13
5	5	6	7	10	11	12	13	14
6	6	7	10	11	12	13	14	15
7	7	10	11	12	13	14	15	16

в) шестнадцатеричная система

+	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

При сложении цифры суммируются по разрядам, и если при этом возникает избыток, то он переносится влево. Например, сложим числа 15 и 6 в различных системах счисления.

Десятичная: $15_{10} + 6_{10}$

$$\begin{array}{r} {\scriptstyle 1} \\ 15 \\ +\ 6 \\ \hline 21 \end{array}$$

$5 + 6 = 11 = 10 + 1$

$1 + 1 = 2$

Двоичная: $1111_2 + 110_2$

$$\begin{array}{r} {\scriptstyle 111} \\ 1111 \\ +\ 0110 \\ \hline 10101 \end{array}$$

$1 + 0 = 1$

$1 + 1 = 2 = 2 + 0$

$1 + 1 + 1 = 3 = 2 + 1$

$1 + 1 = 2 = 2 + 0$

Восьмеричная: $17_8 + 6_8$

$$\begin{array}{r} {\scriptstyle 1} \\ 17 \\ +\ 6 \\ \hline 25 \end{array}$$

$7 + 6 = 13 = 8 + 5$

$1 + 1 = 2$

Шестнадцатеричная: $F_{16} + 6_{16}$

$$\begin{array}{r} {\scriptstyle 1} \\ F \\ +\ 6 \\ \hline 15 \end{array}$$

$15 + 6 = 21 = 16 + 5$

Выполняя умножение многозначных чисел в различных позиционных системах счисления, можно использовать обычный алгоритм перемножения чисел в столбик, но при этом результаты перемножения и сложения однозначных чисел необходимо заимствовать из соответствующих рассматриваемой системе таблиц умножения и сложения.

Деление в любой позиционной системе счисления производится по тем же правилам, как и деление углом в десятичной системе. В двоичной системе деление выполняется особенно просто, ведь очередная цифра частного может быть только нулем или единицей.

Для отображения вещественных чисел, которые могут быть как очень маленькими, так и очень большими, используется форма записи чисел с *порядком основания системы счисления*. Например, десятичное число 1.25 в этой форме можно представить так:

$$1.25 \cdot 10^0 = 0.125 \cdot 10^1 = 0.0125 \cdot 10^2 = \ldots$$

или так: $12.5 \cdot 10^{-1} = 125.0 \cdot 10^{-2} = 1250.0 \cdot 10^{-3} = \ldots$.

Любое число X в системе счисления с основанием s можно записать в виде $X = Ms^p$, где M – множитель, содержащий все цифры числа (*мантисса*), а p – целое число, называемое *порядком*. Такой способ записи чисел называется *представлением числа с плавающей точкой*.

Если мантисса числа является правильной дробью, у которой первая цифра после точки отлична от нуля, то такое число называется *нормализованным*.

Мантиссу и порядок s-ичного числа принято записывать в системе с основанием s, а само основание – в десятичной системе. Примеры нормализованного представления:

а) десятичная система $753.15 = 0.75315 \cdot 10^3$; $-0.000034 = -0.34 \cdot 10^{-4}$;

б) двоичная система $-101.01 = -0.10101 \cdot 2^{11}$ (порядок $11_2 = 3_{10}$);

$0.000011 = 0.11 \cdot 2^{-100}$; (порядок $-100_2 = -4_{10}$).

Основные понятия алгебры логики. *Алгебра логики* – это раздел математики, изучающий высказывания, рассматриваемые со стороны их логических значений (истинности или ложности) и логических операций над ними. Алгебра логики

возникла в середине XIX в. в трудах английского математика Джорджа Буля. Ее создание представляло собой попытку решать традиционные логические задачи алгебраическими методами.

Одним из основных понятий алгебры логики является логическое высказывание. *Логическое высказывание* – это любое повествовательное предложение, в отношении которого можно однозначно сказать, истинно оно или ложно.

Так, например, предложение "*6 – четное число*" следует считать высказыванием, так как оно истинное. Предложение "*Рим – столица Франции*" тоже высказывание, так как оно ложное.

Разумеется, не всякое предложение является логическим высказыванием. Высказываниями не являются, например, предложения "*студент первого курса*" и "*информатика – интересный предмет*". Первое предложение ничего не утверждает о студенте, а второе использует слишком неопределенное понятие "*интересный предмет*". Вопросительные и восклицательные предложения также не являются высказываниями, поскольку говорить об их истинности или ложности не имеет смысла.

Алгебра логики рассматривает любое высказывание только с одной точки зрения – является ли оно истинным или ложным. Заметим, что зачастую трудно установить истинность высказывания. Так, например, высказывание "*площадь поверхности Индийского океана равна 75 млн. кв. км*" в одной ситуации можно посчитать ложным, а в другой – истинным. Ложным – так как указанное значение неточное и вообще не является постоянным. Истинным – если рассматривать его как некоторое приближение, приемлемое на практике.

Употребляемые в обычной речи слова и словосочетания "*не*", "*и*", "*или*", "*если ..., то*", "*тогда и только тогда*" и другие позволяют из уже заданных высказываний строить новые высказывания. Такие слова и словосочетания называются *логическими связками*.

Высказывания, образованные из других высказываний с помощью логических связок, называются *составными.* Высказывания, не являющиеся составными, называются *элементарными*.

Так, например, из элементарных высказываний "*Петров – студент*", "*Петров – шахматист*" при помощи связки "и" можно получить составное высказывание "*Петров – студент и шахматист*", понимаемое как "*Петров – студент, хорошо играющий в шахматы*".

При помощи связки "или" из этих же высказываний можно получить составное высказывание "*Петров – студент или шахматист*", понимаемое в алгебре логики как "*Петров или студент, или шахматист, или и студент и шахматист одновременно*".

Истинность или ложность получаемых таким образом составных высказываний зависит от истинности или ложности элементарных высказываний.

Чтобы обращаться к логическим высказываниям, им назначают имена. Пусть через *A* обозначено высказывание *"Тимур поедет летом на море",* а через *B* – высказывание *"Тимур летом отправится в горы".* Тогда составное высказывание *"Тимур летом побывает и на море, и в горах"* можно кратко записать как *A* и *B*. Здесь "и" – логическая связка, *A*, *B – логические переменные*, которые могут принимать только два значения – "истина" или "ложь", обозначаемые, соответственно, "1" и "0".

Каждая логическая связка рассматривается как операция над логическими высказываниями (или логическими переменными) и имеет свое название и обозначение:

НЕ Операция, выражаемая связкой "не", называется *отрицанием* и обозначается чертой над высказыванием (или знаком ¬). Высказывание $\overline{A}$ истинно, когда *A* ложно, и ложно, когда *A* истинно. Зависимость между такими высказываниями можно записать в виде таблицы истинности (рис. А.1). Пример: "*Луна – спутник Земли*" (*A*); "*Луна – не спутник Земли*" ($\overline{A}$).

A	$\overline{A}$
0	1
1	0

а)

A	B	A ∧ B
0	0	0
0	1	0
1	0	0
1	1	1

б)

A	B	A ∨ B
0	0	0
0	1	1
1	0	1
1	1	1

в)

Рис. А.1

И Операция, выражаемая связкой "и", называется *конъюнкцией* (лат. conjunctio – соединение) или *логическим умножением* и обозначается точкой " · " (может также обозначаться знаками ∧ или **&**). Высказывание $A \cdot B$ истинно тогда и только тогда, когда оба высказывания *A* и *B* истинны (рис. 1, *б*). Например, высказывание "*10 делится на 2 и 5 больше 3*" истинно, а высказывания "*10 делится на 2 и 5 не больше 3*", "*10 не делится на 2 и 5 больше 3*", "*10 не делится на 2 и 5 не больше 3*" – ложны.

ИЛИ Операция, выражаемая связкой "или", называется *дизъюнкцией* (лат. disjunctio – разделение) или логическим сложением и обозначается знаком ∨ (или плюсом). Высказывание A ∨ B ложно тогда и только тогда, когда оба высказывания A и B ложны (рис. 1, *в*). Например, высказывание "*10 не делится на 2 или 5 не больше 3*" ложно, а высказывания "*10 делится на 2 или 5 больше 3*", "*10 делится на 2 или 5 не больше 3*", "*10 не делится на 2 или 5 больше 3*" – истинны.

Есть и другие логических операций, однако их можно выразить через отрицание, дизъюнкцию и конъюнкцию. Таким образом, операций отрицания, дизъюнкции и конъюнкции достаточно, чтобы описывать и обрабатывать логические высказывания.

Порядок выполнения логических операций задается круглыми скобками. Но для уменьшения числа скобок договорились считать, что сначала выполняется операция отрицания ("не"), затем конъюнкция ("и"), после конъюнкции – дизъюнкция ("или").

С помощью логических переменных и символов логических операций любое высказывание можно формализовать, т.е.

заменить логической формулой.

Определение *логической формулы* следующее:

1. Всякая логическая переменная и символы "истина" ("1") и "ложь" ("0") – формулы.
2. Если A и B – формулы, то $\overline{A}$, $A \cdot B$, $A \vee B$ – формулы.
3. Никаких других формул в алгебре логики нет.

В п. 1 определены элементарные формулы; в п. 2 даны правила образования из любых данных формул новых формул.

Некоторые формулы принимают значение "истина" при любых значениях истинности входящих в них переменных. Таковой будет, например, формула $A \vee \overline{A}$, соответствующая высказыванию "*Этот треугольник прямоугольный или косоугольный*". Эта формула истинна и тогда, когда треугольник прямоугольный, и тогда, когда треугольник не прямоугольный. Такие формулы называются *тождественно истинными формулами* или *тавтологиями*. Высказывания, которые формализуются тавтологиями, называются логически истинными высказываниями.

В качестве другого примера рассмотрим формулу $A \wedge \overline{A}$, которой соответствует, например, высказывание "*Иванов самый высокий студент в группе, и в группе есть студенты выше Иванова*". Очевидно, что эта формула ложна, так как либо A, либо $\overline{A}$ обязательно ложно. Такие формулы называются *тождественно ложными формулами* или *противоречиями*. Высказывания, которые формализуются противоречиями, называются логически ложными высказываниями.

Если две формулы A и B одновременно, т.е. при одинаковых наборах значений входящих в них переменных, принимают одинаковые значения, то они называются *равносильными*.

Равносильность двух формул алгебры логики обозначается символом "=" или символом "≡" Замена формулы другой, ей равносильной, называется равносильным преобразованием данной формулы.

Б. ЭЛЕКТРОННЫЕ СХЕМЫ ОБРАБОТКИ ЧИСЕЛ В ДВОИЧНОМ ДОПОЛНИТЕЛЬНОМ КОДЕ

В этом приложении описаны электронные схемы, предназначенные для изменения знака числа на противоположный и для сложения чисел, представленных в двоичном дополнительном коде. Начнем обсуждение со схемы, представленной на рис. Б.1. Она позволяет преобразовать четырехразрядное число в двоичном дополнительном коде в битовую комбинацию, которая представляет то же число, но с противоположным знаком. Например, двоичный код числа 3 будет преобразован в двоичное представление числа –3. Данная схема выполняет данную операцию в соответствии с алгоритмом, описанным в главе 1. Это значит, что 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 самого правого 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.

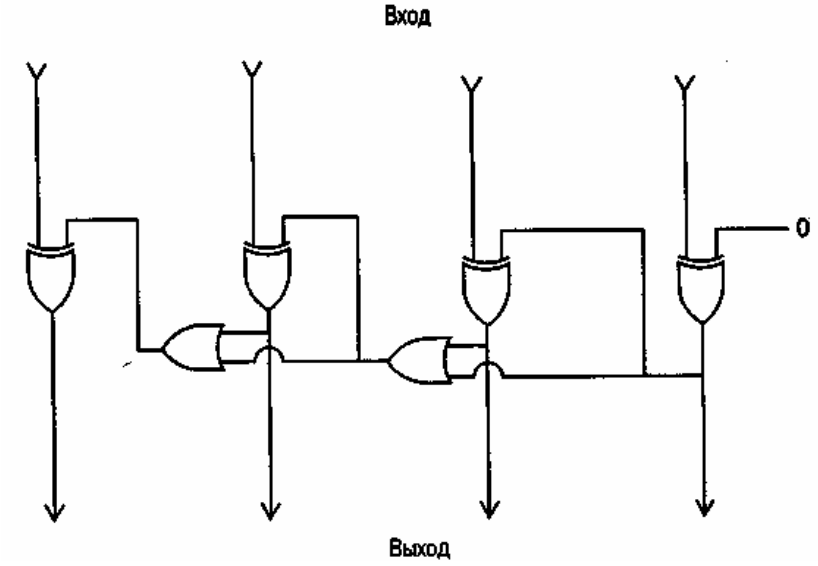


Рис. Б.1. Электронная схема, изменяющая знак числа в двоичном дополнительном коде на противоположный

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.

Далее рассмотрим процесс сложения двух чисел, представленных в двоичном дополнительном коде. Например, рассмотрим решение следующей задачи:

0110
+ 1001

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.

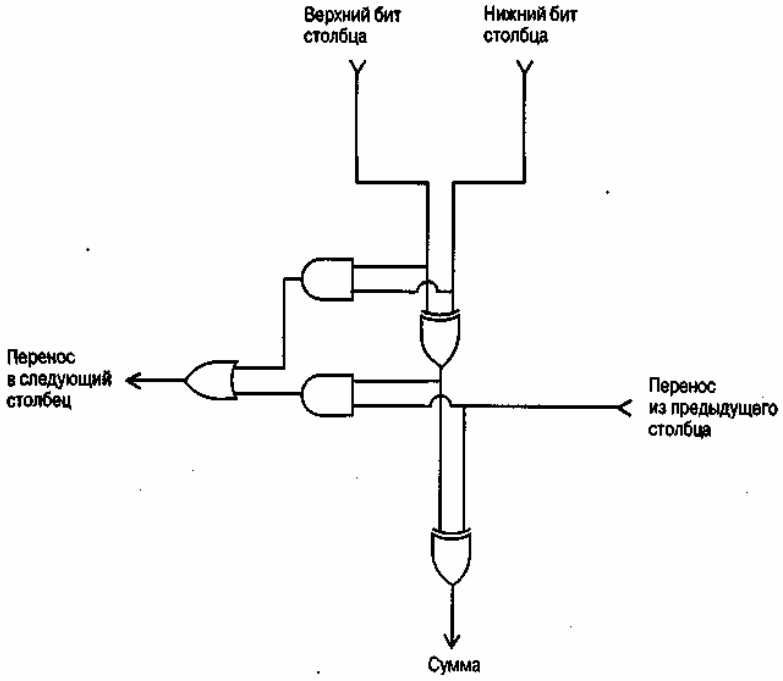


Рис. Б.2. Электронная схема для сложения значений в отдельном столбце

Алгоритм сложения значений в отдельном столбце для задачи сложения чисел из нескольких столбцов состоит в следующем. Требуется сложить два значения в текущем столбце, добавить эту сумму к биту, перенесенному из предыдущего столбца, записать младший значащий бит этой суммы в бит результата и перенести значение избыточного бита в следующий столбец. Электронная схема, реализующая этот алгоритм, представлена на рис. Б.2. В 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 налево. Поэтому если в данном столбце оба суммируемых бита равны 1 или сумма входных битов и бит переноса одновременно равны 1, то в соседний разряд будет перенесено значение 1.

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

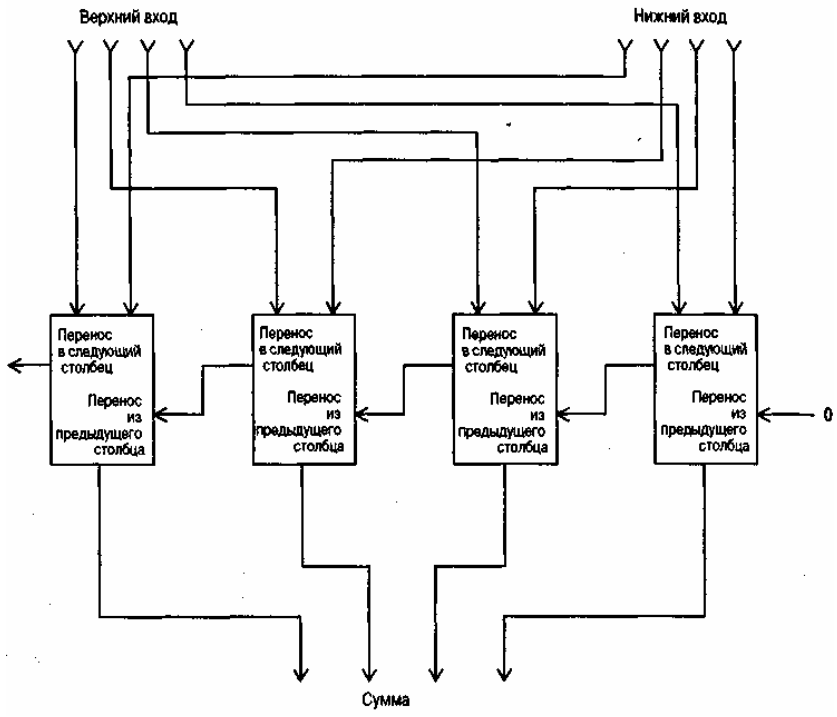


Рис. Б.3. Электронная схема сложения двух четырехразрядных чисел в двоичном дополнительном коде, построенная из четырех копий схемы, представленной на рис. Б.2

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 левого прямоугольника просто игнорируется.

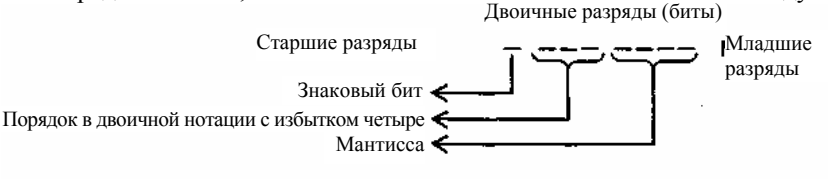
Схема на рис. Б.3 называется сумматором со сквозным переносом, поскольку перенос должен проходить сквозь всю схему, от крайнего правого до крайнего левого столбца. Несмотря на простоту реализации, такие схемы медленнее выполняют свои функции по сравнению с более совершенными схемами, такими как сумматор с ускоренным переносом, минимизирующий переносы от столбца к столбцу. Поэтому схема, изображенная на рис. Б.3, хотя и подходит для наших целей, тем не менее, в современных вычислительных машинах не используется.

В. ПРИМЕР ТИПИЧНОГО МАШИННОГО ЯЗЫКА

Архитектура машины. Рассматриваемая гипотетическая машина имеет 16 регистров общего назначения, пронумерованных от 0 до F (в шестнадцатеричной системе счисления). Длина каждого регистра равна одному байту (восьми битам). Для идентификации регистров в машинных командах каждому регистру присвоен уникальный четырехразрядный двоичный код, который представляет собой номер этого регистра. Таким образом, регистр 0 идентифицируется как 0000 (шестнадцатеричный 0), а регистр 4 – как 0100 (шестнадцатеричное 4).

Поскольку память рассматриваемой машины состоит из 256 ячеек, каждая ячейка будет иметь уникальный адрес, представляющий собой целое число в диапазоне от 0 до 255. Следовательно, адрес любой ячейки памяти может быть представлен восьмибитовыми числами от 00000000 до 11111111 (в шестнадцатеричном представлении от 00 до FF).

Предполагается, что числа с плавающей запятой хранятся в следующем формате:



Машинный язык. Длина каждой машинной команды равна двум байтам. Первые четыре бита содержат код операции, последние 12 битов образуют поле операндов. В приведенной ниже таблице перечислены и кратко описаны команды, показанные в шестнадцатеричном представлении. Буквы R, S и T используются для указания в поле операндов позиции шестнадцатеричных цифр, являющихся идентификаторами регистров, которые меняются в зависимости от конкретной команды. Буквы X и Y используются для указания в поле операндов позиций тех шестнадцатеричных цифр, которые не являются идентификаторами регистров.

Таблица

Код операции	Операнд	Описание
1	RXY	Загрузка в регистр R двоичного кода числа из ячейки памяти с адресом XY. *Пример.* Команда 14A3 помещает в регистр 4 содержимое ячейки памяти с адресом A3
2	RXY	Загрузка в регистр R двоичного кода числа XY. *Пример.* Команда 20A3 помещает в регистр 0 значение A3
3	RXY	Сохранение двоичного кода числа, хранящегося в регистре R, в ячейке памяти с адресом XY. *Пример.* Команда 35B1 помещает содержимое регистра 5 в ячейку памяти с адресом B1
4	0RS	Перемещение двоичного кода числа из регистра R в регистр S. *Пример.* Команда 40A4 копирует содержимое регистра A в регистр 4
5	RST	Суммирование двоичных кодов чисел, хранящихся в регистрах S и T, с сохранением суммы в регистре R. *Пример.* Команда 5726 суммирует двоичные коды чисел, хранящиеся в регистрах 2 и 6, а сумму помещает в регистр 7

6	RST	Суммирование двоичных кодов чисел в формате с плавающей запятой, хранящихся в регистрах S и T, с размещением результата в формате с плавающей запятой в регистре R. *Пример.* Команда 634E суммирует числа в формате с плавающей запятой, хранящиеся в регистрах 4 и E, и помещает результат в регистр 3
7	RST	Выполнение поразрядной операции OR над двоичными кодами чисел, хранящихся в регистрах S и T, и помещение результата в регистр R. *Пример.* Команда 7CB4 помещает в регистр 7 результат операции OR над содержимым регистров B и 4
8	RST	Выполнение поразрядной операции AND над двоичными кодами чисел, хранящихся в регистрах S и T, и помещение результата в регистр R. *Пример.* Команда 8045 помещает в регистр 0 результат операции AND над содержимым регистров 4 и 5
9	RST	Выполняется поразрядная операция XOR над двоичными кодами чисел, хранящихся в регистрах S и T, и результат помещается в регистр R. *Пример.* Команда 95F3 помещает в регистр 5 результат операции XOR над содержимым регистров F и 4
A	R0X	Выполняется операция циклического поразрядного сдвига вправо на X позиций над двоичным кодом числа, хранящегося в регистре R. При каждом одиночном сдвиге бит из младшего разряда перемещается в старший разряд. *Пример.* Команда A403 выполняет циклический поразрядный сдвиг вправо на 3 бита в содержимом регистра 4
B	RXY	Выполняется переход к команде, размещенной в ячейке памяти по адресу XY, если двоичный код числа в регистре R совпадает с двоичным кодом числа в регистре 0. *Пример.* Команда B43C сначала сравнивает содержимое регистра 4 с содержимым регистра 0. Если они равны, последовательность выполнения команд изменится так, что следующей будет выполнена команда, расположенная в памяти по адресу 3C. В противном случае выполнение программы продолжится в обычной последовательности
C	000	Прекращение выполнения программы. *Пример.* Команда C000 останавливает выполнение программы

Г. ПРИМЕРЫ ПРОГРАММ

В этом приложении приведены примеры программ на языках Ada, C, C++, FORTRAN, Java и Pascal. Каждая из программ получает на вход список имен, поступающий с клавиатуры, сортирует его с помощью алгоритма сортировки вставками и выводит отсортированный список на экран дисплея.

Язык Ada. Язык программирования Ada, названный в честь Августы Ады Байрон (Augusta Ada Byron) (1815 – 1851), 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 упор 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 окружающей среды, системы управления в автомобилях и небольшие домашние системы управления. В результате в язык Ada были включены возможности про-

граммирования параллельных процессов, а также удобные средства для обработки особых случаев (называемых исключительными ситуациями), которые могут возникать при работе систем. Самая современная версия языка Ada, известная как Ada 95, охватывает также объектно-ориентированную парадигму программирования.

Пример программы на языке Ada приведен на рис. Г.1.

Язык С был разработан в начале 1970-х гг. Деннисом Ритчи (Dennis Ritchie), работавшим в то время в компании Bell Laboratories. Хотя первоначально язык С 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 С 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 С представляются с помощью специальных символов. Эта лаконичность является одной из причин чрезвычайной популярности языка С, поскольку позволяет программистам эффективно выражать сложные алгоритмы. (Часто краткое представление алгоритма более доступно пониманию, чем его пространное описание.)

Пример программы на языке С приведен на рис. Г.2.

```
--Программа обработки списка
with TEXT_IO;
use TEXT_IO;
procedure MAIN is
   subtype NAME.TYPE is STRING (1..8);
   LIST_LENGTH: constant:=10;
   NAMES: array (1..LIST_LENGTH) of NAME_TYPE;
   PIVOT: NAME_TYPE; HOLE: INTEGER;
begin
--Прежде всего получаем список имен с клавиатуры
   for K in 1 .. LIST_LENGTH loop
      GET(NAMES(K));
   end loop;
--Сортируем список (переменная HOLE содержит номер пропуска в
--списке с момента удаления элемента из списка и до момента
--его повторной вставки)
   for N in 2 .. LIST_LENGTH loop
      PIVOT := NAMES(N);
      HOLE := N;
      for M in reverse 1 .. N-l loop
         if NAMES(M) > PIVOT
            then NAMES(M+l) := NAMES(M);
            else exit;
         end if;
         HOLE := M;
      end loop;
      NAMES(HOLE) := PIVOT;
   end loop;
--Теперь печатаем отсортированный список
   for K in 1 .. LI STRENGTH loop
      NEW_LINE;
      PUT(NAMES(K));
   end loop;
end MAIN;
```

Рис. Г.1. Пример программы на языке Ada

```
/* Программа обработки списка */
#include <stdio.h>
#include <string.h>
main()
{
   char names[10][9], pivot[9];
   int i, j;
/* Ввод имен с клавиатуры */
   for(i = 0; i < 10; ++i)
      scanf("%s", names[i]);
/* Сортировка списка имен */
   for(i = 1; i < 10; ++i)
   {  strcpy(pivot,names[i]);
      j = i - 1;
      while((j>=0) && (strcmp(pivot, names[j]) < 0)
      {  strcpy(names[j+1], names[j]);
         --j;
      }
      strcpy(names[j+1],pivot);
   }
/* Печать отсортированного списка */
   for( i = 0; i < 10; ++i)
      printf("%s\n", names[i]);
}
```

Рис. Г.2. Пример программы на языке С

Язык С++ был разработан Бьярни Страуструпом (Bjarne Strausstrup) из компании Bell Laboratories как усовершенствованная версия языка С. Цель создания языка С++ – достижение совместимости языка С с объектно-ориентированной парадигмой программирования.

Реализация алгоритма сортировки вставками на языке С++ представлена на рис. Г.3. Последние четыре оператора этой программы требуют, чтобы объект namelist был создан как объект, имеющий "тип" (класс) list, после чего новый объект должен выполнить операции getnames, sortnames и printnames. Предыдущий фрагмент программы определяет свойства, которыми должны обладать любые объекты "типа" (класса) list. В частности, любой такой объект должен содержать массив символов под названием names и три процедуры – getnames, sortnames и printnames. Обратите внимание, что определения этих

```
// Программа обработки списка
#include <iostreain.h>
#include <string.h>
const int ListLength = 10;
// Все объекты класса list содержат список имен и три открытых
// метода, которые называются getnames, sortlist и printnames.
class list
{  private:
      char names[ListLength][9];
   public:
      void getnames()
      {  int i;
         for(i = 0; i< ListLength; *+i)
            cin » names[i];
      }
      void sortlist()
      {  int i,j;
         char pivot[9];
         for(i = 1; i < ListLength; ++i)
         {  strcpy(pivot, names[i]);
            j = i - 1;
            while((j >= 0) && (stremp(pivot, names[j]) < 0))
            {  strcpy(names[j+1], names[j]);
               --j;
            }
            strcpy(names[j+1],pivot);
         }
```

```
        }
        void printnames()
        {   int i;
            cout « endl;
            for(i = 0; i < ListLength; ++i)
                cout « names[i] « endl;
        }
}
// Создание объекта с именем namelist и обращение к нему с
// требованием ввести несколько имен, отсортировать их, а
// затем вывести отсортированный список на экран.
void main()
{   list namelist;
    namelist.getnames();
    namelist.sortlist();
    namelist.printnames();
}
```

Рис. Г.3. Пример программы на языке C++

процедур такие же, как и в программе на языке C, представленной на рис. Г.2. Отличие заключается в том, что в программе на языке C++ эти операции рассматриваются как часть свойств объекта, в то время как в программе на языке C они считаются отдельными модулями той части программы, где эти процедуры описываются.

Язык FORTRAN. FORTRAN (сокращение от FORmula TRANslator – транслятор формул) был одним из первых языков программирования высокого уровня (впервые опубликован в 1957 г.) и первым языком, получившим широкое признание в компьютерном сообществе. Со временем его официальное описание претерпело многочисленные изменения, так что теперь можно встретить ссылки на языки FORTRAN IV и FORTRAN 77. Последним в этом ряду стоит FORTRAN 90, который является дальнейшим развитием языка FORTRAN 77 и обладает такими свойствами, как рекурсия и определяемые пользователем типы данных.

Несмотря на то, что язык FORTRAN критикуется многими авторами, он остается популярным в научном мире. В частности, многие пакеты программ по численному анализу и статистике написаны и, вероятно, до сих пор пишутся на языке FORTRAN. Пример программы на этом языке представлен на рис. Г.4.

```
      ! Программа обработки списка
        INTEGER J,K
        CHARACTER(LEN=8) Pivot
        CHARACTER(LEN=8) DIMENSION(10) Names
      ! Сначала получаем имена
        READ(UNIT=5, FMT=100) (Names(K), K=l,10)
      100 FORMAT(A8)
      ! Теперь сортируем список
        OuterLoop: DO J=2,10
          Pivot = Names(J)
          InnerLoop: DO K=J-1,1,-1
            IF (Names(K) .GT. Pivot)
              THEN Names(K+l) = Names(K)
              ELSE InnerLoop
            ENDIF
          END DO InnerLoop
          Names (K+l) = Pivot
        END DO OuterLoop
      ! Теперь выводим отсортированный список на экран
        WRITE(UNIT=6,FMT=400) (Names(K), K=l,10)
      400 FORMAT(",A8)
        END
```

Рис. Г.4. Пример программы на языке FORTRAN

```
// Программа обработки списка
import java.io.*
// Все объекты класса list содержат список имен и три открытых
// метода, которые называются getnames, sortlist и printnames.
class list
{   final int ListLength = 10;
    private String[] names;
    public list()
```

```
    {   names = new String[ListLength]
    }
    public void getnames{)
    {   int i;
        DataInput data = new DataInputStream(System.in);
        for(i = 0; i < ListLength; i++)
        {   try( names[i] = data.readLine();
            catch(IOException e){};
        }
    public void sortnames()
    {   int i,j;
        String pivot;
        for(i = 1; i < ListLength; i++)
        {   pivot = names[i]);
            j = i - 1;
            while((j >= 0) && (pivot.compareTo(names[j]) < 0))
            {   names[j+1]=names[j];
                j--;
            }
            names[j+1] = pivot;
        }
    }
    public void printnames()
    {   int i;
        for(d = 0; i < ListLength; i++)
            System.out.println(names[i]);
    }
}
// Создание объекта с именем namelist и обращение к нему с
// требованием ввести несколько имен, отсортировать их, а
// затем вывести отсортированный список на экран.
class sort
{   public static void main (String args[])
    {   list namelist = new list();
        namelist.getnames();
        namelist.sortnames();
        namelist.printnames();
    }
}
```

Рис. Г.5. Пример программы на языке Java

```
{Программа обработки списка}
program InsertSort(Input, Output);
const Blanks = '          ';
   ListLength = 10;
type NameType = packed array [1 .. 8] of char;
var Names: Array[l .. ListLength] of Nametype;
   Pivot: Nametype;
   LocationFound: Boolean;
   J,M,N: Integer;
{ GetName - это процедура чтения отдельного имени}
procedure GetName(var Name: NameType);
var J: Integer;
begin J := 1;
   repeat read(Name[J]); J := J+l; until (J > 8) or eoln;
   readln;
end;
begin
{Сначала вводим имя с клавиатуры}
   for J := 1 to ListLength do
      begin Names[J] := Blanks; Getname{Names[J]) end;
{Сортируем список}
   N := 2;
   repeat
      Pivot := Name[N];
      M := N - 1;
      LocationFound := false;
      while (not LocationFound) do
         if Names[M] > Pivot
            then
               begin
                  Names[M+l] := Names[M];
```

```
                M := M-l;
                if M = 0
                   then LocationFound := true;
             end
          else LocationFound := true;
       Names[M+l] := Pivot;
       N := N + 1
   until N > ListLength;
{ Теперь выводим отсортированный список на экран }
   for J := 1 to ListLength do writeln (Names[J])
end.
```

Рис. Г.6. Пример программы на языке Pascal

Язык Java. Java – это объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems в начале 90-х гг. Его разработчики много позаимствовали из языков C и C++. Будучи новым языком, Java еще не подвергался стандартизации. Действительно, язык Java все еще находится на стадии эволюции. Однако многие восторженно относятся к этому языку, поскольку он обещает стать стандартом для тех программ, которые известны как "аплеты Java" и которые можно передавать через Internet в виде выполняемых модулей и запускать на любой клиентской машине. Благодаря такой способности статичные по своей природе гипертекстовые документы можно заменить динамичными программами, с которыми пользователь сможет взаимодействовать непосредственно.

Пример программы на языке Java представлен на рис. Г.5. Обратите внимание на сходство языков Java и C++.

Язык Pascal назван в честь французского математика и изобретателя Блеза Паскаля (Blaise Pascal) (1623 – 1662). Разработанный Никлаусом Виртом (Niklaus Wirth) в 1971 г., этот язык подвергался многочисленным более поздним улучшениям, в частности особое внимание было уделено типам данных, созданным в дополнение к структурам, разработан синтаксис со свободным форматом, и добавлены многочисленные управляющие структуры. Сегодня Pascal используется в основном при обучении компьютерным наукам, поскольку способствует выработке у обучающихся организованного подхода к разработке программ. Пример программы на языке Pascal представлен на рис. Г.6.

СПИСОК ЛИТЕРАТУРЫ

1. Брукшир, Дж. Глен. Введение в компьютерные науки. Общий обзор, 6-е изд. / Дж. Глен Брукшир. – М. : Издательский дом "Вильямс", 2001. – 688 с.
2. Симонович, С.В. Информатика : базовый курс / С.В. Симонович и др. – СПб. : Питер. 2002. – 640 с.
3. Острейковский, В.А. Информатика : учебник для вузов / В.А. Острейковский – М. : Высшая школа, 1999. – 511 с.
4. Каймин, В.А. Информатика : учебник для вузов / В.А. Каймин. – М. : ИНФРА-М, 2000. – 232 с.
5. Могилев А.В. Информатика : учебное пособие для пед. учеб. заведений / А.В. Могилев и др. ; под ред. Е.К. Хеннера. – М. : Академия, 2000. – 816 с.
6. Информатика : энциклопед. словарь для начинающих / под общ. ред. Д.А. Поспелова. – М. : Педагогика Пресс, 1994. – 352 с.
7. Экономическая информатика / под ред. П.В. Конюховского, Д.Н. Колесова. – СПб. : Питер, 2000. – 560 с.
8. Информатика для юристов и экономистов / под ред. С.В. Симоновича. – СПб. : Питер, 2001. – 688 с.
9. Информатика : учебник для вузов / под ред. Н.В. Макаровой. – 3-е изд., перераб. – М. : Финансы и статистика, 2001. – 768с.
10. Жаров, А.В. "Железо" IBM 99 или все о современном компьютере. – 6-е изд., испр. и доп. / А.В. Жаров. – М. : Микро-Арт, 1999. – 352 с.
11. Компьютерные сети : учебный курс ; пер. с англ. – 2-е изд., испр. и доп. – М. : Изд. отдел "Рус. редакция" ТОО "Channel Traing Ltd", 1998. – 696 с.
12. Бекон, Д. Операционные системы / Д. Бекон, Т. Харрис. – СПб. : Питер, 2004. – 800 с.
13. Олифер, В.Г. Компьютерные сети: Принципы, технологии, протоколы : учебное пособие для вузов / В.Г. Олифер, Н.А. Олифер. – СПб. : Питер, 2001. – 672 с.
14. Анин, Б.Ю. Защита компьютерной информации / Б.Ю. Анин. – СПб. : БХВ-Санкт-Петербург, 2000. – 384 с.
15. Епанешников, А.М. Программирование в среде TurboPascal 7.0 / А.М. Епанешников, В.А. Епанешников. – М. : Диалог-МИФИ, 1995. – 288 с.
16. Системный анализ в информационных технологиях / Ю.Ю. Громов, Н.А. Земской, А.В. Лагутин, О.Г. Иванова, В.М. Тютюнник. – Тамбов : Изд-во Тамб. гос. техн. ун-та, 2004. – 176 с.
17. Немнюгин С.А. TurboPascal : практикум / С.А. Немнюгин. – СПб. : Питер, 2001. – 256 с.
18. Решение инженерных и экономических задач на языке С++ / Ю.Ю. Громов, С.И. Татаренко, В.М. Тютюник, А.В. Лагутин, О.Г. Иванова. – Тамбов : Изд-во МИНЦ, 2003. – 312 с.
19. Орлов, С. Технология разработки программного обеспечения / С. Орлов. – СПБ. : Питер. 2002. – 464 с.

ОГЛАВЛЕНИЕ

Для заметок